Enhancing Memory Use in Simple Coma: Multiplexed Simple Coma[1]

Sujoy Basu and **Josep Torrellas**

Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801
basu,torrella@cs.uiuc.edu
http://iacoma.cs.uiuc.edu/iacoma/

Abstract

Scalable shared-memory multiprocessors that are designed as Cache-Only Memory Architectures (Coma) allow automatic replication and migration of data in the main memory. This enhances programmability by hopefully eliminating the need for data distribution strategies and page migration schemes. Recently, a variant of Coma called Simple Coma has been proposed as a lower-cost alternative to hardware-intensive systems like Flat Coma. However, we find that Simple Coma is quite slower than Flat Coma. The main reason is the high page mapping, unmapping, and transfer overhead caused by memory fragmentation in Simple Coma.

This paper proposes a solution to the memory fragmentation problem that we call Multiplexed Simple Coma. The idea is to allow multiple virtual pages to map into the same physical page at the same time, therefore compressing the page working set of the application. Multiplexed Simple Coma requires very little support over Simple Coma and reduces its execution time by about 40%. We find that Multiplexed Simple Coma can be very easily implemented with off-the-shelf processors. In addition, there is no need to be selective when choosing what virtual pages are to share the same physical page. Overall, although Multiplexed Simple Coma is still slower than Flat Coma, since it is cheaper to implement, it represents a good cost-performance design point.

1 Introduction

Shared-memory multiprocessors provide a simple and easy to use computing environment that can deliver high performance. However, as these machines have grown in size, the memory subsystem has become more complicated and distributed. As a result, accesses to remote memory modules now involve large latencies. This effect degrades the performance of many applications, especially those with working sets that overflow the caches.

One organization of shared-memory multiprocessors that addresses this problem is the Cache-Only Memory Architecture (Coma) [5, 9, 10, 13, 14]. In Coma, main memory is organized as a cache, known as attraction memory. The physical address of data is no longer associated with a fixed location in memory. Instead, the address is used to access the local attraction memory just as it is used to access the caches. In case the requested data is not present in the local node, a directory-based coherence protocol identifies the remote node that can supply the data. Then, space has to be allocated in the local node's attraction memory to accept the incoming data. Overall, therefore, data automatically replicates and migrates to the node that is using it. This decreases the frequency of costly remote memory accesses. Indeed, on a cache overflow, the local attraction memory is more likely to supply the data. Furthermore, programming becomes simpler because data distribution strategies and page migration schemes are largely unnecessary and there is less need to tune the working sets to fit in caches.

An example of a high-performance hardware-only Coma organization is Flat Coma [14]. In Flat Coma, the directory information for a line is kept in a fixed home node. Consequently, while memory lines are allowed to migrate freely, they can still be located quickly. Indeed, we only need to check the directory entry in the corresponding home node to determine where to find a copy of the line. This is unlike in older Coma machines like KSR1 [9] and DDM1 [5]. To reduce conflicts, the attraction memory in Flat Coma is made set-associative, and each entry is given a tag that contains state and address information for the line. When an incoming line needs to be stored in the attraction memory, the hardware may need to displace another line to make room. If the tag of the displaced line indicates that this is the last copy of the line, the hardware injects the line into another node. This is necessary to ensure that last copies of lines are never lost [8].

Recently, a variant of Coma called Simple Coma [13] has been proposed as a lower-cost alternative to hardware-intensive systems like Flat Coma. The idea is to migrate part of the complexity of Coma from hardware into software. In Flat Coma, some bits in the physical address are used to index the attraction memory and select the right set, while other bits in the address are compared against the tags of the lines in the set. This requires indexing hardware, tag comparators, and full tags for each line. In Simple Coma, this hardware is largely eliminated and the requested line is identified in memory simply by using the TLB or page table information. Of course, this forces the allocation unit of the data to be a page. Unfortunately, this may cause memory fragmentation. Whenever a node accesses a line of data, space has to be allocated in the local attraction memory for the entire page containing that line, even if no other line in the page will ever be used. This memory fragmentation causes the expansion of the working set. If the page working set is larger than the attraction memory, it may induce a high page mapping, unmapping and transfer overhead. The overall result, we have found, is low performance: Simple Coma is quite slower than Flat Coma.

To solve the memory fragmentation problem, this paper proposes an extension to Simple Coma called *Multiplexed Simple Coma.* The idea is to allow multiple virtual pages to map into the same physical page at the same time, therefore compressing the page working set of the application. Multiplexed Simple Coma requires very little support over Simple Coma and reduces its execution time by about 40%. We find that Multiplexed Simple Coma can be implemented with off-the-shelf processors. In addition, there is no need to be selective when choosing what virtual pages are to share the same physical page. Overall, although Multiplexed Simple Coma is still slower than Flat Coma, since it is cheaper to implement, it represents a good cost-performance design point.

This paper is organized as follows: Section 2 compares Simple Coma to Flat Coma; Section 3 presents Multiplexed Simple Coma; Section 4 discusses the experimental setup used for our evaluation; Section 5 evaluates the performance of the base Multiplexed Simple Coma; and Section 6 examines variations of the design.

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

2 Simple Coma Versus Flat Coma

Simple Coma [6, 13] migrates some of the support for data coherence from the hardware to the virtual memory manager in the operating system. Unlike in Distributed Virtual Shared Memory (DVSM) systems [2, 11], however, the unit of coherence in Simple Coma is still the memory line, and traditional implementations support a hardware cache coherence protocol similar to the one in Flat Coma. Overall, the hardware in Simple Coma is simpler than in Flat Coma at the cost of some operating system overhead. In this section, we examine the issues involved.

A Simple Coma, unlike a Flat Coma, uses nodes that do not share a common physical address space. Each node has its own full range of physical addresses. One consequence of this is that a message in Simple Coma needs to go through several address translations. As the message leaves a node, there is a translation from the local address to a common, global address used in the network. When the message reaches the destination node, another translation into the local address in that node is required. Unfortunately, all these translations are performed in hardware. This is expensive and must rely on fast TLB-like tables. Some of these tables are sometimes referred to as *Reverse TLB* tables [12]. Since these tables can overflow, they need to be backed up by page tables in the operating system.

A second consequence of the different physical address spaces is that Simple Coma is easier to build from commodity components like workstations and operating systems. A final consequence of having independent physical addresses in each node is that Simple Coma can use the TLB lookup step to locate the desired line in the attraction memory [13]. If, instead, we had a single physical address space like in Flat Coma, remote lines would have to be stored into fixed locations in the local attraction memory. Since several such lines can conflict for the same position, an extra tag check after the TLB lookup is necessary in Flat Coma. In Simple Coma, the operating system is free to map a page in any part of the local memory of a node and, therefore, the TLB lookup step is enough to locate the desired line.

