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

Zheng Zhang, Marcelo Cintra, and Josep Torrellas

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, data must be carefully laid out in memory for locality and application working sets carefully tuned for caches. It has been claimed that this programming effort is less necessary in hardware COMA machines like Flat-COMA thanks to automatic line-based data migration. Unfortunately, Flat-COMA is complex to design. Consequently, we would like a machine as programmable as Flat-COMA, as simple as plain CC-NUMA, and that outperforms both. This paper presents our proposal: *Excel-NUMA* (EX-NUMA). The idea is to exploit the fact that, after a memory line is written and cached, the storage that kept the line in memory is unutilized. We use that storage to temporarily hold remote data displaced from the local caches. This enables automatic data migration, like in Flat-COMA, enhancing programmability. The hardware required to manage the system is a simple, local module added to a CC-NUMA; the global cache coherence protocol is not changed. Simulations of Splash2 applications show that EX-NUMA outperforms CC-NUMA and Flat-COMA in every single application and eliminates most of the conflict misses.

Index Terms—Shared-memory multiprocessors, NUMA organizations, cache-coherence protocols, caches, performance evaluation.

1 Introduction

Hardware-coherent scalable shared-memory multiprocessors are hoped to deliver abundant computing power without surrendering much programmability. While machines of this type from Convex [1], Sequent [5], or Silicon Graphics [6] are becoming popular, some users 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 data in memory so that it is local to the processors that will use them most and tune the working sets to fit in caches. The goal is to minimize remote memory accesses.

To reduce remote accesses, several of the new machines 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.

It has been claimed that hardware COMA machines like Flat-COMA [7] minimize this programmability problem. In Flat-COMA, memory lines automatically migrate to the 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 attractive because, by supporting migration at a memory-line grain size, it does not have the page-related overheads that a software incarnation of COMA, like Simple-COMA, may suffer. For example, Simple-COMA may suffer from page unmapping, copying, and remapping overheads induced by internal page fragmentation.

Unfortunately, Flat-COMA is complicated to design. Its cache coherence protocol must ensure that last copies of lines are not lost [7]. This issue introduces corner cases in the protocol. In addition, some types of data access patterns do not use the COMA memory as well as they use a plain remote cache [11]. As a result, the performance may suffer.

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 line-based 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. We propose *Excel-NUMA* (EX-NUMA).

The idea is to use the fact that, after a memory line is written and cached, the copy in main memory becomes stale and its storage is unutilized. We can use that storage to temporarily hold remote data that is being displaced from the local caches. This enables automatic data migration, like in Flat-COMA, enhancing programmability. This remote data in local memory can be managed with a simple, local module that we add to a NUMA-RC; the global cache coherence protocol does not need to be changed. Simulations of Splash2 applications show that EX-NUMA outperforms NUMA-RC and Flat-COMA in every single application and eliminates most of the 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,

- *Z. Zhang is with Hewlett-Packard Laboratories, 1501 Page Mill Rd., M/S 3U-7, Palo Alto, CA 94304. E-mail: zzhang@hpl.hp.com.*
- *M. Cintra and J. Torrellas are with the University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801. E-mail: {cintra, torrella}@cs.uiuc.edu.*

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 108234.

Section 5 evaluates the baseline EX-NUMA, Section 6 evaluates enhancements to EX-NUMA, and Section 7 considers related work.

2 A Model for NUMA-RC and Flat-COMA

To motivate EX-NUMA, we compare a NUMA-RC and a Flat-COMA such that they have the same L1 and L2 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 remote cache in NUMA-RC and the extra memory for data replication in Flat-COMA. We neglect the small difference in tag space. In the rest of the paper, we refer to Flat-COMA as COMA.

Memory accesses can be *loc*, *remote-cold*, *remote-coh*, or *remote-conf* [11]. *Loc* are those satisfied by the local caches 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 at the expense of *loc* accesses.

Usually, *remote-conf* accesses are significant only if the remote data in the working set of a thread is larger than *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*) [11]. *Repl* occurs when a line is accessed in a read-mostly manner by several processors. *Mig* occurs when a line allocated remotely is accessed largely by one processor at a time and can be read-mostly (*MigR*) or read-write (*MigRW*).

Mig is key to distinguishing COMA from NUMA-RC. Ideally, a COMA's *MemOvhd* only needs to fit the *Repl* data: The *Mig* data can simply use up in one attraction memory the same space that it frees up in another one. A NUMA-RC's *MemOvhd*, however, needs to house both the *Repl* and the *Mig* data. Consequently, ideally, we would start seeing *remote-conf* accesses only when:

$$\mathrm{COMA}: \quad \mathrm{Repl} > \mathrm{MemOvhd}$$

$$\mathrm{NUMA-RC}: \mathrm{Repl} + \mathrm{Mig} > \mathrm{MemOvhd}$$

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, because the data automatically migrates.

