A DUAL ADDRESS SPACE ARCHITECTURE: IMPLEMENTATION AND EVALUATION

by

Collin B. McCurdy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2008

For those who believed in me.

ACKNOWLEDGMENTS

Thank you to my advisor, Charles Fischer, who has steadily guided me through this process for all these years, giving me freedom to pursue my research interests but also direction when necessary.

Many thanks to all my committee members, David Wood, Mary Vernon, Barton Miller and Zhengyu Liu, each of whom contributed enormously to this dissertation by forcing me to think about its contents from their perspective. David Wood's contributions to the architecture portions of this work were especially notable and greatly appreciated.

Thanks to all my colleagues at Oak Ridge National Laboratory. Special thanks are due to Scott Studham, who first brought me to the lab; and to Jeffrey Vetter, leader of the Future Technologies group, who has kept me there. I also owe a huge debt of gratitude to NASA Ames for providing me with access to computational facilities over the past three years.

Thanks to my teammates on the "Oak Ridge Red" and the"ORNL Green" for putting up with my rather distracted play the last couple of months, and to Ron Felix at the Knoxville Yoga Center for helping me to maintain my balance during this process.

Warmest thanks and love to my family: my brother Neil (who showed me how it's *done*) and his lovely wife Michelle; my favorite kiddies Collette, Ian and baby Annalise; and finally, especially, Mummy, the strongest of women who never allowed me to give up.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

ABSTRACT ix

1 Introduction 1

1.1 Artifactual Communication 2
1.2 The Dual Address Space Architecture 3
1.3 Example 1: False Sharing Model 5
1.3.1 Localization 6
1.3.2 Reducing Inherent Communication 8
1.4 Example 2: AMMP 10
1.4.1 Distributed Implementation 12
1.4.2 Performance 13
1.4.3 Localization 14
1.5 Contribution 15

2 Related Work 17

2.1 Reducing Artifactual Communication 18
2.1.1 Addressing False Ownership 18
2.1.2 Addressing False Sharing 19
2.2 Software Control Over Coherence 21
2.3 Transactional Memory 23
2.4 Address Spaces 23
2.5 OpenMP Extensions 24

3 Architecture and Programming Model 25

3.1 Dual Address Space 25
3.2 New Memory Instructions 27
3.2.1 Local Memory Instructions 27
3.2.2 Global Memory Instructions 28

Page

3.2.3 Update Instruction . . . 28
3.2.4 IA-64 Extension . . . 29
3.3 Memory Model . . . 29
3.3.1 Data-Race-Free-0 . . . 30
3.3.2 Localized-Data-Race-Free-0 . . . 30
3.3.3 Extending Other SCNF Models . . . 32
3.4 Programming Model . . . 33
3.4.1 Data Partitioning . . . 34
3.4.2 Communication . . . 36
3.4.3 Higher-level Interface . . . 38
3.5 Summary . . . 39

4 Implementation . . . 40

4.1 Node Design . . . 40
4.1.1 Processor Interface . . . 41
4.1.2 Memory and Directory Interface . . . 42
4.1.3 Network and I/O Interfaces . . . 42
4.2 Cache Coherence Protocol . . . 42
4.2.1 Abstract Protocol . . . 42
4.2.2 Directory Protocol . . . 46
4.3 Block-based Storage . . . 52
4.3.1 Multiple Local Writers . . . 54
4.3.2 Operator Reductions . . . 54
4.4 Memory Consistency . . . 56
4.4.1 Global Consistency Model and Enforcement . . . 56
4.4.2 Localization . . . 57
4.5 Local Memory . . . 58
4.5.1 Local Memory as *Memory*. . . . 58
4.5.2 Local Memory as *Cache*. . . . 59
4.6 Summary . . . 61

5 Simulation Platform . . . 62

5.1 Simulator Prototype . . . 63
5.2 Full-Scale Simulator . . . 65
5.2.1 Handling Multiple Outstanding Loads . . . 65
5.2.2 Other Micro-architectural Considerations . . . 67
5.3 Simulator Parallelization . . . 68
5.3.1 Network Parallelization . . . 69