This has three advantages. One of them is that it simplifies handling the replacement of last copies of data. Recall that, in Flat Coma, the cache coherence protocol must be extended to make sure that last copies of lines are not purged from the system [8]. This requires the handling of several corner cases which complicate the protocol. In Simple Coma, all replacements are handled by the operating system, irrespective of whether or not they handle last copies. Specifically, the last-copy problem is handled as follows. One copy of each different virtual page in the system is designated as the *master* copy. The operating system can discard a non-master copy of a page from a node by sending the lines in that page that are last copies to the node with the master copy of the page. If a master copy of a page needs to be replaced, the operating system makes another node accept the master copy.

The second advantage is that, when the attraction memory of Simple Coma is accessed, there are no address tags to check as in Flat Coma. The final advantage is that, since a given virtual page can be mapped to any physical page irrespective of its mapping in another node, the attraction memory effectively becomes fully associative at the page level. This should help reduce the number of conflicts in the attraction memory.

However, attraction memory space in Simple Coma must now be assigned in page-sized chunks. When a processor misses on a line of a remotely-mapped page, the operating system allocates physical memory for the entire page, even though only the line missed on is brought in. The rest of the lines remain invalid. This is in contrast to Flat Coma, which only allocates space for the line missed on. Flat Coma can do this because each line has an address tag. Unfortunately, since not all the memory lines in a page are likely to be accessed frequently, Simple Coma may suffer from memory fragmentation. It is possible that, if the application has little spatial locality, most pages in the attraction memory have only a few valid lines. The result will be a waste of memory.

This sub-optimal memory utilization combined with large working sets may result in working set overflow from the attraction memory. This may cause a high rate of page allocation, deallocation, and transfer, and result in performance degradation for two reasons. First, it may induce much operating system overhead. Second, every time that a page is deallocated to make room for another one, we may be displacing many useful lines and, therefore, induce future conflict misses.

To summarize, Simple Coma has two advantages and two disadvantages over Flat Coma. The first advantage is that nodes require less hardware and software custom design because they have separate physical addresses. The second advantage is that the hardware is simpler. This is because there is no need for hardware to support line replacements (of last copies or otherwise) from the attraction memory and for the checking of address tags in the attraction memory. One disadvantage of Simple Coma is that it needs special hardware to translate physical addresses to global addresses in the network and vice-versa. The second disadvantage is that it suffers from memory fragmentation, which may cause low performance. Overall, if a solution is found for Simple Coma's shortcomings, Simple Coma may be a good approach for relatively cheap systems.

3 Multiplexed Simple Coma

In this paper, we propose a solution to Simple Coma's problem of memory fragmentation. In this section, we discuss the basic idea, the algorithms implemented, and the hardware and operating system support required.

3.1 The Basic Idea

To gain insight into the magnitude of the memory fragmentation problem, we measured, for seven Splash2 applications [17], what fraction of a page is used. Table 1 shows, for each application, the fraction of a page that is used before replacement in Simple Coma with 70% memory pressure (*Real* column), and the fraction that would be used if replacements could be avoided by having an infinite attraction memory (*Infinite* column). The data corresponds to the average of all the shared pages in the application, 32-processor runs, 4 Kbyte pages and 64-byte lines. From the table we see that, in the real system, an average of only 17% of a page is used while, without replacements, up to 26% would be used. Clearly, we are not using the memory storage in the node efficiently and may run the risk that the working set overflows the attraction memory.

Table 1: Fraction of a page that is used before replacement in Simple Coma (*Real* column) and in a system with infinite attraction memory (*Infinite* column).

Application	Real	Infinite
Barnes	0.20	0.25
Cholesky	0.30	0.40
FFT	0.06	0.06
Ocean	0.20	0.21
Radix	0.06	0.11
WaterNsq	0.20	0.60
WaterSp	0.16	0.22
Average	0.17	0.26

If we could identify several virtual pages with only a few frequently-accessed lines such that these lines had largely different page offsets in the different virtual pages, we could force all of these virtual pages to share the same physical page. This

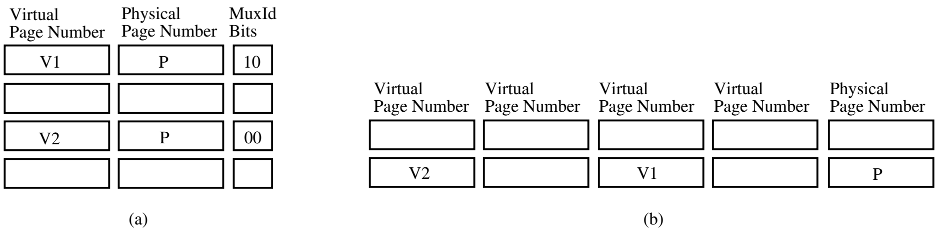


Figure 1: Organization of the TLB for a system that supports a degree of multiplexing of up to 4 (a). A second organization, which suffers from entry fragmentation, is shown in (b). In both cases, virtual page numbers *V1* and *V2* are mapped to physical page number *P*.

approach would reduce memory fragmentation. The resulting system we call *Multiplexed Simple Coma*. The different virtual pages mapped to the same physical page are said to be *multiplexed* into the the same physical page. For a given physical page, the number of virtual pages multiplexed into it at a given time is called the *degree of multiplexing* of the physical page.

Multiplexing several virtual pages into a single physical page requires simple architectural support. Suppose that we support the multiplexing of up to *NumMux* virtual pages per physical page. We need to add a tag of $log(NumMux)$ bits to each line in the caches and attraction memory. These bits can be placed together with the tag bits of the line in the caches and with the state bits of the line in the attraction memory. We call these bits the *MuxId* bits. For a given memory line, they specify the owner virtual page among the *NumMux* possible ones multiplexed into this physical page. In addition, $log(NumMux)$ bits are added to each entry in the page table and TLB. The resulting TLB organization is shown in Figure 1-(a). In this scheme, a separate TLB entry is used for every virtual page mapped to a given physical page. As a result, the same physical page number is replicated in several entries. We could use a different TLB organization, shown in Figure 1-(b), which eliminates this replication and does not need the *MuxId* bits. However, it will lead to TLB entry fragmentation because many of the virtual page number fields may remain unused. For a given TLB area, the organization in Figure 1-(a) will likely contain more useful entries. Besides, it can be implemented in an off-the-shelf processor (Section 3.3).

