User-Controllable Coherence for High Performance Shared Memory Multiprocessors [*]

Collin McCurdy and Charles Fischer
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706

{cmccurdy, fischer}@cs.wisc.edu

ABSTRACT

In programming high performance applications, shared address-space platforms are preferable for fine-grained computation, while distributed address-space platforms are more suitable for coarse-grained computation. However, currently only distributed address-space systems scale beyond the low hundreds of processors. In this paper we introduce a hybrid architecture that allows users to trade off local memory usage for coherence communication, making possible larger-scale shared memory architectures. We introduce a programming model and examine possible implementations of hardware mechanisms, evaluating some of the trade-offs inherent in each. Preliminary experiments on an application with particularly fine-grained communication requirements indicate that effective placement of directives can reduce coherence communication by more than a factor of 10 for 64 processors.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: General—*Parallel Programming*

General Terms

Performance, Design, Languages

Keywords

Parallel computation, shared memory architectures, distributed memory architectures, irregular computation.

1. INTRODUCTION

Current high-performance multiprocessor platforms may be broken into two categories: distributed address-space and shared address-space. For programs running on a distributed address-space multiprocessor, data visible on one processing unit is not visible on the remaining processor units. In contrast, hardware mechanisms in high-performance shared address-space environments ensure that all data is in principle visible to all processors. While a shared address-space can offer programmability advantages and better performance for fine-grained applications, the very mechanisms that create those advantages appear to prevent the machines from scaling to large numbers of processors.

Programmability. Programming a distributed address-space machine requires adherence to a model in which "computation-follows-data" – the data-parallel model – which necessitates careful user attention to data placement and movement. Shared address-space machines offer a model in which "data-follows-computation."

For *coarse-grained* application phases – in which non-local data requirements are easily identified before a large section of computation – the distinction is not fundamental. However, the task of parallelizing *fine-grained* application phases – in which non-local data dependences can only be identified for small pieces of computation at a time – is significantly simplified when data placement does not require user attention.

Performance. Performance trade-offs break along similar lines: coarse-grained application phases enjoy better performance on distributed hardware, where all communication takes place before any computation. Non-local data, once made local, stays local. On the other hand, hardware controlled shared address-space machines, based on cache-coherence protocols, can suffer from the need to fetch "non-local" data multiple times, due to cache replacements or writes to shared data.

However, fine-grained application phases are not well-suited to the communication overheads inherent in a distributed address-space environment, where all non-local references must be identified and the data they refer to requested, packed, sent, received and unpacked. Meanwhile, fine-grained application phases are exactly the conditions in which a hardware-based coherence system thrives, providing

data very quickly without the overheads associated with message-passing, one cache-block at a time.

Ideally, we could use hardware-based shared address-space platforms to run applications with predominantly fine-grained computation, and distributed platforms for predominantly coarse-grained applications. Unfortunately, the overheads inherent in providing the "data-follows-computation" model prevent those machines from scaling beyond tens, or perhaps the low hundreds, of processors. Meanwhile, distributed address-space machines with *tens of thousands* of processors are currently in development.

Hybrid. This work addresses the question: What if we need to perform fine-grained computation on a large-scale platform? We conjecture that shared and distributed address-space architectures are actually just endpoints on a spectrum, and consider a midpoint: a machine with both address spaces. We consider changes to the basic shared address-space architecture that allow users to take advantage of two address spaces to gain the scalability of distributed architectures while retaining the benefits of the shared address-space architecture.

In our model there are two kinds of memory: *global*, which hardware keeps coherent, and strictly *local*, which is left incoherent. We provide users with a mechanism for specifying, at a very high level, data that would benefit from *localization* – being moved from global to local memory – offering the following benefits for memory accesses that have been localized:

- Faster access to localized data.

- Elimination of redundant protocol traffic.

- Elimination of locking/contention for written shared data.

We propose modifications to a basic hardware shared memory coherence protocol that allow the efficient, dynamic movement of data between local and global address spaces, *without* copying.

The remainder of the paper describes these ideas in more detail. First, in Section 2, we describe a specific application and its performance on currently available architectures, making more concrete the motivation for our *localization* extensions.

We then discuss how references become localized in Section 3. We introduce our programming model, which relies on compilers to translate high-level user knowledge about locality into low-level instructions, allowing users to logically move global data to local memory without physically moving it.

In Section 4, we describe preliminary experiments that show that the effective placement of directives in the application described in Section 2 can reduce coherence communication by more than a factor of 10 for 64 processors.

Then, in Section 5, we delve into some architectural details to demonstrate that the addition of mechanisms to implement the low-level instructions is possible without vast modifications to existing hardware. We examine several possible approaches, and evaluate the trade-offs inherent in each.

Finally, we discuss related work in Section 6, before concluding in Section 7.

2. MOTIVATION

In this section we motivate our hybrid through an analysis of the programmability and performance of an application on existing "endpoint" architectures.

First we describe the application: *ammp*, a molecular simulation from the SPEC OMP2001 benchmark suite. We chose to work with this benchmark for two reasons:

1. It has been described as having poor scaling properties [19], and is notably absent from the large version of the SPEC OMP2001 benchmark suite.

2. While only a benchmark, its relevance is increased by its similarity to the protein folding problem that has prompted IBM's Blue Gene [2] petaflop supercomputer project.

We then describe two parallelizations of the benchmark: first the SPEC OMP2001 OpenMP shared-memory version, then a version with the significant modifications necessary to run on distributed address-space architectures. Finally, we analyze the relative performance of the variants, motivating a possible midpoint along the address-space spectrum.

2.1 Application: *ammp*

Ammp is a molecular dynamics simulator found in both SPEC CPU2000 benchmark suite and the *medium* version of the SPEC OMP2001 suite. The OpenMP version performs a molecular dynamics simulation of a protein-inhibitor complex embedded in water. Computation is dominated by the loop nest that computes the contribution of non-bonded forces, demonstrated in [3] to account for more than 96% of the execution time of the entire benchmark. The loop nest computes the non-bonded forces that simulated atoms exert on one another. If no approximation is used, the solution requires $O(n^2)$ computation: for each atom, determine the force exerted on it by every other atom.