Real caches, however, suffer conflicts, even with small working sets. Interestingly, NUMA-RC can utilize *MemOvhd* better and, therefore, suffer fewer conflicts than COMA. This results from the limited associativity of attraction memories. Indeed, consider 4-way set-associative attraction memories and remote caches and a *MemOvhd* of 25 percent of all memory. In COMA, each set in the attraction memory has one free entry. Consequently, the *MemOvhd appears* like a direct-mapped cache. However, the *MemOvhd* in NUMA-RC is 4-way set-associative. As a result, NUMA-RC suffers fewer *remote-conf* accesses. This effect particularly affects the data with the highest memory appetite, namely *Repl*. Overall, therefore, while COMA will likely have fewer *remote-conf* accesses if *Mig* dominates, NUMA-RC will likely have fewer *remote-conf* accesses if *Repl* dominates [11].

Note, however, that the performance difference between COMA and NUMA-RC is not only determined by the relative number of *remote-conf* accesses. It is also affected by how much processor stall time is induced by the average *loc*, *remote-cold*, *remote-conf*, and *remote-coh* accesses as well. The average remote access latency tends to be higher in COMA than in NUMA-RC [11]. The reason is that the fraction of remote accesses that involve three-node hops, instead of two, is higher in COMA than in NUMA-RC. These additional three-hop COMA transactions may be caused by accesses to data that used to be in its home memory and was displaced due to attraction memory conflicts. Additionally, they may be caused by accesses to data that COMA did not push to its home when NUMA-RC did. Specifically, in a read miss on data that is exclusive in a second node, our protocol updates the home node's memory in NUMA-RC, but not in COMA. We do not update home in COMA to minimize conflicts in the home memory.

Aside from these performance issues, COMA has a higher design complexity than NUMA-RC because its cache coherence protocol has to ensure that last copies of lines are not lost [7]. This issue introduces corner cases in the protocol.

Overall, we would like an architecture that combines all the advantages of both COMA and NUMA-RC: programmable as COMA, simple as NUMA-RC, and that handles the data in the best way, namely *Mig* as COMA and *Repl* as NUMA-RC. Our proposal is called *Excel-NUMA* (EX-NUMA).

3 The EX-NUMA Architecture

3.1 The Concept of EX-NUMA

The memory of a CC-NUMA is underutilized. Indeed, when a processor writes a datum, its cache gets a copy of the memory line, while the memory is left with a stale copy that will never be used. Any stale line left in memory after being written by either the *local* or a *remote* processor we call a *cell*. The up-to-date copy of the line, resident in a cache, we call the *owner* of the cell. The idea behind EX-NUMA is that each node records the cells that exist in its memory and temporarily uses them to house remote data that gets displaced from its cache. When the owner finally returns, either from this node's cache or from another node, the cell is destroyed. If the line stored in the cell is in exclusive state, it is sent to its own home.

The idea described allows effective support of *Mig* data. Consider a two-processor machine with six *Mig* arrays (Fig. 1a). Cell creation is shown in Fig. 1b, where processor *P1* writes to *2* and *P2* to *1*. Automatic migration

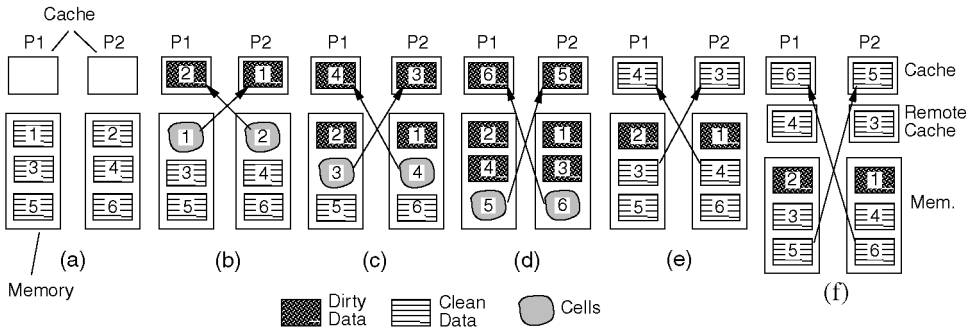


Fig. 1. Basic operation of EX-NUMA. The figures correspond to a 2-processor EX-NUMA with arrays *1* to *6* allocated in memory.

occurs in Fig. 1c, when *P1* writes to and brings to its cache array *4* while, at the same time, displacing *2* and saving it in its cells. *P2* does a similar operation for *3* and *1*. Finally, full migration completes in Fig. 1d, when *P1* 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.

Based on its ability to migrate *all* the data in a memory module to another one, EX-NUMA 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: It can only replicate as much data as it fits in its caches. This is shown in Fig. 1e, a continuation of Fig. 1b after processor *P1* reads *4* and *P2* reads *3*. However, if we add remote caches to EX-NUMA, we do not decrease its ability to handle *Mig* data, while we enable it to handle *Repl* data as effectively as NUMA-RC. For example, Fig. 1f shows Fig. 1e after *P1* reads *6* and *P2* reads *5*.