With this support, we access the memory system with both the *MuxId* bits and the physical address. An access in Multiplexed Simple Coma should not take any longer than one in Simple Coma. The *MuxId* bits and line state bits are looked up in tandem with the memory action. Implementation-wise, the *MuxId* bits are simply an extension of the line state bits.

Within a given physical page, memory lines from the different virtual pages are mixed. While the memory line with the correct physical address is being accessed, the *MuxId* lines coming from the TLB are being matched against the *MuxId* bits in the tag. A hit occurs only if the *MuxId* bits match. Note that, in our scheme, the actual value of the *MuxId* bits assigned to a given virtual to physical mapping is arbitrary. The *MuxId* bits act as an identifier – they are not related to the actual virtual page numbers that they help identify.

Two lines from two virtual pages that are multiplexed into the same physical page can conflict in the attraction memory if they both have the same page offset. We will call this a *line collision*. In that case, one line displaces the other from the node. If the displaced line is the last copy in the machine, it has to be stored in another node [8].

To handle line collisions easily, we simplify the cache coherence protocol as follows. Each virtual page that is multiplexed in one or more nodes in the machine will have a non-multiplexed mapping in one node. The mapping that is not multiplexed is called the *repository* copy of the page. The repository copy may have lines in any of the possible states, including invalid. The purpose of the repository copy of a page is to absorb any lines of the virtual page when they are displaced from other nodes due to line collisions. The repository copy is first created when the page is read in from disk and can migrate from node to node. Of course, the directory knows at all times which node has the repository copy of the page. To gain insight into how all these operations occur, we now describe the page management algorithms that we use.

3.2 Page Management Algorithms

There are three related problems that we need to solve, namely how to allocate a page, when to multiplex a page, and how to accept a page that another node needs to displace. We consider each issue in turn.

3.2.1 How to Allocate a Page?

When a processor references a virtual page that is not mapped in the node, the operating system executes the Page Allocation algorithm (Figure 2-(a)). The algorithm first checks if there is a free physical page in the node. If so, it is used. Otherwise, if the virtual page faulted on (which we call incoming page) does not have a repository copy in any node, a repository page must be created. Since such a page cannot be multiplexed, the entire test for multiplexing is skipped in this case. Otherwise, the operating system tries to multiplex the incoming page into some partly-used physical page. The algorithm proceeds as follows. The list of physical pages in the node is scanned starting from the most recently-used (MRU) page. The MRU algorithm can be approximated by selecting pages based on their aged reference bits. We start from the MRU page because we want to multiplex together pages that are used during approximately the same time period. Otherwise, any LRU-related page replacement policy would be unable to swap out old data simply because it is multiplexed with newer data. Also, when we traverse the list of physical pages for multiplexing, we do not consider repository pages. This is because repository pages should not be multiplexed. For the other pages, the first page that we find that can host the incoming page (Section 3.2.2) is selected and further multiplexed to hold the incoming page too.

If we traverse the whole list of physical pages and all pages are either repository pages or are such that the incoming page cannot be multiplexed into them (Section 3.2.2), we have no alternative but to free up one physical page from the attraction memory. The same thing occurs if we are trying to allocate a repository page. In any case, we choose as the victim page the first page in LRU order that is non-repository, and apply to it the Page Send algorithm (Figure 2-(b)). This algorithm sends all the lines that the page has in state *master* or *dirty* to the corresponding repository pages (Figure 2-(b)).

In rare occasions, all non-repository pages may have pending transactions and, therefore, be unselectable as victim pages. In that case, we select the LRU repository page as victim. A page-sized space has to be found in another node to hold the new repository page, and any *master* or *dirty* lines in the victim page have to be sent to that new node. To find

```
If (free page exists)
    select it
Else
    If (incoming page is not repository)
        select first page in MRU order into
         which the incoming page can be muxed
        If (success)
            exit
    select first non-repository page in LRU order
    if none, select LRU page as victim
    If (victim is repository)
        ask other nodes until one accepts
         the victim as repository
    invoke Page Send algorithm
```

(a): Page Allocation algorithm

```
send all master and exclusive lines to
 their respective repository pages
unmap all the virtual pages mapped
 into this page
```

(b): Page Send algorithm

```
If (a non-repository copy of victim with
    a multiplexing degree of 1 exists)
    make it the repository copy and merge
     the victim with it
Else
    select first non-repository page in
     LRU order up to ReplFrac fraction
    If (success)
        invoke the Page Send algorithm on
         selected page
        invoke the Page Receive algorithm
         on the victim
```

(c): Page Acceptance algorithm

```
create mapping for the page
initialize all lines to state invalid
receive the lines for this page
```

(d): Page Receive algorithm

Figure 2: Page management algorithms. They are all executed by the operating system.

space, node P first interrogates the other nodes that have a multiplexed copy of the page. If none of them can accept the victim page (Section 3.2.3), it checks the nodes that do not have a multiplexed copy of the page. In each case, the nodes are considered in increasing node ID, starting from $P+1$ and wrapping around until $P-1$. If no node can accept the victim page, we contact the node containing the repository copy of the incoming page and exchange the victim page for the incoming page. In any case, the directory is informed of the new location of the repository page. The other nodes that had multiplexed copies of the victim page are unaffected.

3.2.2 When to Multiplex a Page?

A physical page can host the incoming page only if its current degree of multiplexing is less than *NumMux*. This is a constraint imposed by the limited size of the *MuxId* field. Intuitively, a second constraint might be that the virtual pages currently mapped to the physical page and the incoming page be *compatible*. This means that their actively used lines tend to be in different parts of the page and, therefore, the pages can all coexist together in the same physical page without many line conflicts. However, as we will show when we examine parameter sensitivity (Section 6.2), there is no need to take into account compatibility when selecting the pages that are mapped together. Consequently, we do not do it.

3.2.3 How to Accept a Page?

In some cases, a node may be asked to accept a repository page that has been selected as victim for replacement in another node. To decide whether or not the node can accept such a victim page, it executes the Page Acceptance algorithm (Figure 2-(c)). In it, we first check for a multiplexed copy of the victim page with multiplexing degree of 1. Such a copy, if found, is made the repository copy and the victim page is merged with it. Otherwise, we select for replacement the first non-repository page in LRU order on this node and invoke the Page Send algorithm on it (Figure 2-(b)). Then, the victim page is received (Figure 2-(d)). If this node already has a multiplexed copy of the victim page and it is not the page selected for replacement, then the multiplexed copy has to be removed from the page table and TLB entries, and its lines merged with the incoming page. If, instead, all the pages traversed are repository pages or have pending transactions, the Page Acceptance algorithm fails and the node cannot accept the victim page. In our algorithm, we examine only a *ReplFrac* fraction of the pages because we do not want to replace any of the MRU pages.