Page

5.3.2 User-Controllable Coherence . . . 70
5.4 Simulator Validation and Performance . . . 71
5.4.1 Validation . . . 72
5.4.2 Performance . . . 73
5.5 Summary . . . 74

6 Application One: HYCOM . . . 75

6.1 Parallelization . . . 76
6.2 Artifactual and Inherent Communication in HYCOM . . . 78
6.2.1 SPMD HYCOM . . . 78
6.2.2 Artifactual: Multi-dimensional Domain Decompositions . . . 79
6.2.3 Inherent: Communication Frequency Optimization . . . 80
6.3 Experimental Evaluation . . . 81
6.3.1 Communication . . . 82
6.3.2 Performance . . . 84
6.4 Summary . . . 85

7 Application Two: Fast Multipole Method . . . 87

7.1 Algorithm . . . 88
7.2 Parallelization Strategies . . . 89
7.2.1 Message Passing: MPI . . . 89
7.2.2 Shared Memory: OpenMP . . . 92
7.3 Experiments . . . 94
7.3.1 Communication . . . 95
7.3.2 Performance . . . 96
7.4 Implications for the Dual Address Space Architecture . . . 98
7.5 Summary . . . 99

8 Conclusions . . . 101

LIST OF REFERENCES . . . 106

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

1.1 The artifactual communication model: (a) code fragment describing the model implementation, and (b) performance of various methods of addressing the problem normalized to the performance of a standard MESI cache coherence protocol on a release consistent processor. 5

1.2 Two approaches to reducing artifactual communication through the use of multiple address spaces: (a) localization followed by globalization, and (b) localization with an explicit update. 7

1.3 Using redundant computation to reduce synchronization: (a) in a dual address space, and (b) using multidimensional arrays under a normal protocol. 9

1.4 (a) Original uniprocessor algorithm and (b) the algorithm transformed for distributed memory. 11

1.5 Results from runs on an SGI Altix at NASA Ames: (a) lock, (b) no-lock, (c) MPI. . . . 13

1.6 Improvement from localization, focusing on atom-atom computations, results normalized to localized version: (a) cache misses that result in communication , (b) performance. 14

3.1 A thread's-eye view of the standard global address space (left), and of the Dual Address Space (right). 26

3.2 Example of a cross-thread local-to-local causal conflict. 31

3.3 Using a Dual Address Space to: (a) create a data partition, (b) repartition data. 33

3.4 Using a Dual Address Space to implement communication patterns: (a) producer/consumer, (b) reduction. 36

4.1 Block diagram of a node in a Dual Address Space implementation, based on that of the SGI Altix and its predecessor, the Origin 2000. 41

Figure Page

4.2 Basic MSI protocol state diagram (based on Culler and Singh [17]). 43

4.3 Extensions to the MSI protocol to enable transparent use of local memory. 44

4.4 Pictorial descriptions of the complete set of transactions necessary to maintain coherence in the four protocol phases described in the text. Types 1, 2 and 3 may be familiar as the transactions in a standard protocol, while types 4, 5 and 6 are new. 46

4.5 Reduction algorithm. 53

5.1 Prototype code. 63

5.2 Per-processor cycle counts predicted by the simulator vs. those measured by performance counters on hardware. 71

5.3 Per-processor cache miss rates predicted by simulator vs. those measured on hardware: (a) L3 cache, (b) L2 cache. Note that the simulated L3 miss rates are much more accurate, and that, as discussed in the text, low-latency L2 inaccuracies have more effect on the simulated performance of *one-d*. 71

5.4 Performance of the parallel simulator, as measured in simulated cycles per second (per processor). The x-axis labels indicate the number of simulated threads followed, in parentheses, by the number of processors performing the simulation. 73

6.1 Typical loop-nest from the MPI version of HYCOM. 76

6.2 The sample loop-nest transformed for SPMD-style OpenMP. 78

6.3 Results from HYCOM runs using the GLBA0.72 data set on an SGI Altix BX2. Here, as in all figures in this chapter, "cycles" indicates the cycle count measured by performance counters on the last processor to complete one time-step (the second). 79