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

$$\mathrm{Repl} + \mathrm{MigR} > \mathrm{MemOvhd}$$

MigR is, in fact, relatively uncommon [9]. Furthermore, Section 3.4 shows that EX-NUMA can be easily enhanced to handle *MigR* as well. Therefore, EX-NUMA can behave very much like COMA for *Mig* data.

3.2 The Implementation of EX-NUMA

The functionality described is supported 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 simple baseline implementation, the RDT keeps the cell pointers in a DRAM table. Fig. 2a shows the resulting organization of an EX-NUMA node. Section 3.4 presents a more advanced RDT implementation.

The RDT records the location and state of all the cells present in the local memory. The cells can be thought of as a dynamic extension of the remote cache. When a remote line is brought into the L2 cache in state nonexclusive, the RDT saves a copy in one of the cells or in the remote cache. This is done because that line may be later displaced silently from the L2 cache. When a remote line in state exclusive is displaced from the local L2 cache, the RDT tries to save it in one of the cells or in the remote cache. Later, if the L2 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 L2 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 L2 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 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 (Fig. 2b). A fraction of the RDT entries are the tag array of the remote cache. 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 per memory line. Each RDT entry contains an address tag and state fields. The tag holds the address tag of the remote line stored in the cell or remote cache entry. 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 line does not have a cell. *Invalid* means that the location has an empty cell, while *valid* and *exclusive* mean that the location has a cell filled with a remote line in one of these states.

To locate an entry in the RDT, we use the address bits of the remote line. To use cells better, the RDT is set-associative. In that case, a given RDT entry can point to one of several memory locations and, therefore, a given remote line can be inserted in one of several such locations. Fig. 2c shows the mapping of a 4-way set-associative RDT.

The state diagram for RDT entry states is shown in Fig. 2d. 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 L2 cache in state nonexclusive or a remote line in state exclusive is displaced from the L2 cache, the RDT tries

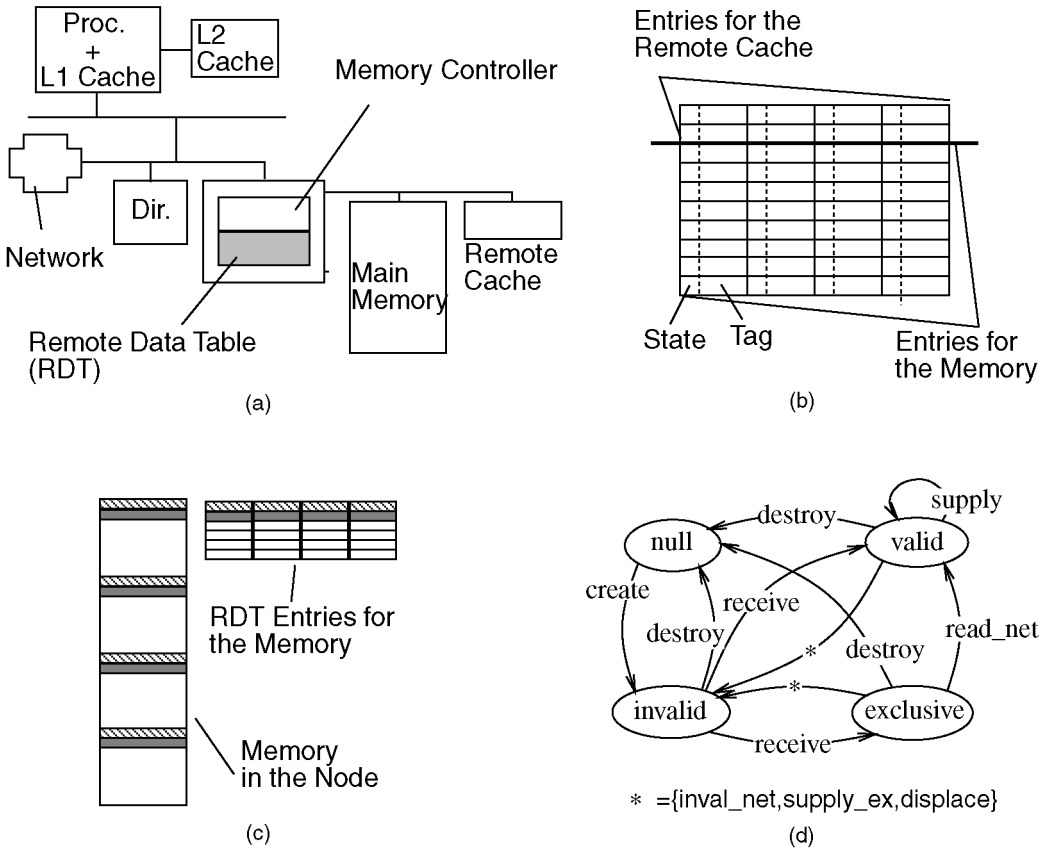


Fig. 2. Baseline EX-NUMA: node organization (Chart (a)), 4-way set-associative RDT (Chart (b)), mapping between the RDT entries for the memory and memory locations (Chart (c)) and state diagram for RDT entry states (Chart (d)).