3.3 Hardware and OS Support

Multiplexed Simple Coma can be implemented with simple hardware and operating system modifications to Simple Coma. The hardware modifications do not affect the processor chip, which can support the MuxId bits and still be off-the-shelf. To understand why, recall that, in Multiplexed Simple Coma, each TLB entry and cache line tag is extended with the MuxId bits. Every time that a processor issues a memory request, the TLB supplies the MuxId bits in addition to the physical page number. In the caches, these bits are compared to the MuxId bits added as address tags. These operations can be supported by an off-the-shelf processor if the physical address space currently used by *each* processor is smaller than the physical address space for which its TLB and cache tags were dimensioned. The latter is given by the hardware bits available to store the physical page number in the TLB and the address tag in the caches. All we need is for these two fields to have enough unused bits in their most-significant-bit region such that we can store the MuxId bits there. This "bit stealing" is made easier by the fact that, in Simple Coma, physical addresses need fewer bits than in Flat Coma: each processor only addresses the physical memory in its own node.

There are two hardware modifications that need to be made to Simple Coma to support Multiplexed Simple Coma. The first one is to add the MuxId bits to the lines of the attraction memory. These bits are placed beside the state bits of the line. They are a logical extension of them. The checking of the MuxId bits is done in parallel with the checking of the state bits. Therefore, Multiplexed Simple Coma is as fast as Simple Coma. Of course, in Multiplexed Simple Coma, the memory controller must be made to ignore the MuxId bits in the address lines.

The second modification is to extend the cache coherence protocol to handle line collisions in the attraction memory. Recall that, in Simple Coma, a line never needs to displace another one in the attraction memory in hardware while, in Multiplexed Simple Coma, this is possible. This, however, is a simple operation that involves sending the line to the node that, according to the directory, contains the repository copy of the corresponding page. It is simpler than trying to find space for the last copy of a line as it is done in Flat Coma.

Finally, the operating system modifications are also simple. They involve supporting the algorithms in Figure 2. Note that the operating system in Simple Coma also needs to support algorithms similar to those in Figure 2. The main difference is that Simple Coma's algorithms do not need to deal with multiplexing.

4 Evaluation Methodology

To evaluate the performance of Multiplexed Simple Coma, we perform execution-driven simulations of parallel applications. We compare different Flat Coma, Simple Coma, and Multiplexed Simple Coma configurations. In this section, we describe the architectures simulated and the applications run.

4.1 Architectures Simulated

We model Flat Coma, Simple Coma, and Multiplexed Simple Coma machines with similar memory hierarchies. The default parameters are as follows. Each machine has 32 processors running at 200 MHz, 8-Kbyte primary caches and 32-Kbyte secondary caches. In each application, the size of the attraction memory is adjusted for 70% memory pressure in the baseline architecture, and for 50% memory pressure in the architecture of Section 6.4. The caches are chosen to be small because the working sets of the applications are small too. The primary caches are direct-mapped and have 32-byte lines, while the secondary caches are 4-way set-associative and have 64-byte lines. The attraction memories have 64-byte lines and, in Flat Coma, are 4-way set-associative. Caches are maintained coherent with a full bit-vector, write-invalidate directory-based protocol. The algorithm used when node P displaces the master copy of a line in Flat Coma or the master copy of a page in Simple Coma is as follows. First, we check the other nodes that have a copy of the line or page. If no other node has a copy, we check all nodes. In both cases, the nodes are considered in increasing node ID, starting from $P+1$ and wrapping around until $P-1$. If all checks fail, we swap the line or page with the node that provides the new data. The round trip latencies in an unloaded machine to the secondary cache, local memory, and remote memory in two and three hops are 10, 60, 180, and 260 cycles respectively. The TLB has 64 entries, is fully associative, and has a miss penalty of 20 cycles. All contention in the machine is modeled except in the network, where a constant latency is assumed.

Table 2: Default memory management parameters in Multiplexed Simple Coma.

Parameter	Explanation	Value
Page Size	-	4 KB
Memory Pressure	Ratio of data size to total physical memory	68-73%
NumMux	Maximum number of virtual pages multiplexed in a physical page	4, 8
ReplFrac	Fraction of physical pages traversed in the Page Acceptance algorithm	0.9
AllocCycles	Cost of the Page Allocation algorithm	1,000 Cycles
AcceptCycles	Cost of the Page Acceptance algorithm	1,500 Cycles
BaseMovCycles; *IncMovCycles*	Parameters to compute the cost of moving a page between two nodes	1,000 ; 100 Cycles

The times taken by the algorithms in Section 3.2 are estimated as follows. We measure the number of instructions executed when each of the algorithms in Section 3.2 is invoked in our simulator. After running the simulator for all our applications, we know, for each algorithm, the average number of instructions executed per invocation. Then, we make allowance for cache misses and kernel invocation costs. The resulting cycle cost of the most important algorithms, plus the value of some memory management parameters is shown in Table 2.

The cost incurred by a node that is sending a victim page to another one is computed as $BaseMovCycles + IncMovCycles$ * #LinesMoved cycles. This cost includes forcing the cache to write back dirty lines to the attraction memory, sending the lines across the network, and updating page table and TLB. If the receiver node needs to map the incoming page, it spends *BaseMovCycles* cycles. This only occurs when it receives a repository page. In all cases, incoming lines are merged with the attraction memory in hardware. For page allocation, we use first-touch after the parallel section starts. For this first allocation, no cost is charged. The size of the pages is 4 Kbytes.

4.2 Applications Run and Simulation System

We trace seven applications from the Splash-2 suite [17]. Table 3 summarizes some of application-specific information relevant for this paper. Our simulations are performed within the execution-driven simulation framework of TangoLite [4]. The applications are instrumented so that all memory accesses result in calls to our memory simulator.

Table 3: Characteristics of the applications traced.

Application	Description	Input
Barnes	Simulates the interaction of a system of galaxies	16K bodies
Cholesky	Cholesky factorization of a sparse matrix	*tk14.O*
FFT	Complex 1-D version of a six-step FFT algorithm	64K points
Ocean	Simulates eddy and boundary currents in a square ocean basin	258 x 258 grid
Radix	Iterative radix sort of integers with radix set to 1024.	256K ints
WaterNsq	Simulates in $O(n^2)$ time a system of water molecules	512 mols
WaterSp	Same as *WaterNsq*, but uses an $O(n)$ algorithm	729 mols

5 Basic Performance Comparison

To compare the performance of the Flat Coma, Simple Coma, and Multiplexed Simple Coma organizations, we examine three aspects: execution time of the applications, page management activity, and attraction memory misses.