Ammp uses an approximation similar to that used by well known tree-based solutions like Barnes-Hut [4], in that force contributions of spatially co-located *clusters* of atoms a sufficient distance away from the atom under consideration are lumped together. However, likely because the number of atoms under consideration tends to be fairly small for molecular dynamic problems, the groupings are made at only a single level, saving the overhead of creating and maintaining a tree.

The loop nest takes as input a list of atoms and a list of nodes, and executes the following algorithm:

```
foreach atom A
    foreach node N
        if N contains A or is one of 26 nearest neighbors
            foreach atom B in node N
                compute forces between A and B
        else
            compute forces between A and node N
```

Note the fine-grained nature of the access to atoms in the "nearest neighbors" part of the computation: as the algorithm is constructed, the atoms that A will interact with are not known until immediately before the force computation.

The uniprocessor algorithm takes computational advantage of the symmetric nature of force computation: in computing the forces exerted by atom B on atom A, one is doing much of the work necessary to compute forces exerted

by atom A on atom B. The algorithm therefore computes forces for both A and B at the same time.

Finally, we note that the code makes no attempt to take advantage of the potential spatial locality of atoms. Clearly, the atoms with which another atom interacts will be near to it in space. However, in the program atoms are stored in an array with no consideration for where they exist in space. Well-known "linear sorting" techniques – along a Hilbert curve, for instance, as described in [16] – can significantly improve locality at minimal cost.

2.2 OpenMP Implementation

The ability to assume rapid, fine-grained access to all data in the shared address-space significantly simplifies the parallelization process for this application.

The SPEC OMP2001 OpenMP implementation parallelizes the outer loop, using a simple "omp parallel for" directive. The directive makes use of the "guided" scheduling parameter, a work-list solution to computation partitioning whereby each processor takes N/P atoms, where N is the number of *remaining* atoms after the previous processor has taken its share. This dynamic partitioning strategy allows for iterations that take varying amounts of time, providing computation balance at some cost to locality.

The only additional complication is the placement of locks around data involved in the computation-saving device for atom-atom interactions noted above. The locking is necessary because data for atom B is potentially read and updated by multiple processors at the same time. In our experiments, we have found that as the number of processors increases, redundant computation becomes significantly more cost-effective than the serialization caused by locking. The cost difference is evident in the results below, where we compare the performance of a version from which we have removed the locks (and added the extra computation) with the performance of the benchmark version.

2.3 Distributed Implementation

In this section, we describe our distributed memory parallelization, noting how it is hindered by the *lack* of a shared address-space.

There are two potential sources of non-local data immediately evident: the node-list and the atom-list. We chose to take advantage of our knowledge of the problem – and of the specific input data – and replicated the node-list, which is small because the number of atoms is small. However, we distribute the atom-list data (in a simple block distribution) in order to distribute the computation of the outer loop, leaving the problem of gathering non-local atom data for atom-atom interactions.

As the algorithm is constructed, this is very fine-grained proposition: we don't know what non-local atom will be required by a processor until it is actually accessed. We can force the algorithm into a more coarse-grained construction by breaking it into two parts, separated by a communication event, as follows:

```
foreach atom A
    foreach node N
        if N contains A or is one of 26 nearest neighbors
            foreach atom B in node N
                add B to A's gather-list
        else
            compute forces between A and node N

foreach atom A
    gather non-local atoms on A's gather-list

foreach atom A
    foreach atom B on A's gather list
        compute forces between A and B
```

This approach requires significantly more memory per atom, to account for its gather list.

Gathering non-local atoms from gather lists requires several steps. First, each processor determines which processors own each atom on each gather list, taking care to ensure that an atom that appears on more than one list is only gathered once. Note that this determination must be made every time-step, since atoms may change position and, therefore, node. Next, each processor sends every other processor the indices of the atoms it requires, while at the same time servicing requests from every other processor. Finally, each processor receives from every other processor the data it requested. Our MPI implementation requires about 100 lines of C-code despite the use of very high-level MPI collective communication primitives, including one call to *MPI_Alltoall* and two calls to *MPI_Alltoallv*.

In order to make the number of gathered atoms manageable, we are forced to use the linear sorting technique described above, to introduce spatial locality into the distribution of atoms across processors.

2.4 Performance and Discussion

2.4.1 Experimental Platforms

We ran our experiments on two different implementations of the shared memory architecture, the HP Superdome [12] and the Sun Enterprise 10000 [9], and one distributed memory architecture, IBM's Blue Horizon [7]. Below we describe some of the major defining characteristics of the representatives of each platform that we used to gather data.

HP Superdome: 64 750Mhz PA-8700 processors. Single level of data cache, 1.5MB. Very tightly coupled, directory-based shared memory coherence protocol.

Sun Enterprise 10000: 36 400Mhz UltraSparc processors available (up to 64 possible). 16KB first level data cache, 4MB external cache. Snoopy broadcast-based shared memory coherence protocol.

Blue Horizon: 1152 375Mhz IBM Power3 processors, organized as 144 SMP nodes, each consisting of eight processors, 1.7 teraflops peak performance.