6.4 Results from runs using the GLBA0.08 data set, a larger data set which enables the use of more processors. 80

6.5 Last-level cache miss categorization for small (top) and medium (bottom) data set. . . 83

6.6 (a-b) Performance for small data set, comparing (a) different localization strategies, and (b) the best strategy with the standard protocol. (c) Performance for medium data set, versus the standard protocol 84

Figure Page

7.1 Example of an adaptive *quad*tree. 88

7.2 (a) The steps of the adaptive FMM algorithm and their division into *Tree-code* and *Interactions* summary categories. (b) Interaction lists for a box in the example quad-tree from Figure 7.1. 90

7.3 Inspection and communication in the MPI implementation. 91

7.4 Rough comparison of bytes communicated by MPI vs OMP (128 procs). 95

7.5 Results for 512 processor Altix Bx2 at SGI (3rd iteration): overview (a), and detailed: (b) OpenMP tree-code, (c) OpenMP interactions, (d) MPI tree-code, (e) MPI interactions. 97

ABSTRACT

While a hardware supported shared address space offers programmability advantages and better performance for fine-grained applications, the very mechanisms that create those advantages appear to prevent the machines that use them from scaling to large numbers of processors. For this reason, hardware-based distributed shared memory (DSM) platforms have largely been dismissed in the context of scientific computing. This dissertation proposes changes to a hardware-based DSM architecture that allow users to use two address spaces to gain the scalability of distributed architectures while retaining the benefits of the shared address space architecture.

The thesis of this dissertation is that the Dual Address Space Architecture provides both an efficient mechanism for enabling increased user and compiler control over data consistency and data locality, *and* a high-level interface for programmers to conceptualize and take advantage of the mechanism. The Dual Address Space Architecture thus provides users and compilers an efficient means, particularly appropriate to the structure of scientific applications, of keeping data consistent and local, enabling improved performance over current global address space implementations.

The dissertation evaluates its thesis in two parts. The first part demonstrates the feasibility of the ideas by describing the details of the high-level architecture, including the programming model, and its implementation via extensions to a standard directory-based cache coherence protocol. The second part evaluates the performance of Dual Address Space implementations of two applications that currently underperform on distributed address space platforms. First, cycle-accurate simulation results indicate that a Dual Address Space version of HYCOM, an ocean model which features an irregular data decomposition, significantly reduces the number of last-level cache misses that

cause communication, resulting in substantial improvement in performance. Second, experiments with two implementations (distributed memory and shared memory) of the Fast Multipole Method, a tree-based algorithm for solving N-body problems, expose weaknesses in both that would be cleanly and efficiently addressed by a Dual Address Space implementation of the algorithm.

A DUAL ADDRESS SPACE ARCHITECTURE: IMPLEMENTATION AND EVALUATION

Collin B. McCurdy

Under the supervision of Professor Charles N. Fischer
At the University of Wisconsin-Madison

While a hardware supported shared address space offers programmability advantages and better performance for fine-grained applications, the very mechanisms that create those advantages appear to prevent the machines that use them from scaling to large numbers of processors. For this reason, hardware-based distributed shared memory (DSM) platforms have largely been dismissed in the context of scientific computing. This dissertation proposes changes to a hardware-based DSM architecture that allow users to use two address spaces to gain the scalability of distributed architectures while retaining the benefits of the shared address space architecture.

The thesis of this dissertation is that the Dual Address Space Architecture provides both an efficient mechanism for enabling increased user and compiler control over data consistency and data locality, *and* a high-level interface for programmers to conceptualize and take advantage of the mechanism. The Dual Address Space Architecture thus provides users and compilers an efficient means, particularly appropriate to the structure of scientific applications, of keeping data consistent and local, enabling improved performance over current global address space implementations.

The dissertation evaluates its thesis in two parts. The first part demonstrates the feasibility of the ideas by describing the details of the high-level architecture, including the programming model, and its implementation via extensions to a standard directory-based cache coherence protocol. The second part evaluates the performance of Dual Address Space implementations of two applications that currently underperform on distributed address space platforms. First, cycle-accurate simulation results indicate that a Dual Address Space version of HYCOM, an ocean model which features an irregular data decomposition, significantly reduces the number of last-level cache misses that cause communication, resulting in substantial improvement in performance. Second, experiments with two implementations (distributed memory and shared memory) of the Fast Multipole Method,

a tree-based algorithm for solving N-body problems, expose weaknesses in both that would be cleanly and efficiently addressed by a Dual Address Space implementation of the algorithm.