5.1 Execution Time

The most important performance metric is the time it takes to execute the applications. Figure 3 compares the execution time of the applications under Flat Coma (*Flat*), Simple Coma (*Sim*), and Multiplexed Simple Coma with a maximum degree of multiplexing NumMux of 4 (*Mux4*) or 8 (*Mux8*). In the figure, for each application, the execution time is normalized to *Sim*. The execution time is decomposed into several categories, namely time when processors are busy executing application instructions (*Busy*), time when the operating system is executing the Page Allocation algorithm(*PgAlloc*) or performing page transfers (*PgMove*), time that the processor spends synchronizing (*Sync*), and processor stall time due to read misses in the primary caches and write buffer overflow (*Mem*). The time spent by the operating system executing the Page Acceptance algorithm is too small to appear in the chart.

The figure shows that, for the environment measured, Simple Coma is much slower than Flat Coma. On average, Flat Coma reduces the execution time of Simple Coma by about 65%. This is the cost of using the simpler hardware of Simple Coma[1]. The figure also shows that Multiplexed Simple

[1]The change in the execution time of *Cholesky* is largely due to a change in *Busy* time. This effect is caused by the non-determinacy

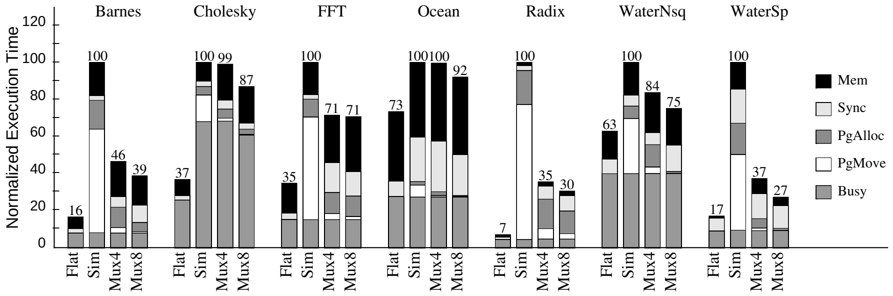


Figure 3: Breakdown of the execution time of the applications under different architectures. For each application, the bars are normalized to *Sim*.

Coma is significantly faster than Simple Coma. Indeed, *Mux4* and *Mux8* reduce the execution time of Simple Coma by an average of 33% and 40% respectively. The speedups of Multiplexed Simple Coma are achieved with the hardware and operating system support described in Section 3.3. Consequently, Multiplexed Simple Coma is more cost-effective than Simple Coma. Although Flat Coma still has 45% less execution time than *Mux8*, Multiplexed Simple Coma is less expensive and, therefore, may represent an acceptable cost-performance design point in some situations.

If we focus on Simple Coma, we see that the main reason why *Sim* is slower than *Flat* is the overhead of page management. The *PgMove* overhead appears when a page needs to be transferred between nodes. It may involve forcing the cache of the sender node to write back to the attraction memory the dirty lines of the page, sending the lines across the network, and updating page table and TLB in the sender and (in some cases) receiver nodes. The *PgAlloc* overhead comes from the execution of the algorithm in Figure 2-(a).

For a given application, the magnitude of the page management overhead (*PgAlloc* + *PgMove*) is an inverse measure of the page locality of the application. Let us define the page working set as the total data in pages that have at least one line belonging to the working set. In *Sim*, what matters is the page working set instead of the traditional working set. For applications where the page working set of a processor largely fits in the attraction memory, few page allocations and deallocations are necessary. As a result, *PgAlloc* and *PgMove* will be small. However, the opposite occurs for applications where the page working set overflows the attraction memory. The latter situation occurs in *Barnes*, *FFT*, *Radix*, *WaterNsq* and *WaterSp*.

In addition, this page management overhead often compounds its effect by inducing large *Sync* times. The reason is that different overheads in different processors may cause load imbalance in barriers or the swelling of critical sections. This in turn increases the time that other processors are waiting at synchronization points. *Sync* time is especially high in *WaterSp* and *Ocean*.

With Multiplexed Simple Coma, we compact the useful data from different pages into a single physical page and, therefore, decrease the size of the page working set. Intuitively, this should have two effects. First, it should reduce the number of page allocations and deallocations, therefore reducing the *PgMove* and *PgAlloc* overheads. Second, it should reduce the number of conflict misses in the attraction memory, therefore reducing the *Mem* overhead. The first effect is clear from Figure 3. Relative to *Sim*, the combined impact of *PgMove* and *PgAlloc* is reduced by an average of about 75% in *Mux4* and 90% in *Mux8*. This is the most obvious impact of Multiplexed Simple Coma. The second effect, namely a reduction in *Mem* overhead, did not occur. Relative to *Sim*, *Mem* tends to increase in *Mux4* and *Mux8*. The impact of *Mem* on overall performance, however, is less important. To understand these two effects of Multiplexed Simple Coma better, we now look in detail at the page management activity and the attraction memory misses.

of the application, not by fundamental Simple Coma effects.

5.2 Page Management Activity

To a first approximation, the magnitude of the page management overhead is a function of how often the Page Allocation algorithm is invoked. Table 4 shows the number of invocations of this algorithm per 10^5 application accesses to shared memory. The table shows that for *Sim*, an average of 1% of the accesses cause the invocation of the Page Allocation algorithm. In *Radix*, the figures are 6 times higher. These high frequencies explain the slowdown of *Sim* over *Flat* in Figure 3. The page working sets of *Radix*, *Barnes*, *WaterSp*, and *FFT* are too large to fit in the attraction memory of our *Sim*.

Table 4: Number of invocations of *PgAlloc* per 10^5 application accesses to shared memory for different machines and applications.

Application	Sim	Mux4	Mux8
Barnes	725	129	29
Cholesky	32	4	1
FFT	173	50	24
Ocean	16	4	1
Radix	6866	1488	574
WaterNsq	41	18	1
WaterSp	443	34	4
Average	1185	247	91

If we now consider *Mux4* and *Mux8*, we see that, relative to *Sim*, the invocation frequency on average decreases by 5 times in *Mux4* and 13 times in *Mux8*. The average value is misleading because it is heavily biased by *Radix*. In fact, for *Mux8*, 4 applications have negligible invocation rates. Overall, we can see that the multiplexing optimization effectively reduces the page working set of all the applications.