2.4.2 Results

Figure 1 demonstrates the effect of removing locks. Without locks, scaling is linear all the way out to 64 processors on the HP. Figure 2 demonstrates that the increased scalability does not come at the cost of absolute performance. The figure also shows the performance benefits of "linearly sorting" atoms, though the HP's ability to deal nearly as well with the unsorted atoms is perhaps more telling. Re-
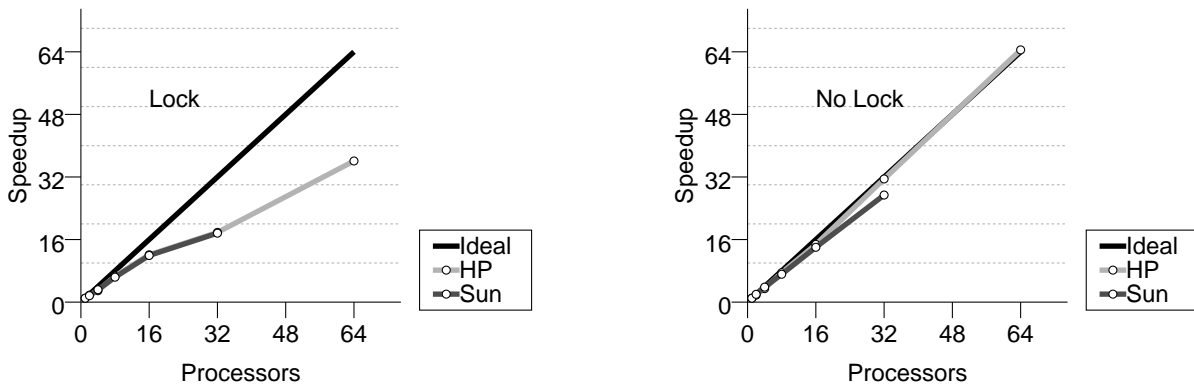
Figure 1: Relative performance of ammp with locking (left), and without locking (right), on two shared address-space implementations, snoopy (Sun) and directory-based (HP).
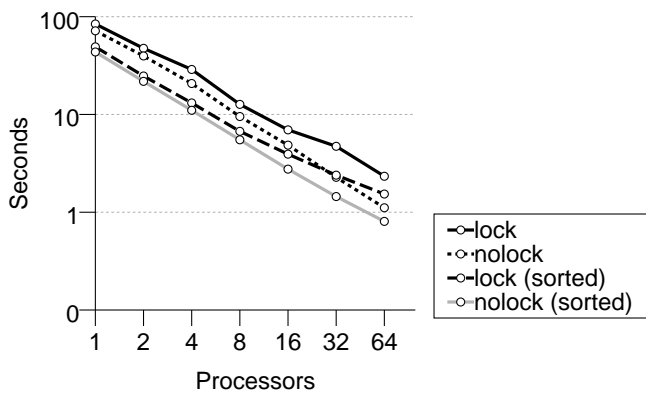

Figure 2: Absolute performance on shared memory platform with and without locking, when atoms are unsorted and sorted.

Figure 3: Scaling of *ammp* on the distributed platform.

call that the distributed memory version *requires* sorting the atoms in order to get results in a reasonable amount of time.

Figure 3 demonstrates the program scalability problems of the distributed memory platform. Scaling performance, while not linear, is respectable to 64 processors but then levels off until completely dropping at 512 processors. Figure 4 provides some insight into the reasons for the leveling and drop-off: it shows the minimum, maximum, and average time a processor takes to execute each computation and communication phase. Two problems are evident. First, as the number of processors increase, load imbalance becomes an issue. Second, and more crucially, by the time we reach 256 processors, communication begins to dominate computation. Load balance can be addressed, but doing so would require even more communication.

We have shown both the difficulty in programming the distributed address-space platform and the performance benefits that a shared-address space platform offers, particularly when locking can be avoided. However, inspection of the difference in the number of available processors speaks volumes about the *architectural* scalability of each platform. In the next section we look at how we might increase the scalability of a shared-address space implementation while keeping many of its inherent good qualities.
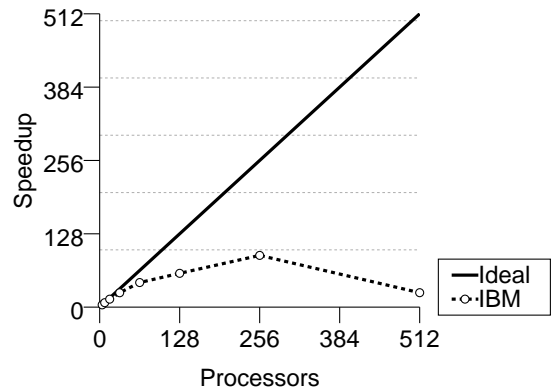
3. LOCALIZATION

We call the act of moving a reference from global memory to local memory *localization*. In this section we describe how references become localized at the program-level. First, we identify and distinguish between two targets of localization. Then we discuss a multi-level programming model, and present examples showing how the levels interact. We conclude the section with a discussion of some of the potential drawbacks of localization.

3.1 Targets of Localization

The aim of localization is to allow users to remove two sources of unnecessary overhead inherent in hardware-controlled shared address-space architectures:

1. Redundant coherence traffic:

   (a) Capacity cache misses, i.e. misses that result when an application's data set does not fit in cache.

   (b) Invalidate/re-read cycles due to updates to shared data involved in a reduction (explained below).

2. Serialization due to locking of data involved in a reduction.
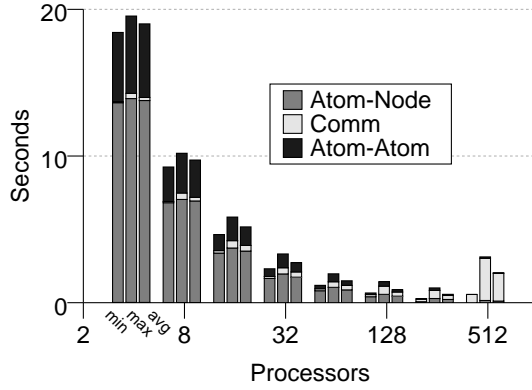
Figure 4: Analysis of contributions of computation and communication components to the runtime on the distributed platform.

The idea is to allow programmers to identify at a very high level references that are likely to cause these problems, and *logically* move the data to local memory. As we shall see in Section 5, we do not have to physically move the data but need only ensure that if it is evicted from the cache, it goes to local memory.

Clearly, localization can be effective for read-only data since multiple copies of such data can exist without interference. Similarly, data that is written but not shared during a particular computation can also be localized: since there is a single copy, there is no interference.

On the other hand, writes by one processor to data that is subsequently read by another processor would appear to be unsuitable for localization. However, there is a special case that occurs when a memory location is used to *accumulate* values within a loop – i.e., the value written on one iteration is only read on the next iteration in order to update it again. In this case, termed a "reduction" computation in the parallel programming literature, multiple processors can privately accumulate multiple copies of the value and delay a final – global – accumulation until the end of the loop.