Charles N. Fischer

Chapter 1

Introduction

The trend in large-scale High Performance Computing systems design has been toward maximizing peak performance by connecting as many microprocessors as possible with the fastest interconnect available, leaving the management of the distributed address space to applications. An undesired side-effect of this trend has been to narrow the range of applications that can take full advantage of the hardware.

In order to be successful on these platforms, applications require a simple *parallelization strategy*, as embodied by data decomposition (how data is distributed across processors), computation partitioning (how computation is divided among processors), and communication patterns (how required non-local data is made local). An example of an application that maps well to the largest platforms available today is S3D [33], which models combustion through fluid dynamics simulation. The application uses an explicit stencil-based solver, implying only nearest-neighbor communication, and a regular, static distribution of data.

Molecular dynamics applications [59, 62, 60], on the other hand, are not well-suited for these platforms. In addition to dynamic data distributions and communication patterns, a particular problem for the scalability of these simulations is long-range force computation: on distributed hardware, the most practical solution requires transformation to Fourier space, and the three-dimensional FFT requires communication equivalent to a transpose. While tree-based solutions exhibit significantly more locality, they are not practical to implement in distributed memory.

Notably, even applications with only slightly more complex parallel behavior than S3D have difficulty mapping onto distributed hardware. An example is HYCOM [11], an ocean model that,

like S3D, features a static data distribution and uses an explicit stencil-based solver. However, the distribution of data across processors is slightly complicated by land-avoidance, resulting in load imbalance which in turn reduces scalability.

While a hardware-maintained shared address space offers programmability advantages and improved performance for applications with irregular data distributions and communication patterns, the very mechanisms that create those advantages appear to prevent machines that use them from scaling to large processor counts. For this reason, hardware-based distributed shared memory (DSM) platforms have largely been dismissed in the context of scientific computing.

However, the confluence of two trends in High Performance Computing may serve to resurrect interest in DSM research for scientific computing. The first is the challenge presented by multicore microprocessor architectures. Current multicore implementations use broadcast-based protocols to maintain cache coherence, which are known not to scale beyond a few 10's of processors. Projections call for hundreds of cores per socket in the near future [6]. The second is the growing recognition that a global address space can simplify parallel programming, making users more productive [74].

1.1 Artifactual Communication

The primary challenge facing applications attempting to scale-up on DSM implementations like the SGI Altix [75] is artifactual communication. The term (which originates with Culler and Singh [17]) refers to unnecessary communication due to unintentional interactions with the extended memory hierarchy that implements the global address space in hardware, as distinct from inherent communication required by the application.

False Sharing The most notorious example of artifactual communication is "false sharing." This phenomenon arises when multiple processors write *different* words co-located in a cache block, causing ownership of the block, and the data, to ping-pong between the processors.

False sharing is a particular problem for scientific applications that make use of multi-dimensional domain decompositions in order to reduce communication volume.

The root cause of false sharing is eager invalidation-based cache coherence, which helps maintain the *memory consistency model* visible to software. The consistency model determines when the effects of writes by each processor are visible to other processors. Programmers intuitively expect *sequential consistency*, which has significant impact on performance.

Relaxed consistency models can improve the situation without placing too much burden on programmers. However, *full* advantage of the relaxation cannot be leveraged when the underlying coherence protocol mechanism remains unchanged: the coherence protocol continues to eagerly propagate actions for write instructions, regardless of whether the actions are required by the consistency model.

False Ownership A less widely understood aspect of artifactual communication arises on DSM implementations, like the Altix, where the protocol *home* for data is determined independently of the parallelization strategy.

"False ownership" results when there is a mismatch between the owner as defined by the coherence protocol and the owner as defined by the data decomposition. The situation results in communication for every cache miss, potentially severely degrading performance.