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 one of 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 [9]. 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 L2 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/COMA

To assess the performance of EX-NUMA, we use the model of Section 2. As discussed before, if all the data is *Repl*, EX-NUMA has as few *remote-conf* accesses as NUMA-RC because it reverts to it. If all the data is *MigRW*, EX-NUMA can have as few *remote-conf* accesses as COMA because it supports large-scale line-level data migration. Line migration in EX-NUMA appears trickier than in COMA because it needs other transactions to first create cells in the destination locations. However, we show in Section 5 that migration in EX-NUMA proceeds largely unimpeded. Finally, if all the data is *MigR*, EX-NUMA reverts to NUMA-RC and, therefore, has more *remote-conf* accesses than COMA. However, *MigR* data has little weight [9] and, in addition, Section 3.4 shows that it can also be handled by enhancing EX-NUMA. Finally, in all cases, the average stall time of a remote access in EX-NUMA is similar to NUMA-RC's and smaller than in COMA. The reason is that EX-NUMA does not suffer the line displacements from home memories that cause three-hop transactions in COMA. More detail can be found in [10].

The DRAM-based RDT does not need to slow down EX-NUMA relative to NUMA-RC. In NUMA-RC, after an L2 cache miss, the DRAM tags of the remote cache are accessed in parallel with the remote cache data. In EX-NUMA, we double the associativity of the search. The RDT entries for the remote cache and the remote cache are accessed in

parallel with the RDT entries for the memory and the memory. This system can be organized such that multiple requests can be pipelined, therefore reducing the resource occupancy.

The memory required for the baseline RDT is modest. 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 a memory line is 32, 64, or 128 bytes, the ratio between the size of the RDT dedicated to memory and the memory size is 3.9 percent, 1.9 percent, and 1 percent, respectively. Adding this much DRAM is not very expensive. Furthermore, there are two schemes to reduce the DRAM needs. First, the RDT can have fewer entries dedicated to memory than memory lines. In this case, each RDT entry can identify one cell in a group of several memory lines. Consequently, we risk wasting cells [10]. The second scheme is the advanced RDT design discussed in Section 3.4, which practically eliminates all DRAM needs.

Finally, the design complexity of EX-NUMA is modest because all the modifications required to produce an EX-NUMA out of a NUMA-RC are local. In contrast to COMA, there are no changes to the global cache coherence protocol [9].

3.4 Advanced Implementation Issues

The baseline EX-NUMA can be enhanced in several ways [9]. We briefly summarize them here. First, given that the RDT entries dedicated to memory and to the remote cache work in the same way, we can make the boundary that separates the two types of entries flexible. For example, if the application does not need much physical memory, we can increase the size of the remote cache at the expense of the memory. One way to do this "logically" is to mark up all the RDT entries that are pointing to lines in unused memory pages as if these lines all had cells. We can then store remote data in all these unused memory pages.

To support *MigR* data, we can create a cell every time that a home node receives a read request for a line that is currently not in any cache. 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 support *MigR* data and is beneficial to *MigRW* data because cells are created as soon as the first read occurs. However, it hurts *Repl* data because a subsequent read by a second processor needs to fetch the data from the first reader processor [9].

Another way to increase the effectiveness of cells is to minimize the chances that, when an owner line is written back, it displaces the line in its cell. Such an event could cause another cell destruction downstream. To cut short any such chain of cell destruction, we can treat the returning owner as if it were a remote line displaced from the secondary cache. As such, it can be written into any location pointed to by the corresponding RDT sets (memory cells or remote cache) as determined by the receiving algorithm. If the place where the owner is saved is not its default location, we mark the new location as its new back-up location. This is called *location relaxation*.

There are several ways of implementing location relaxation. A simple and inefficient approach is to add one bit called *moved* to each directory entry. This bit indicates whether the line associated with the entry has undertaken location relaxation. In addition, the state field of each RDT entry can take an extra state, namely *relaxed_location*. With this support, assume that an owner is written back and, since its RDT entry i is full, we 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 *relaxed_location*, respectively. In addition, the *moved* bit in the incoming line's directory is set. The next request that accesses the line finds the *moved* bit set in the directory and, while accessing the RDT, finds the correct tag, and reads the line from entry j. If the line is written to, an empty cell is created in its new location, the *moved* bit is reset, and, therefore, the line is not under location relaxation any more. When the line is written back again, we try to save it in its original location. A more refined location relaxation scheme that involves swapping the contents of directory entries is described in [9].

A final performance optimization is high RDT associativity. With high associativity, cells retain more data, which reduces the pressure on the remote cache and results in fewer conflict misses. To tolerate the higher RDT busy time, we pipeline the RDT and memory access [9], [10].