This special case is often leveraged to parallelize loops that would otherwise appear to have dependences that prevent parallelization. Distributed memory parallelizations further use these opportunities to perform as much computation on local data as possible before combining results in a reduction. In shared address-space environments, programmers typically add locks around such updates to ensure that the data is not modified by another processor in between the read and the write phases of the update. However, the expense of the resulting serialization may lead a programmer to instead hand-code reductions in privatized memory. In fact, OpenMP provides a reduction primitive, but only for scalars.

The above discussion points to two distinct targets for localization:

1. Read-only and *unshared*-written data that will likely be fetched (due to cache misses) *multiple* times.

2. *Shared*-written data involved in a reduction computation.

```
#pragma omp parallel for, \
        local_reduce(x,+), local(y)
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        x = x + y[j];
    }
}
                    (a)


        ...
        ldp     r3,y(r4)
        ld+     r2,x
        add     r4,r4,4
        add     r2,r2,r3
        stp     r2,x
        ...

                    (b)
```

i	j	stp	ld+	ldp
0	0	local	local	global
0	1	local	local	global
0	...	local	local	global
0	M	local	local	global
1	0	local	local	local
...	...	local	local	local
N	M	local	local	local

(c)

Figure 5: Example demonstrating the programming model for data localization.

3.2 Programming Model

We envision a two-level process by which high-level user knowledge is propagated to hardware. User-level directives naming data – arrays and scalars – that would benefit from being localized are translated by the compiler into instructions that indicate to hardware that individual addresses should be localized. We describe this process by example before delving into the instruction-level details.

3.2.1 Example

Figure 5(a) demonstrates the programming model we envision with a simple C-code for-loop. Note the extensions to the OpenMP "parallel for" directive, indicating the variables to be localized, as well as the reduction operator. Assuming y has M elements, every element of y is read N times.

Figure 5(b) provides the assembly code that a compiler might generate for the reference:

- "ldp" indicates a load from private memory; however, data from the *first* load to that address must come from global memory.

- "ld+" indicates a load from private memory, initialized to zero in preparation for an additive reduction.

- "stp" indicates a store to private memory.

The table in Figure 5(c) shows the locality state of each reference as the loops iterate, assuming, for simplicity, a cache block size of one. The first iteration of the inner loop initializes a local version of x to 0 on the first "read," performs the add, and then writes to the local x. Subsequent

```
#pragma omp parallel for, local_reduce(y,+)
for (i = 0; i < N; i++) {
    y[idx[i]] = y[idx[i]] + x[i];
}
```

<center>(a)</center>

```
        ...
        ld      r3,idx(r5)
        ld      r4,x(r6)
        ld+     r2,y(r3)
        addu    r2,r2,r4
        stp     r2,0(r3)
        ...
```

<center>(b)</center>

Figure 6: A second example illustrating the potential of localization.

iterations read and write the local x. Meanwhile, the first iteration of the outer loop reads each element of y from global memory. Subsequent iterations of the outer loop read y from local memory (assuming a cache miss).

Figure 6(a) and (b) illustrate the power of localization with a more complicated example, an unstructured array reduction. Ordinarily a programmer would need to put a lock around the access to y, to ensure that two processors do not access the same element of y at the same time. Note that, because each element of x and idx is read only once in this fragment, we do not localize either access. If we knew either was going to be read again, and would likely not be found in the cache, we would localize it in the pragma statement.

Unstructured references, subscripted subscripts like those in Figure 6, are the hallmark of irregular applications. Because such references are impossible to analyze at compile time, they complicate parallelization for distributed memory platforms. Localization is especially well-suited for this domain of applications.

3.2.2 Instruction-Level Details

We have identified two approaches to the instruction-level localization depicted in Figures 5(b) and 6(b), each the inverse of the other. We call the approaches **implicit** and **explicit** respectively, referring to the manner in which the programmer uses each to mark the "region of localization" for an address.

The **implicit** approach, demonstrated in the examples above, marks in some way all load and store instructions that reference data that should be localized. The first instance of a marked load or store to a particular address acts like a normal load or store, causing any necessary coherence events to transfer the global data. However, the coherence mechanism also marks the address as "localized" so that all future *marked* loads and stores to that address reference local data. A *normal* load or store to the localized address, issued by any processor, implicitly ends its localization era.

The second approach, which we call **explicit**, explicitly marks the beginning and end of a localization era for a particular address. A control instruction is placed by the compiler before the first load or store of the period of localization, signaling any coherence events necessary to transfer the global data. Another control instruction, to end the lo-

calization, is then placed at some point before the first load or store to that address *after* the period of localization. All loads and stores to that address within the localization era refer to local memory.

Each approach has its advantages and disadvantages:

Implicit. The biggest advantage of the implicit approach is that changes to the source code are limited to the region that needs to be localized. Since any unmodified load or store returns a localized address to the global state, there is no need to worry about finding the end of a localization. Separate compilation of modules is therefore not a problem.

Also, when modifying existing code there is no need to add instructions; only existing instructions need be modified. On the other hand, all loads and stores that reference the address in the localization region must be found.

The most important disadvantage of this approach is that it would require modifications to the instruction set to add new flavors of load and store instructions.

Explicit. The explicit approach, in contrast, does not require instruction set modifications, provided the instruction set already contains a control instruction for implementation specific functions, as most do. Additionally, the control instruction to localize an address only needs to be added before the first load or store to that address. Breaking the construct into two parts allows for compiler optimizations that separate the localization of an address from its first load or store, perhaps allowing additional time for any necessary communication.

The primary disadvantage of this approach, however, is that changes are not necessarily limited to the region of source code that is undergoing transformation. In order to avoid adding references to an application, the compiler must find paths to all potential future references and add control instructions before them.

As we shall see in Section 5, the implicit versus explicit choice has significant ramifications on the implementation mechanisms and would likely be influenced by performance considerations.

3.3 Other Concerns

Finally, we mention two potential drawbacks of our approach, the first the result of tying our localization approach to existing coherence protocols, and the second the result of adding processor-specific local memory.