Table 4, however, does not show what fraction of the potential effectiveness of multiplexing is actually achieved. In particular, we would like to know what fraction of a physical page is actually used by the virtual pages mapped into it and how many virtual pages end up getting multiplexed in the same physical page. Figures 4-(a) and 4-(b) show these parameters for representative pages. Representative pages are multiplexable physical pages that have been in use for a while. Specifically, we measure all non-repository pages that are selected for replacement in the Page Allocation and Acceptance algorithms and those that are left at the end of the application's run. Note that, in Figure 4-(a), used lines also include lines that are currently invalid.

Figure 4-(a) shows that, for *Sim*, only an average of 17% of each page is used. This small figure suggests that we are wasting a lot of memory. The applications that have the lowest page usage are the ones that have the highest page management overheads in Figure 3. With *Mux4*, pages are used more. On average, 28% of each page is used. While, for some appli-

Application	*Sim*	*Mux4*	*Mux8*
Barnes	0.20	0.44	0.56
Cholesky	0.30	0.34	0.65
FFT	0.06	0.17	0.35
Ocean	0.20	0.37	0.36
Radix	0.06	0.08	0.16
WaterNsq	0.20	0.37	0.69
WaterSp	0.16	0.20	0.41
Average	0.17	0.28	0.45

(a)

Application	*Sim*	*Mux4*	*Mux8*	Up. Bound
Barnes	1	4.0	8.0	65
Cholesky	1	4.0	7.0	7
FFT	1	3.9	7.7	47
Ocean	1	3.7	4.2	5
Radix	1	4.0	8.0	45
WaterNsq	1	4.0	7.6	8
WaterSp	1	4.0	8.0	53
Average	1	3.9	7.2	33

(b)

Figure 4: Effectiveness of page multiplexing. Chart (a) shows the fraction of used lines per page, while Chart (b) shows the number of virtual pages multiplexed per page.

cations like *Barnes*, about 45% of the page is used, the overall average is still low. Surprisingly, Figure 4-(b) shows that, for *Mux4*, an average of 3.9 pages are multiplexed in a page. For *Mux8*, the figure shows that about half of each page is used and that there are an average of 7.2 multiplexed pages per page. Overall, although *Mux8* uses the pages more efficiently, the data seems to imply that there is still room for improvement. In Section 6.3, we address this issue while discussing the last column of Figure 4-(b).

5.3 Attraction Memory Misses

Sim is likely to suffer more attraction memory misses than *Flat*. This is because the allocation of page-sized chunks of memory in *Sim* increases the number of conflicts. When a page currently in use gets replaced to make space for a new one, all the lines of the page are displaced. Subsequent accesses to these lines suffer misses. As a result, Figure 3 shows that the processor stall time due to memory accesses (*Mem* in that figure) in *Sim* is about 40% higher than in *Flat*. We expected that, as we multiplex several pages into one in *Mux4* and *Mux8*, we would reduce the number of misses. However, Figure 3 showed that the *Mem* time in *Mux4* and *Mux8* is often higher than in *Sim*. Since the *Mem* time is rarely a small part of the execution time in Multiplexed Simple Coma, we would like to understand this effect. To do so, we measure the number of attraction memory misses for all the machines. The measurements are shown in Figure 5. In this figure, the bars of each application are normalized to *Sim* and are divided into categories depending on the type of misses. The types of misses are those resulting from start-up effects (*Cold*), coherence activity (*Coh*), and conflicts in the attraction memory (*Confl*).

In this figure, we are interested mostly in how *Confl* misses change. Note that *Confl* misses dominate for 70% memory pressure. The rest of the misses, namely *Cold* and *Coh*, should not change much across architectures. In reality, they change a bit because, when we transfer pages, we may be prefetching data and, therefore, may eliminate misses. Focusing on *Confl* misses, we see that, as expected, *Sim* suffers more *Confl* misses than *Flat*. However, the behavior of *Mux4* and *Mux8* is a bit erratic. In *WaterSp*, *Mux4* and *Mux8* have fewer misses than *Sim* while, in the rest of the applications, the behavior is the opposite. This is because there are two effects. On the one hand, multiplexing reduces page replacements and the line displacements and conflict misses that they cause. On the other hand, multiplexing creates the possibility of line collisions, which cause conflict misses. Which effect dominates depends on the application. On average, *Mux4* and *Mux8* suffer more attraction memory misses than *Sim*. In Section 6.1, we will try to reduce the impact of these misses.

6 Optimizations and Parameter Sensitivity

After the previous base evaluation, we now look at how to improve the performance of Multiplexed Simple Coma and examine its sensitivity to some parameters. The optimization analyzed is subpage prefetching. The parameters varied are the multiplexing selectivity, the maximum degree of multiplexing, and the memory pressure.

6.1 Subpage Prefetching

A possible way to reduce the *Mem* time in Multiplexed Simple Coma is to perform data prefetching. Prefetching whole pages, however, is likely to bring much useless data, as Table 1 suggests. In addition, prefetching a whole page can potentially displace useful data in the receiver node. Instead, we divide each page into 8 512-byte subpages and prefetch only one subpage. Every time that a node suffers a read miss on a line, the coherence engine brings the whole subpage where the line is. The referenced line is requested first, followed by requests to increasing line addresses, wrapping around at the end of the subpage. As soon as the referenced line arrives, execution resumes. One possible problem with this unconditional prefetching is that it may cause line conflicts if the physical subpage that the prefetches overwrite is being used by the other multiplexed pages. Suppose, however, that we had hardware support to tell us whether any other virtual page multiplexed in this physical page is using that subpage. A virtual page is using a subpage if at least one line of the page mapped in the subpage has been referenced by the processor since the page was last mapped in this node. In that case, a second scheme that we evaluate uses conditional prefetching: if more than $NumMux/2$ pages multiplexed into that physical page are using the subpage, we do not prefetch the subpage. The hardware support required to determine subpage use is presented in [1]. We do not present it here because we are about to show that it is not cost-effective.

Figure 6 shows the impact of these schemes on the execution time of the applications. The figure shows, for *Mux4* and *Mux8*, the impact of unconditional prefetching (*M4Pref* and *M8Pref*) and conditional prefetching (*M4Cond* and *M8Cond*). Unconditional prefetching works acceptably well, reducing the execution time of the applications by an average of 7%. Conditional prefetching works less well, while still reducing the execution time of the applications by an average of 5%. Even unconditional prefetching, however, can only eliminate a small fraction of the *Mem* time on average. The overall speedup is also affected by increases or decreases in *Sync* time induced by the changes in misses. Overall, unconditional prefetching of subpages is a simple optimization that works well.