We can also reduce RDT overheads. We chose the baseline RDT design to minimize the modifications to a NUMA-RC. However, if we are willing to redesign the directory to integrate it with the RDT, we can eliminate practically all of the RDT memory requirements. Specifically, the tags and state of the RDT entries dedicated to memory can be stored in the directory entries. This is possible if the machine uses a pointer-based directory scheme with invalidations [4]. In such schemes, after a memory line is written, the corresponding directory entry contains only one pointer. Consequently, we can use the unused directory bits to store the RDT tag and state of the incoming line. A design is presented in [9].

4 Experimental Setup

We perform execution-driven simulations of 32-node CC-NUMA, NUMA-RC, COMA, and EX-NUMA architectures using Tangolite [3]. We refer to the CC-NUMA as NUMA. Each node includes a 200 MHz processor, two levels of cache, a memory controller, and a portion of the global memory and directory (Fig. 2a). The on-chip L1 cache is direct-mapped and has 16-byte lines. The L2 cache, remote cache, and attraction memory are 4-way set-associative and have 32-byte lines. The RDT is 4-way set-associative for the cells and for the remote cache, and has as many entries for memory as there are memory lines. The remote cache in NUMA-RC and EX-NUMA and the extra memory in COMA have the same size. The small difference in tag space is neglected. Caches are kept coherent with the DASH protocol [4]. The memory bus is 64-bit wide and supports split transactions. The memories, RDTs, and remote caches are pipelined and deliver the first word in 12 clocks. There is a global network with a fixed two-node latency of 100

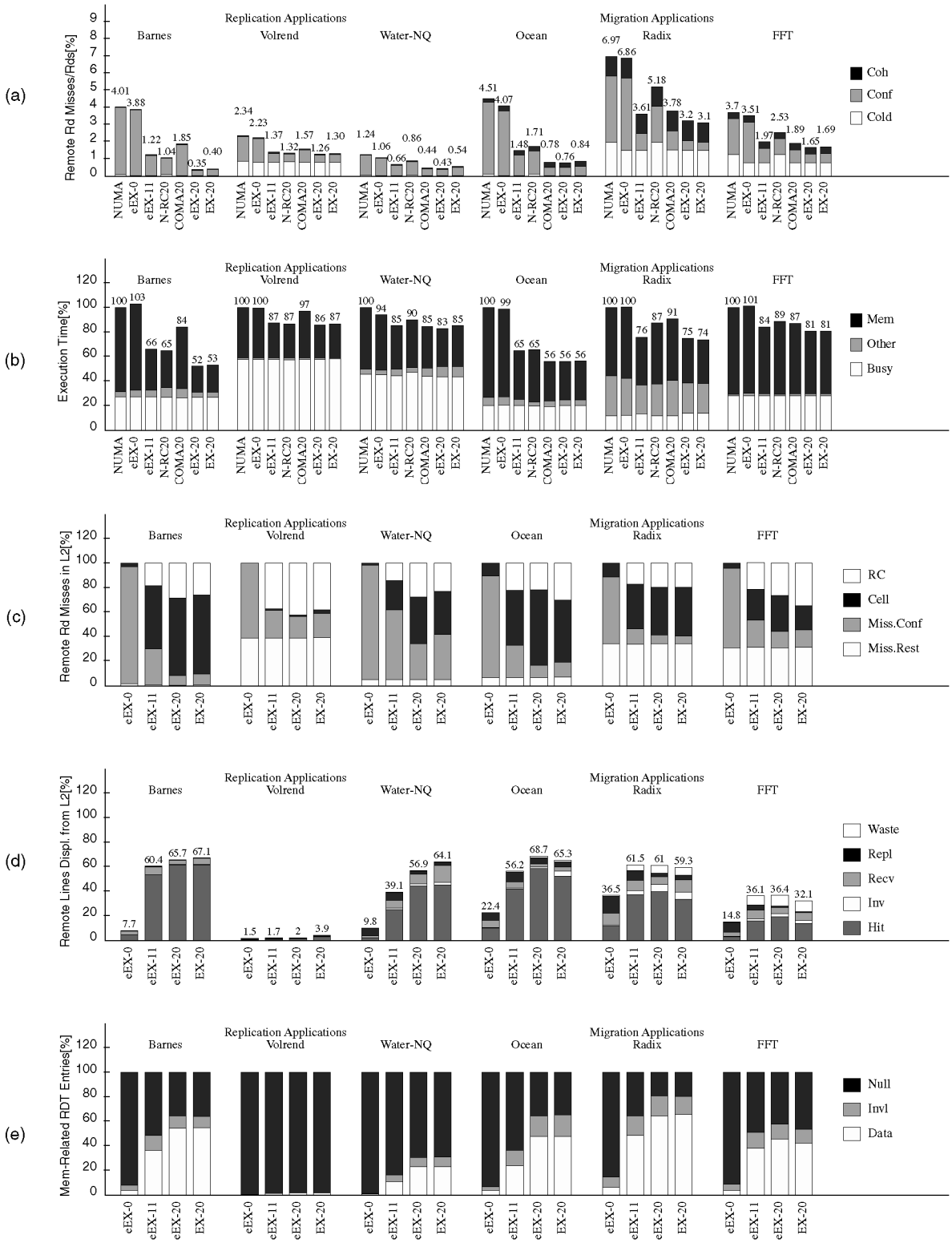


Fig. 3. Results of the evaluation. For several architectures, Chart (a) shows the fraction of reads not satisfied in the local node and Chart (b) the execution time. For EX-NUMA, Charts (c)-(e) examine cell behavior: reads to remote data not intercepted by the L2 caches (c), memory lines with remote data displaced from the L2 caches (d), and state of RDT entries dedicated to memory (e).