First, because we are localizing *cache blocks* rather than individual data items, care must be taken to avoid "false localization," an analogue to the well-known false sharing problem. For example, suppose a read-only data item x is collocated in a cache block with another data item involved in a reduction, y: the first read of y, using "ld+," would initialize the entire cache block to zero. A subsequent read of x would therefore incorrectly yield zero. Compilers are capable of ensuring the proper data alignments necessary to avoid such problems, though they must have knowledge of the block-size in order to do so. This has the potential drawback of making the block-size an architectural parameter as opposed to a micro-architectural parameter.

The second potential drawback is perhaps more a mind-set adjustment than a problem: because local memory is physically associated with a processor, localization's presence-of-data guarantee (absent from normal caching) implies pinning a process to the processor it starts on. This

requirement is at odds with one of the perceived advantages of shared address-space architectures: the ability of the operating system to schedule processes on different processors. On the other hand, in high performance situations, operating system intervention is often seen as a hindrance.

In the OpenMP paradigm it might be sufficient to require that "omp parallel" regions be pinned to the processors they begin on, though it must be kept in mind that localized reduction data might potentially need to stay in local memory beyond the scope of a parallel region.

4. PRELIMINARY EVALUATION

Here we describe experiments we performed to estimate how many coherence events localization could potentially remove, and at what cost in terms of memory usage. Experiments consist of simulations of the same portion of the same application, ammp, described in Section 2.

We have chosen to limit our experiments to a single full-scale application in order to demonstrate the full impact that our technique can have on performance at the application level. However, as we pointed out in the discussion of Figure 6, the technique potentially has much wider applicability, especially in the irregular application domain.

We present results for three versions of the application. The first two we have already seen: **lock** is the SPEC OMP2001 version, and **nolock** is the version we modified to remove locks, at the expense of extra computation.

The third version, called **local**, makes extensive use of our localization mechanism. While the instructions were hand-modified, we believe that a compiler could have interpreted properly placed directives to achieve the same result. This version retains the computation-saving device of the **lock** version, but we have replaced all references enclosed by locks with local reductions, and have localized almost every other signal-causing reference. All three versions have an additional "use" loop nest to ensure that all signals due to globalization of variables that are targets of local reductions in **local** are properly accounted for.

Next, we describe the experimental platform before discussing some results of our experiments.

4.1 Experimental Platform

We performed experiments on a modified version of the Simplescalar "sim-cache" simulator [6], a uniprocessor cache simulator that interprets MIPS-like binaries.

We added functionality to the basic simulator in order to model the caches of a shared memory multiprocessor, including the cache-coherence events defined by the MSI protocol. Like the simulator described in [21], our simulator assumes that every instruction, including memory operations, takes one unit of time to execute. Coherence events and their effects are instantaneous. In addition, we added instructions facilitating multiprocessor synchronization – i.e., locks and barriers – to the instruction set.

Finally, we added support for our localization and globalization instructions, modifying the cache-coherence protocol as described in Section 5. We chose the **implicit** method of localization, which has some ramifications for our results, as explained below.

We only simulated the data cache and we modeled its parameters on those of the HP PA-8700 processors, found in the Superdome: each processor has a single level of cache, 1.5 MB, four-way set associative, with a 64 byte cache block

size. Finally, the page size is 4KB, implying 64 cache blocks per page.

4.2 Discussion

We break our discussion into two parts, first describing results pertaining to coherence events and then results pertaining to memory requirements.

Coherence Events. The first set of results we are interested in is the number of coherence events that we are able to avoid by localizing all possible references.

The left side of Figure 7 shows the total number of coherence events generated by each application variant when the atoms are not sorted to increase locality (as described in Section 2). The right side of the figure shows the same when atoms are sorted to increase locality. The results are qualitatively the same, though the increased locality substantially decreases the total number of signals.

We show results for power-of-two machine sizes from 2 to 64 processors, with three bars for each machine size describing the event count of the three variants of the program. The total size of the bar is equal to the total number of events generated by that variant, while colored portions of the bar show the distribution of the types of signals generated. While "BusRead" and "BusReadX" signals derive from the unmodified MSI protocol, "BusLocal" is a new signal required by our modifications, as described in Section 5. Note that the "BusUpdate" signal, another new addition, is not counted as it always coincides with a BusRead or BusReadX event, and is thus already accounted for.

As expected, the **lock** version generates many more signals than either of the other two. Writes to data protected by the lock invalidate the data in other caches, causing them to generate signals upon re-reading the data, and so on. The more caches, the worse the problem gets.

The **nolock** version is interesting in that the total number of signals, almost exclusively BusReads, does not vary much as the number of processors grows. Due to the read-only nature of the data, adding processors does not translate into additional signals. The number of signals correlates very strongly with the total number of cache misses, which also does not vary much with the number of processors.

Finally, the **local** version generates a very small number of signals, nearly all BusLocals, though the total number grows as processors are added. As with **nolock**, the number of cache misses remains nearly constant, but not every cache miss causes a signal. The reason for the growth in the number of signals is somewhat subtle. Suppose two atoms A and B, both residing on processor P, each interact with atom C. C must be localized, but only once. Now suppose that doubling the processors means that A and B end up on different processors, P and Q respectively. Now C must be localized twice, once for P and once for Q.

The above reasoning implies that even fewer total signals would be generated by the **local** version if no signal were necessary to indicate the beginning of localization for an address, and no references needed to be added to the program by the compiler. That situation would favor the **explicit** approach for localizing instructions, described in Section 5.