6.2 Varying the Multiplexing Selectivity

We now examine the sensitivity of the performance of Multiplexed Simple Coma to parameter changes. We start by increasing the selectivity of multiplexing. So far, the only restriction that we have placed when we want to map a new virtual page to a physical page is that the degree of multiplexing of the physical page be less than *NumMux*. We now add

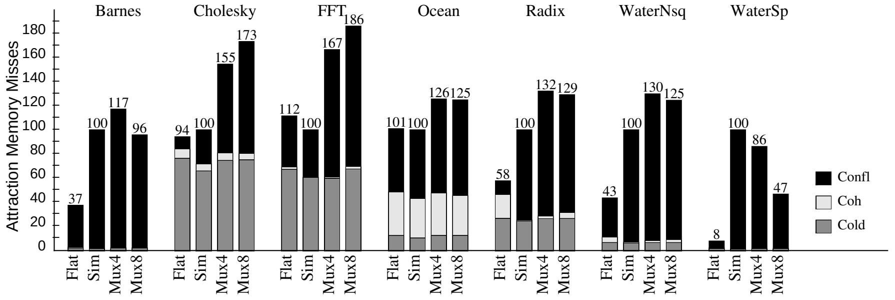


Figure 5: Breakdown of the misses in the attraction memory. For each application, the bars are normalized to the *Sim* bar.

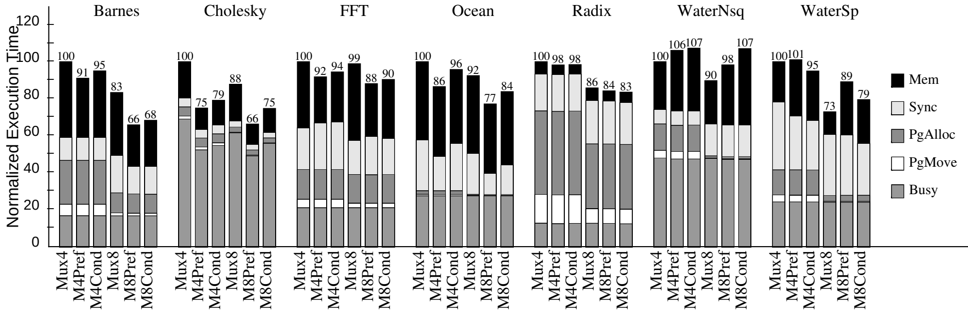


Figure 6: Effect of unconditional and conditional prefetching on the execution time of *Mux4* and *Mux8*.

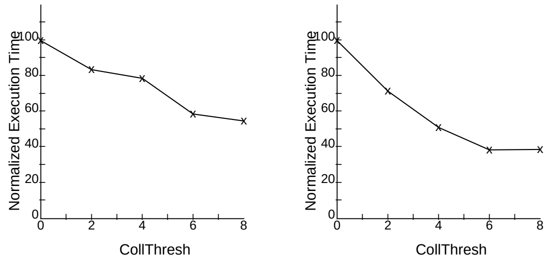
another restriction, namely that the virtual pages currently mapped to the physical page and the new virtual page be *compatible*. This means that their actively-used lines should be in largely different parts of the page. With this added selectivity, we hope to increase the performance of Multiplexed Simple Coma.

To estimate compatibility, we again divide each page into 8 subpages. Each virtual page currently mapped in a given processor has a *Use Vector* with as many bits as subpages per page. For a given virtual page, bit i in its use vector is set if the processor has accessed a line of the page in subpage i since the page was last mapped in this node. The use vector is reset when the virtual page is unmapped. In addition, each virtual page has a software data structure that saves, on a per-node basis, the last 4 use vectors for the page in that node.

With all this information, we determine compatibility as follows. We take the current and the last 4 use vectors of each of the virtual pages currently mapped in the physical page and bit-wise OR them together to get an intermediate use vector. We also bit-wise OR the last 4 use vectors of the new virtual page for this node to get another intermediate use vector. Then, we bit-wise AND the two intermediate vectors. In this final vector, the number of 1's is called the number of *collisions*. The new page is compatible only if the number of collisions is less than or equal to a threshold that we call *CollThresh*. An efficient implementation of the use vectors and collision analysis is discussed in [1]. We do not include it here because we are about to show that selectivity is unnecessary.

The value of CollThresh can range from 0 to 8. A low value means that we are selective. High selectivity has a twofold impact on the execution time. On the one hand, there will be fewer line collisions and, most likely, fewer conflict misses. On the other hand, however, the page management overhead is likely to increase. This is because, in the Page Allocation algorithm (Figure 2-(a)), we must scan the pages until we find one that is compatible. The more selective we are, the more pages we need to scan.

Figure 7 shows the effect of varying CollThresh for *Mux4* (Chart (a)) and *Mux8* (Chart (b)). The figure shows the average execution time for the seven applications considered. The curves are normalized for the case of threshold equal to zero. From the figure, we see that, in both curves, increasing the collision threshold always results in a reduction in execution time. The lowest execution time corresponds to a threshold of 8, where there is no selectivity. There is no need for multiplexing selectivity.



(a): *Mux4* (b): *Mux8*

Figure 7: Effect of CollThresh on the execution time. Each data point corresponds to the average of our seven applications.

6.3 Varying the Maximum Degree of Multiplexing

A second parameter of Multiplexed Simple Coma that we vary is the maximum degree of multiplexing (NumMux). The last column of Figure 4-(b) shows an estimate of the value of NumMux for which no page replacements would be necessary. To compute this value, we assume that repository pages are uniformly distributed across nodes. Consequently, since the memory pressure is 70%, 30% of the pages of each node are available to multiplex virtual pages other than the node's own repository pages. The last column of the table shows the degree of multiplexing that is necessary in these 30% physical pages so that no page replacements are required. A virtual page, once mapped on a node, will remain so until the end of the application. From the table, we see that *Barnes*, *FFT*, *Radix* and *WaterSp* have values much higher than 8, the max-

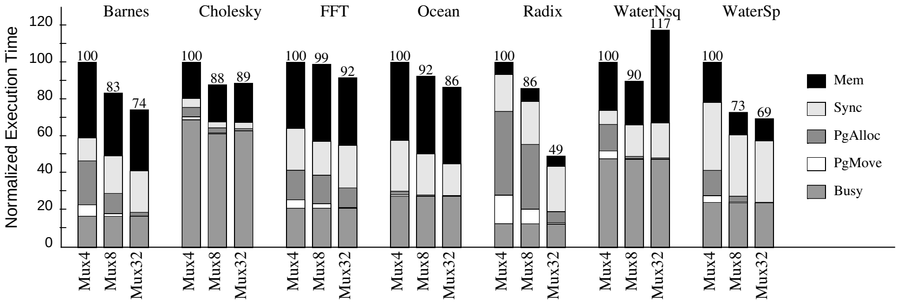


Figure 8: Execution time of the applications for different maximum degrees of multiplexing. For each application, the bars are normalized to *Mux4*.