cycles. The unloaded round-trip latency to access the L1 cache, L2 cache, local memory, and remote memory in two and three hops is 1, 4, 39, 249, and 351 cycles, respectively. The remote cache has the same latency as the local memory. The machines use release consistency. All contention is modeled except in the network, where it is neglected. The NUMA and NUMA-RC allocate pages in first-touch after the parallel section starts. For COMA and EX-NUMA, since the allocation should not matter much, we use round-robin.

TABLE 1
Characteristics of the Applications Used

Application	Problem Size	Footprint (MB)	L2 Cache (KB)	Dominant Data Type
Barnes	16K bodies	3.8	8	*Repl*
Volrend	Head	22.9	32	*Repl*
Water-NQ	512 molecules	7.0	8	*Repl*
Ocean	258 × 258 grid	15.5	32	*Mig*
Radix	256K keys	2.8	8	*Mig*
FFT	64K points	3.7	8	*Mig*

Applying round-robin to NUMA and NUMA-RC does not change the results qualitatively.

To expose the performance differences between architectures, we dimension the memory hierarchy for each application so that the size of the per-processor working set is comparable to or larger than *MemOvhd*. 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*, L2 cache, and L1 cache, respectively. The cache sizes are rounded to a power of two. The resulting *MemOvhd* is, therefore, 20 percent of the total memory.

The Splash2 parallel applications executed are listed in Table 1, together with their data footprint size, the L2 cache used, and their major data type. *Repl* dominates in *Barnes*, *Volrend*, and *Water-NQ*. While in *Barnes* and *Volrend*, the *Repl* data is a tree structure that gets scattered around and uses COMA's *MemOvhd* ineffectively, in *Water-NQ*, it is a 1D array of molecules that uses COMA's *MemOvhd* well. *Mig* dominates in *Ocean*, *Radix*, and *FFT*. The *Mig* data is *MigRW* in *Ocean*, often *MigRW* in *FFT*, and often *MigR* in *Radix*. The *Mig* behavior in *Radix* and *FFT* is harder to exploit because processors change working sets frequently, creating transient periods.

5 Evaluating the Baseline EX-NUMA

This section summarizes the evaluation in [10]. We compare a NUMA and three machines with 20 percent *MemOvhd*: NUMA-RC (*N-RC20*), COMA (*COMA20*), and EX-NUMA (*EX-20*). For reference purposes, we also consider an ideal EX-NUMA where the L2 cache unrealistically informs the RDT of displacements of lines in state nonexclusive. In this case, the remote cache and cells do not need to keep copies of such lines—their data is exclusive with that in L2. As a result, the local memory hierarchy can hold more remote data. We examine this case for 0 percent, 11 percent, and 20 percent *MemOvhd* (*eEX-0*, *eEX-11*, and *eEX-20*, respectively, where *eEX* means exclusive EX-NUMA). For these seven architectures, we compare the fraction of read accesses not satisfied in the local node (Fig. 3a) and the execution time (Fig. 3b). The remote read misses are broken down into start-up (*Cold*), conflict (*Conf*), and coherence (*Coh*), while the execution time is divided into execution of instructions (*Busy*), stall due to memory accesses (*Mem*), and other stalls (*Other*).

Fig. 3b shows that, compared to *NUMA*, architectures with extra caching space like *N-RC20* and *COMA20* are significantly faster. As expected, *N-RC20* tends to perform better for *Repl* applications, while *COMA20* tends to be better for *Mig* ones. The exceptions are *Water-NQ*, where *COMA20* does well because the *Repl* data is regular and uses *MemOvhd* effectively, and *Radix*, where *COMA20* is slower because the *Mig* behavior of the data is weakened by processors frequently changing working sets. In all cases, *EX-20* runs faster than *N-RC20* and *COMA20*. *EX-20* adapts to both *Repl* and *Mig* data. On average, it runs 10 percent and 12 percent faster than *N-RC20* and *COMA20*, respectively. While this figure may seem modest, it is close to the upper bound possible. Indeed, Fig. 3a shows that *EX-20* has a very low remaining *Conf* miss rate. The speed-up has been accomplished across all types of applications and with a simple architecture. Finally, the similar performance of *eEX-20* and *EX-20* indicates that we do not need an unrealistic fully exclusive system. The low speed of *eEX-0* suggests that the lack of a remote cache strongly inhibits cell exploitation.