Memory Usage. The second set of results gets at the cost of localization in terms of local memory usage. The left side of Figure 8 shows the *maximum* local memory usage for any single processor when the atoms are not sorted, and
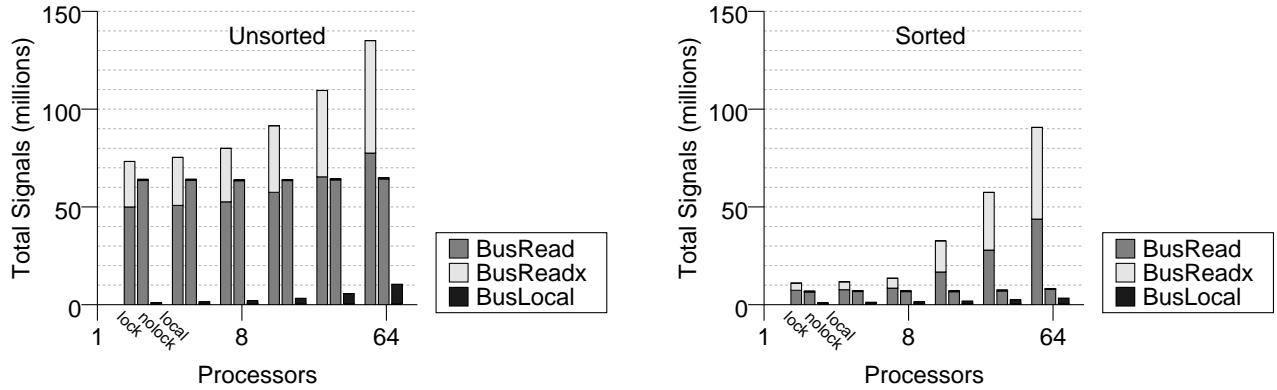
Figure 7: Total coherence events generated by all processors for the three application variants: lock, nolock, and local. For each variant the total signal count is broken further into classes: BusRead, BusReadX, and BusLocal.
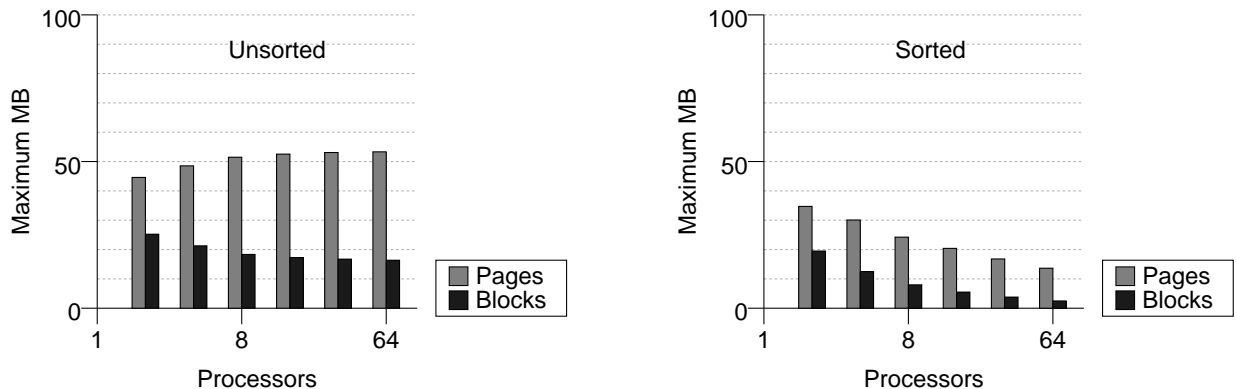

Figure 8: Maximum local memory usage of any one processor.

the right side of Figure 8 shows the usage when atoms are sorted for locality. For each processor count, two results are shown, the first describing the bytes required for a page-based configuration of local memory, and the second showing the requirement for a block-based configuration.

The unsorted application variant requires about 50MB worth of pages per processor as opposed to a little over 15MB worth of blocks (at 64 processors). Also favoring a block-based solution to local memory storage is that, while the number of pages per processor increases as processors are added, the number of blocks per processor decreases. On the other hand, recall that there are 64 blocks per page in this implementation, so a page-based solution requires substantially fewer searchable entries than a block-based solution.

As we would expect, memory requirements are much reduced when the data is sorted to increase spatial locality. Both the number of pages and the number of blocks diminish as the number of processors increases, though the rate of decrease is faster for blocks. The ratio of pages to blocks, while increasing, still favors a page-based solution if the number of searchable entries is a deciding factor.

We describe potential realizations of these memory organizations in Section 5.2.

5. IMPLEMENTATION

In Section 3, we introduced instructions that logically move data between local and global memory with the promise that data need not be physically moved. In this section, we describe possible implementations of these instructions in hardware, dividing the discussion into three parts. First we describe our modifications to a basic cache-coherence protocol. Then we consider the organization of the local memory that contains localized data. Finally, we examine the hardware required to perform reductions.

5.1 Coherence Protocol Modifications

By tying our implementation to an existing coherence protocol, we are able to use existing mechanisms to fetch data at a very fine granularity. However, changes to the protocol are necessary to ensure that localized data evicted from the cache does not have to be refetched from global memory. The basic implementation idea is this: mark localized addresses in the cache; if the data is evicted, store it in *local* memory so that future references will not generate protocol traffic.

We now describe, at a high level, modifications to the "MSI" cache-coherence protocol [10] to realize this goal. Lower level details, including state diagrams, can be found

in [15]. In order to simplify our discussion, we assume snoopy broadcast cache-coherence, though we believe our ideas extend equally well to directory-based protocols.

The basic MSI protocol remains intact: the *global* states in which a cache block can be – "Modified," "Shared," and "Invalid" – and the transitions between the states are exactly the same. To this base, we add three new states, essentially mirrors of the original states, but for localized data. We call them "Local Modified," "Local Shared," and "Local Invalid." The transitions between the new states closely mirror the transitions between the old states.

The transitions between new and old states represent transitions between local memory and global memory, and result from the execution of the instructions described in Section 3. The exact nature of these transitions depends on the choice of localization approach – i.e., **implicit** versus **explicit** – so here we divide our discussion between the two.

Implicit. Recall that in the implicit approach an annotation on a load or store instruction indicates that the instruction should reference local memory, and a *normal* load or store, issued by any processor, moves the data back to global memory. So in this model, transitions from "global" MSI states to "local" MSI states are determined by the first annotated load or store to an address, and the transition back is prompted by a normal load or store issued by any processor.

While a processor needs only to inspect the instruction it is executing to determine whether to look for data in local or global memory, it also must be aware of other processors' requests, in case one requires a transition back to global memory. This requirement implies that local memory must be snooped on every "BusRead" and "BusReadX" signal. This in turn implies that any localization information kept must be available to the snooping device, and that every snoop must determine whether data has been localized before responding. So this approach has the potential for making every snoop response a little slower.