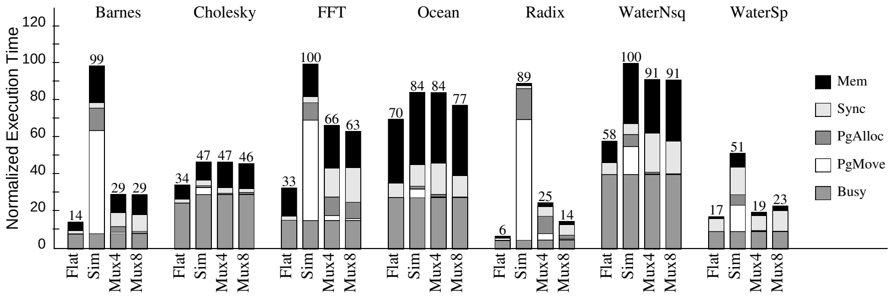


Figure 9: Breakdown of the execution time of the applications under different architectures for 45-50% memory pressure.

imum multiplexing possible in *Mux8*. The other applications have values less than or equal to 8. The latter set of applications can hardly benefit from more multiplexing beyond *Mux8*. Their page management overhead is already very low. This can be seen from Figure 3. The set of applications with large values can potentially benefit from a higher degree of multiplexing beyond that of *Mux8*. They can lower their page management overhead except if, as in *WaterSp*, such overhead is already very small for *Mux8*.

To determine the impact of a high NumMux, we simulated a system with NumMux equal to 32. Such a system, called *Mux32*, is compared to *Mux4* and *Mux8* in Figure 8. The figure shows the execution time and is organized as Figure 3. As expected, *Mux32* runs faster for the applications where the number in the last column of Figure 4-(b) was much higher than 8. *Ocean* also runs faster due to a reduction in *Sync* time. *WaterNsq*, however, runs slower due to an increase in *Mem* time resulting from more line collisions. Overall, although *Mux32* reduces the execution time of *Mux8* by an average of 6%, we feel that *Mux8* is a good design point and that the extra hardware necessary to support a degree of multiplexing beyond 8 is probably not worth.

6.4 Varying the Memory Pressure

Finally, we change the memory pressure of the systems from the 70% of the baseline systems to 45-50%. The resulting execution time of the applications on the four architectures is shown in Figure 9. The figure is organized as Figure 3, and is normalized to *Sim in Figure 3* so that we can see the changes in speed for each architecture. The figure shows that, on average, all the architectures are a bit faster than in Figure 3. The relative speeds of the different architectures are largely unchanged. Flat Coma is fastest, although the Simple Coma-based systems improve a little relative to Flat Coma. The lower memory pressure only helps *Sim* significantly in *WaterSp* and, to a lesser extent, in *Ocean* and *Radix*. Other factors that are not directly related to Coma issues play an important role. Examples of these factors are a reduction in *Busy* time in *Cholesky* for all Simple Coma-based systems and an increase in *Sync* time in *WaterNsq* for *Mux4*. Overall, the conclusions from Section 5.1 still hold for 45-50% memory pressure. Multiplexed Simple Coma is significantly more cost-effective than Simple Coma for this memory pressure.

7 Related Work

Three other pieces of work bear some relation to the work presented here. Falsafi and Wood [3] describe a hybrid architecture called Reactive NUMA in which a page is initially mapped in Numa mode, and gets remapped in Simple Coma mode when the number of conflict misses from that page exceeds a threshold. Memory fragmentation is still possible because a page may have only a few frequently-used lines which caused the page to be remapped in Simple Coma mode. That paper compares the performance of CC-Numa to Simple Coma. Although the architectural parameters used in that paper and in this paper are quite different, we compared the results for the applications that are common to both papers. We find that Simple Coma seems to perform relatively worse in this paper. The reason for this is twofold. First, we use slightly higher values for the page management overheads. Second, our applications suffer more page faults. The number of page faults is heavily dependent on the architecture parameters and algorithms used. We believe that a major reason why we observe more page faults in our system is that we use a higher memory pressure in the memory. The work by Talluri *et al* [15, 16] discusses superpages. Superpages are a way of expanding the coverage of the TLB at the possible expense of memory fragmentation. Finally, the work by Jamrozik *et al* [7] discusses operating system-induced page transfers in a network of workstations. While the environment is totally different to ours, they use the concept of subpages in data transfers. We also use subpages for the prefetching scheme.

8 Conclusions

Cache-only memory architectures are an interesting class of shared-memory multiprocessors. Recently, one such machine called Simple Coma has been proposed as a low-cost alternative to other, hardware-intensive Coma solutions like Flat Coma. However, in this paper, we have found that Simple Coma is quite slower than Flat Coma due to the high page mapping, unmapping, and transfer overhead caused by memory fragmentation.

The contribution of this paper was to propose a solution to the memory fragmentation problem that we call Multiplexed Simple Coma. The idea is to allow multiple virtual pages to map into the same physical page at the same time, therefore compressing the page working set of the application. Multiplexed Simple Coma requires very little support over Simple Coma and reduces its execution time by about 40%. Multiplexed Simple Coma can be very easily implemented with off-the-shelf processors. In addition, there is no need to be selective when choosing what virtual pages are to share the same physical page. Overall, although Multiplexed Simple Coma is still slower than Flat Coma, since it is cheaper to implement, it represents a good cost-performance design point.

Acknowledgments

We thank the referees and members of the I-ACOMA group for their valuable feedback. Josep Torrellas is supported in part by an NSF Young Investigator Award.

References

[1] S. Basu and J. Torrellas. Enhancing Memory Use in Simple Coma: Multiplexed Simple Coma. Technical Report 1528, Department of Computer Science, University of Illinois, December 1997.

[2] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[3] 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.

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

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

[6] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Annual Hawaii International Conference of System Sciences*, pages 522–533, January 1994.

[7] H. A. Jamrozik, M. J. Feeley, G. M. Voelker, J. E. II, A. R. Karlin, H. M. Levy, and M. K. Vernon. Reducing Network Latency Using Subpages in a Global Memory Environment. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 258–267, October 1996.

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

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

[10] A. Landin and F. Dahlgren. Bus-Based COMA – Reducing Traffic in Shared-Bus Multiprocessors. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 95–105, February 1996.

[11] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[12] S. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

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

[14] 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.

[15] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, October 1994.

[16] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. 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.