We now examine EX-NUMA cell behavior. Consider the reads to remote data that are not intercepted by the L2 caches. Fig. 3c breaks them into those that are intercepted by the remote cache (*RC*), those intercepted by cells (*Cell*), and those that cause a remote miss. The latter can be caused by conflicts (*Miss.Conf*) or not (*Miss.Rest*). Remote caches and cells can only eliminate conflict misses. The figure shows that the cells in *eEX-0*, which do not have the help of a remote cache, do not eliminate many conflict misses. With a remote cache, *eEX-11* and *eEX-20* eliminate many more conflict misses. A remote cache has a multiplicative effect: It intercepts requests and is a catalyst to create more cells that intercept requests as well. The exception is *Volrend*, where the *Mig* data is largely read-only and, therefore, unable to create cells. Comparing *eEX-20* to *EX-20*, we confirm that ideal exclusivity makes little difference.

Fig. 3d 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 L2 caches. They can either be absorbed by cells, by the remote cache or, in *EX-20*, displaced silently. The height of the bars in the figure shows the fraction that are absorbed by cells. In turn, these lines can be: eventually reused 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 intercepted by cells. In *eEX-20*, an average of 50 percent of the lines are. Of these, the fraction that are reused (*Hit*) depends on the size of the remote cache. For *eEX-20* and *EX-20*, the fraction that are reused (*Hit*) is 70 percent on average. This is good news. When the reuse is low (*Radix* and *FFT*), it is because the *Mig* data moves from one processor to another frequently, inhibiting data reuse.

Finally, Fig. 3e shows the state of the RDT entries dedicated to memory at the end of the program. A memory location can have a cell with data (*Data*), an empty cell (*Invl*), or no cell (*Null*). In *EX-20*, an average of 50 percent of the memory locations have cells. Of those, about 80 percent contain data. These two numbers show that EX-NUMA-style migration works. We also point out that we have not seen line write backs that trigger long chains of cell destruction across the machine. Indeed, the presence of a

remote cache, the associativity of the RDT, and an RDT receiving algorithm that favors replacements of nonexclusive lines, all make EX-NUMA very stable [10].

6 Evaluating Enhancements to EX-NUMA

To fully assess the EX-NUMA organization, we now evaluate the performance impact of some of the enhancements discussed in Section 3.4. To save space, our discussion is brief. For more details, see [9]. In the following, all the numbers quoted are relative to the baseline EX-NUMA.

We first consider supporting *MigR* data by creating a cell on every read to an uncached line. As indicated in Section 3.4, this optimization has negative side effects on *Repl* data. Moreover, *MigR* data is relatively insignificant. As a result, while this optimization improves the performance of some applications, the average impact is small.

The location relaxation of Section 3.4 allows the EX-NUMA to retain more cells. While some applications run up to 6 percent faster, the average application runs 3 percent faster.

We examine RDT associativities of 2, 4, and 32. Recall that the baseline EX-NUMA has an RDT associativity of 4. This means that, for each RDT access, we check four RDT entries dedicated to memory and four dedicated to the remote cache. Our results show that the largest performance gains occur while going from 2- to 4-way set-associative RDTs. On average, the applications run 6 percent faster. Increasing the associativity from 4 to 32 has a large impact in some applications. For example, *Ocean* runs 11 percent faster. On average, the applications run 5 percent faster. However, the design of such a fast, high-associative circuitry is challenging.

Finally, we examine the effect of having fewer RDT entries dedicated to memory than lines in memory. Each RDT entry can now record a cell from one of several memory locations by including a few more bits that specify which location it actually points to. This organization may help a line find a cell because each RDT entry maps to a group of memory locations. However, the RDT loses the ability to record all cells, since only one cell per group can be recorded at any one time. In our experiment, we simulate *EX-20* with only 50 percent of the baseline RDT entries dedicated to memory. Each RDT entry handles two consecutive memory locations. The results show that *Barnes*, *Water-NQ*, and *Ocean* are slowed down by 2-4 percent. On average, however, the six applications run only about 1 percent slower.

7 Related Work

Perhaps the closest work is R-NUMA, proposed by Falsafi and Wood [2] 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. Data migrates at the page granularity, which is less effective. Switching between CC-NUMA and Simple-COMA is performed at the coarse grain of a page at a time, and requires operating system involvement via an interrupt. Finally, the interrupt is triggered 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 overhead. No run-time feedback or interrupts are necessary.

Other related work is operating system-induced migration and replication of pages based on run-time feedback, e.g., [8]. This technique has some of the effects of EX-NUMA, although migration is only supported at page granularity and there is operating system overhead.

8 Discussion and Concluding Remarks

EX-NUMA attempts to build on the good points of both Flat-COMA and NUMA-RC: It 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 *Mig* and *Repl* data access patterns well. It eliminates most of the conflict misses. Overall, EX-NUMA represents a good cost-performance design point, as we consider complexity, performance, and programmability.