The approach also requires two new signals. The first, call it "BusLocal," is necessary to indicate to other processors that a given address is being localized and therefore should be invalidated. The invalidation is necessary to ensure that other processors generate a signal – i.e., don't find data in their cache – if they issue a normal load or store to the address. A simple BusReadX would not work because other processors may have already localized the address, and be listening for that signal to end its localization.

The second signal, called "BusUpdate," is necessary to indicate that flushed data should be combined, in a reduction, with data from memory, rather than simply replacing memory. We describe the reduction mechanism further in Section 5.3.

Under the implicit approach, a localization cycle looks like this:

1. Processor executes annotated reference.

2. Coherence mechanism sends a BusLocal signal, and localizes address. Coherence mechanism continues to snoop signals for that address, even after a possible eviction of the block to local memory.

3. Cache ends localization after a normal load or store, or a BusRead or BusReadX signal, initiating a flush if

the data has been modified and asserting BusUpdate if the data is part of a reduction.

Explicit. Recall that the explicit approach calls for explicit control instructions that mark the beginning and end of a localization era. In this approach, the control instructions would initiate transitions from "global" MSI states to "local" states and back again.

Since localization information is not conveyed through instruction choice, whether to look for data in local or global memory must be determined via external information, i.e., every cache miss results in a table lookup to determine whether the data has been localized. Further, since the coherence mechanism must know that data is *not* local by the time it is ready to generate a signal, this approach has the potential to slow down every miss. On the other hand, the local memory is also potentially significantly *speeding up* the response to misses. Provided important references are localized, this gain would more than balance out the slow-down for references that are not localized.

Furthermore, in this approach snooping of local memory is not necessary, provided we add a signal to the protocol – call it "BusGlobal." The processor that executes the control instruction that ends the localization for an address issues this signal to convey that information to the remaining processors.

The explicit approach also needs the BusUpdate signal mentioned in the discussion for the implicit approach, to indicate that an address is involved in a reduction.

A localization cycle in this approach looks like this:

1. Processor executes explicit start instruction.

2. Coherence mechanism generates a signal if necessary (only if the line is not present on a read), and marks the block local.

3. Coherence mechanism ignores signals for that address until explicit stop instruction, or BusGlobal signal, initiating a flush if the data has been modified and asserting BusUpdate if the data is part of a reduction.

As we have noted in Section 4, the lack of a signal to indicate the beginning of a localization could result in the overall generation of fewer signals under this approach, provided extra references need not be added to the code simply to ensure that a localization region ends.

5.2 Local Memory

As we have seen above, the organization of local memory can have a significant effect on the performance of the coherence protocol changes we have proposed. In particular, if the local memory must be snooped on every transaction, the organization must be able to provide negative search result information quickly. We explore three potential organizations below:

Per-cache-block storage. In this organization, local data would be stored as it is in a hardware cache, on a per-cache-line basis. However, the need to guarantee the presence of localized data appears to require fully associative lookup properties that are prohibitively expensive to implement for large memories. Recent work on software-managed caches [11] proposes improving the scalability of fully associative memory structures via a hashing scheme. However,

while such a scheme might provide good performance for the average case, a fundamental drawback for our purposes could be the lack of a reasonable maximum time bound for negative search results. Another drawback is that it puts a limit on the amount of local memory.

Per-page storage, virtual memory. In this organization, local memory is organized as if it were *the* memory of a uniprocessor, with page table, translation lookaside buffer (TLB) and software management of page faults. Every page in the virtual address-space then potentially maps to two physical pages: a page in the *global* physical address-space and a page in the *local* physical address-space. Where the page actually resides for a given processor is determined by locality information kept in the local page table entry, and therefore is cached in a local TLB.

One benefit of this approach is that it provides unlimited virtual local memory. Another is that, provided the localized data exhibits good locality, it can result in a factor of *pagesize/blocksize* fewer total tags to search, since we would be searching for pages rather than blocks.

The main drawback to this organization is again the lack of an upper-bound guarantee for negative search results, given the potential for TLB misses and software replacements during the search.

Per-page storage, fixed memory. A possible compromise organization is a fixed amount of storage organized into pages. Again, if the localized data exhibits good locality, then this organization results in a factor of *pagesize/blocksize* fewer total tags to search than the per-cache-block storage scheme. Perhaps, again depending on the amount of storage required, these tags could be stored in a fully-associative structure.

We have analyzed memory usage results for one application in Section 4. To arrive at the optimal memory organization for a particular implementation, similar analysis of data for a representative set of applications is required, while taking into account constraints (such as memory size) imposed by the implementation.

5.3 Reductions

Finally, the hardware we envision to perform the reductions looks something like the Tree Module in IBM's Blue Gene/L system, described in [1]. In that implementation an entire network is devoted to a combining tree, and reductions occur in logic located in that network's per-processor interface.

Our implementation has similar logic, located in the coherence interface. As the result of the BusUpdate signal described above in Section 5.1, the requesting processor receives *multiple* responses, one from memory, and one from all processors that have updates. The requesting processor's network interface is responsible for combining results and delivering them to the processor. Details such as how the reduction operator and operand-size are conveyed are still under investigation.

We note that responses are in the form of *blocks* of data, so care must be taken to ensure that all data in the block should actually be involved in the reduction. As mentioned earlier, we believe that compilers can handle these alignment issues based on the user-directives described in Section 3.

6. RELATED WORK

A great deal of work has been done on distributed shared memory systems, with different approaches focusing on different weaknesses of snoopy broadcast-based shared memory systems. Hardware-controlled, usually cache-line based, approaches attack scalability concerns by replacing the broadcast-based snoopy cache coherence protocol with point-to-point message-based directory protocols. (Unfortunately, as we noted in the introduction, these systems lose the "data-follows-computation" characteristic the larger they get.) At the other extreme, software-controlled systems, generally page-based, attempt to give programmers the benefits of a shared address space without the high cost. Hybrid systems add minimal hardware in an attempt to decrease the cost of hardware controlled systems while improving on the performance of purely software-controlled systems.

While the notion of "local" memory is an integral feature of distributed shared memory systems, in general the local memories are only considered as components of the aggregate global address space. Another feature of our work that immediately distinguishes it is its applicability to both snoopy- and directory-based coherence protocols.

Of particular relevance to our efforts is work on hybrid systems done by the Wisconsin Wind Tunnel project. Cooperative Shared Memory [13] describes user-level "check-in" and "check-out" directives similar in spirit to the **explicit** model we described in Section 3. Aside from the hardware targeted, there are two major distinctions between our directives and their directives: first, in their system the processor that issues a check-out must be the same processor that checks it in. Second, there is no directive to describe and implement reductions.

Later work on the Tempest [18] system proposed user-programmable software coherence protocols on top of local memory. While in this system protocols could be programmed on a per-application basis to perform reductions, gone was the notion of user-level coherence directives. Loosely Coherent Memory [14] proposes mechanisms that a compiler could invoke that, in conjunction with the programmable coherence protocols provided by Tempest, could be used to perform reductions. Separate work done at Illinois proposes hardware to perform such compiler-found reductions in cache, as opposed to local memory [22].

Our approach, so far as we know, is the first to propose user-control of hardware-based cache-coherence protocols to take advantage of local memory.

More recent related work stems from attempts in the OpenMP community to extend the standard with specific support for distributed shared memory platforms. For example, [5] describes HPF-like extensions to partition data and computation, while other work advocates runtime page migration [17]. One sub-branch of particular relevance to our work advocates privatization of memory references in software [20, 8]. That work suffers in comparison to ours in its need to explicitly copy data from global to private memory. Also, while it is clear that their approach would apply well to coarse-grained applications, it does not appear to apply equally well to fine-grained applications. For example, in applications like ammp it is not clear what data should be copied into private memory until it is actually needed.

7. CONCLUSION

We have introduced a hybrid address-space architecture that, in tandem with a high-level programming model, combines the scalability of a distributed address-space architectures with the programmability and superior fine-grained performance of shared address-space architectures.

We motivated our approach through an analysis of a specific application and its performance on currently available address-space architectures. We then described a programming model that allows users to, at a very high level, logically move global data to local memory without physically moving it, and we provided potential hardware implementations that do not require substantial changes to an existing cache-coherence protocol. Our initial experiments showed that effective placement of these directives in an application reduces coherence communication by more than a factor of 10 for 64 processors.

Although our experiments were limited to a single application, we believe our technique is well-suited for irregular applications in general. The lack of knowledge available for these applications maps well to the limited knowledge required in order to perform localization.

As future work we plan to look more closely at the applicability of our ideas to this class of applications as a whole. Additionally, while we believe we have shown that implementation is possible in principle, future work will focus on the implementation details, and estimate actual performance with analytic models and a timing simulator.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] N. Adiga et al. An Overview of the Blue Gene/L Supercomputer. In *Proc. 2002 ACM/IEEE Conf. Supercomputing*, 2002.

[2] F. Allen et al. Blue Gene: A vision for protein science using a petaflop computer. *IBM Systems Journal*, 2001.

[3] V. Aslot and R. Eigenmann. Performance Characteristics of the SPEC OMP2001 Benchmarks. In *Proc. European Workshop on OpenMP (EWOMP'2001)*, 2001.

[4] J. Barnes and P. Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 1986.

[5] J. Bircsak et al. Extending OpenMP For NUMA Machines. In *Proc. 2000 ACM/IEEE Conf. Supercomputing*, 2000.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Ver 2.0. Technical report, University of Winsconsin-Madison, 1997.

[7] S. D. S. Center. Blue Horizon. http://www.sdsc.edu/Resources/bluehorizon.html.

[8] B. Chapman, A. Patil, and A. Prabhakar. Performance Oriented Programming for NUMA Architectures. In *Proc. Int'l Workshop on OpenMP Applications and Tools (WOMPAT 2001)*, 2001.

[9] A. Charlesworth. Starfire: Extending the SMP Envelope. *IEEE Micro*, 1998.

[10] D. Culler and J. Singh. *Parallel Computer Architecture A Hardware/Software Approach*. Morgan Kaufman, 1999.

[11] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proc. 27th Annual Int'l Symp. on Computer Architecture*, 2000.

[12] Hewlett-Packard Company. *Meet the HP Superdome Servers*, 2002.

[13] J. R. Larus, S. Chandra, and D. A. Wood. CICO: A Shared-Memory Programming Performance Model. In *Portability and Performance for Parallel Processors*. John Wiley & Sons, Ltd., 1994.

[14] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1994.

[15] C. McCurdy and C. Fischer. Details of Cache-coherence Protocol Modifications for Hybrid Shared Memory. http://www.cs.wisc.edu/~cmccurdy/hybid.ps, 2002.

[16] C. McCurdy and J. Mellor-Crummey. An Evaluation of Computing Paradigms for N-body Simulations on Distributed Memory Architectures. In *Proc. Seventh Symp. on Principles and Practice of Parallel Programming*, 1999.

[17] D. Nikolopoulos et al. Is Data Distribution Necessary in OpenMP. In *Proc. 2000 ACM/IEEE Conf. Supercomputing*, 2000.

[18] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Typhoon and Tempest: User-Level Shared Memory. In *Proc. 21st Int'l Symp. on Computer Architecture*, 1994.

[19] H. Saito et al. Large System Performance of SPEC OMP2001 Benchmarks. In *Proc. WOMPEI2002, the Workshop on OpenMP: Experiences and Implementations*, 2002.

[20] A. J. Wallcraft. SPMD OpenMP vs MPI for Ocean Models. In *Proc. First European Workshop on OpenMP (EWOMP 1999)*, 1999.

[21] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methedological Considerations. In *Proc. 22nd Annual Int. Symp. on Computer Architecture*, 1995.

[22] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Reduction Parallelization and Optimization in DSM Multiprocessors. In *Proc. 1st Workshop on Parallel Computing for Irregular Applications*, 1999.