EX-NUMA supports line-based automatic data migration without the costly Flat-COMA hardware. In theory, migration in EX-NUMA may be harder than in Flat-COMA. The reason is that the space that a node has available to hold remote *Mig* 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 met for the applications and page allocation policies analyzed. Furthermore, when they are not, the large remote cache can absorb the fluctuations. Therefore, EX-NUMA-style migration works well. In the future, we will examine applications that are less tuned for caches than Splash2, for example databases.

Acknowledgments

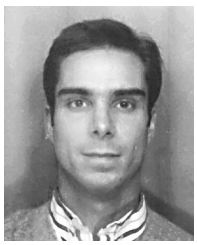
This work is supported by the U.S. National Science Foundation (NSF) under grants MIP-9457436, ASC-9612099, and MIP-9619351; DARPA DABT63-95-C-0097; and Intel and IBM. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] Convex Inc., "Convex Exemplar Description," http://www.convex.com/prod_serv/exemplar/exemplar.html.

[2] B. Falsafi, D. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 229-240, June 1997.
[3] S. Goldschmidt, "Simulation of Multiprocessors: Accuracy and Performance," PhD thesis, Stanford Univ., June 1993.
[4] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford Dash Multiprocessor," *Computer,* pp. 63-79. Mar. 1992.
[5] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 308-317, May 1996.
[6] Silicon Graphics Inc., "Origin Servers," http://www.sgi.com/Products/hardware/servers.
[7] P. Stenstrom, T. Joe, and A. Gupta, "Comparative Performance Evaluation of Cache-CoherentNUMA and COMA Architectures," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* pp. 80-91, May 1992.
[8] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Oct. 1996.
[9] Z. Zhang, "Architectural Alternatives to Reduce Remote Conflict Misses in Shared-Memory Multiprocessors," PhD thesis, Univ. of Illinois at Urbana-Champaign, Oct. 1996.
[10] Z. Zhang, M. Cintra, and J. Torrellas, "Excel-NUMA: Toward Programmability, Simplicity, and High Performance," Technical Report CSRD-1544, Univ. of Illinois at Urbana-Champaign, Nov. 1996.
[11] Z. Zhang and J. Torrellas, "Reducing Remote Conflict Misses: NUMA with Remote Cache versus COMA," *Proc. Third Int'l Symp. High-Performance Computer Architecture,* pp. 272-281, Feb. 1997.

Zheng Zhang received his BS degree in electrical engineering from Fudan University in 1987 and an MEng degree from the University of Texas at Dallas in 1992. He joined the Center for Supercomputing Research and Development (CSRD) at the University of Illinois at Urbana-Champaign as a research assistant in the spring of 1993 and received his PhD degree in electrical and computer engineering in 1996. He joined Hewlett-Packard Laboratories as a member of the technical staff in 1996. Dr. Zhang's primary interests are high-performance computer architectures. During his stay at CSRD, he conducted research on multistage networks, innovative techniques to reduce cache coherence traffic and misses in shared-memory multiprocessors, and the evaluation of shared-memory multiprocessors with commercial applications. He is the author of eight refereed papers in major journals and conference proceedings. His early research activities at HP Labs includes high-performance server architecture and high-availability memory systems. He is currently leading a team to develop the next generation I/O architectures. He has written more than 10 internal technical reports at HP Labs and filed four patents.



Marcelo Cintra received the BEng degree in electrical engineering and the MS degree in computer engineering from the University of Sao Paulo, Brazil, in 1992 and 1996, respectively. He is currently a PhD candidate in computer engineering at the University of Illinois at Urbana-Champaign. His research interests include scalable high-performance computer architecture and scientific and engineering applications.

Josep Torrellas received a PhD in electrical engineering from Stanford University in 1992. He is an associate professor in the Computer Science Department at the University of Illinois at Urbana-Champaign with a joint appointment in the Electrical and Computer Engineering Department. He is vice-chair of the IEEE Technical Committee on Computer Architecture (TCCA). In 1998, he was on sabbatical at the IBM T.J. Watson Research Center. He received the U.S. National Science Foundation (NSF) Research Initiation Award in 1993, the NSF Young Investigator Award in 1994, the C.W. Gear Junior Faculty Award, and the Xerox Award for Faculty Research from the University of Illinois in 1997. Dr. Torrellas' primary interests are new processor and memory technologies and organizations to build uni- and multiprocessor architectures. He is leading the Illinois Aggressive COMA multiprocessor project (I-ACOMA). He is the author of more than 60 refereed papers in major journals and conference proceedings. He has been a member of the organizing committees of many international conferences and workshops. Recently, he was the co-organizer of the Workshop on Computer Architecture Evaluation Using Commercial Workloads, the Workshop on Scalable Shared-Memory Multiprocessors, and is the general cochairman of the Sixth International Symposium on High-Performance Computer Architecture (HPCA). His web site is http://iacoma.cs.uiuc.edu/~torrella.