False ownership exacerbates the false sharing problem for applications using multi-dimensional domain decompositions: if a block owned in the data-parallel sense but not according to the protocol is evicted from the cache due to false sharing, both the communication that causes the eviction and the communication to get it back must go through a third node, the home.

False ownership is also problematic for applications that attempt to dynamically modify their data decomposition: if there is no way to change the home of the data, then the true owner becomes a false owner.

1.2 The Dual Address Space Architecture

This dissertation proposes modifications to a hardware-based distributed shared address space architecture that enable users and compilers to address both types of artifactual communication. It

conjectures that shared and distributed address space architectures are actually just endpoints on a spectrum, and considers a midpoint: a machine with both address spaces.

In a Dual Address Space there are two kinds of memory: global, which hardware keeps coherent with a standard coherence protocol, and strictly local, which is left incoherent. Users are provided with a mechanism for specifying, at word-granularity, data that would benefit from localization – being moved from global to local memory – yielding the following benefits for memory accesses that have been localized:

- Control over the placement of data in the memory hierarchy.
- Elimination of false sharing.
- An efficient global reduction mechanism.

A Dual Address Space implementation takes advantage of User-Controllable Coherence [51], which proposes modifications to a standard hardware-based coherence protocol. The new protocol manages the two address spaces and the transition of data between them, enabling efficient, dynamic movement of data between local and global address spaces, without copying. By providing users and/or compilers limited control over the coherence protocol mechanism, User-Controllable Coherence offers complete control over both the consistency policy and the locality of data.

The following two sections demonstrate, with the aid code examples, how increased control over data placement and the consistency policy can enable improved performance. The first example illustrates with a micro-benchmark how false sharing and false ownership can be avoided in applications like HYCOM which feature irregular data distributions but more regular computation and communication patterns. The second example uses a benchmark fragment to show how localization along with the reduction mechanism can be used to efficiently manage data partitions in applications, like molecular dynamics, which feature more dynamic data distributions and irregular communication patterns.

```
#define SIZE 16  // doubles per cache block
double a[SIZE] __attribute__ ((aligned(128)));
double b[SIZE] __attribute__ ((aligned(128)));

        ...

        if (myid == 0) {
    // init a & b
        }
        else if (myid == 4) {
            first = 1;
            last = 8;
        }
        else if (myid == 8) {
            first = 8;
            last = 15;
        }

        if (myid == 4 || myid == 8) {
            for (j = 0; j < n; j++) {
                for (i = first; i < last; i++) {
                    a[i] = (b[i-1]+b[i]+b[i+1])/3;
                }
                SYNCH();

                for (i = first; i < last; i++) {
                    b[i] = (a[i-1]+a[i]+a[i+1])/3;
                }
                SYNCH();
            }
        }
```

(a)

	Inst	Communication	Cycles
standard	1.00	1.00	1.00
inv	1.04	0.84	0.90
update	1.02	0.60	0.53
red-inv	1.17	0.30	0.31
red-upd	1.16	0.23	0.21
multidim	1.17	0.30	0.25

(b)

Figure 1.1 The artifactual communication model: (a) code fragment describing the model implementation, and (b) performance of various methods of addressing the problem normalized to the performance of a standard MESI cache coherence protocol on a release consistent processor.

1.3 Example 1: False Sharing Model

The purpose of the false sharing model is to hold a magnifying glass to a situation in which artifactual communication has been maximized, enabling a clear demonstration of how localization can be used to combat it. The code fragment in Figure 1.1(a) describes the model. To induce false sharing, two threads – pinned to processors P_4 and P_8, and located on different nodes since there are four processors-per-node – perform a simple stencil computation, each computing and storing results for half of a cache block. The block is thus owned, in the *data-parallel* sense, by both processors. To induce false-ownership, the block is initialized, and therefore owned as far as the

coherence protocol is concerned, by a thread pinned to a third processor P_0, located on a third node.

Note that the model contains true-sharing, in addition to false sharing. The stencil simply averages each element with its nearest neighbors. To permit parallelization, the computation is broken into two parts, thus splitting the dependence between the write of a border element by one processor and the read of the same element by another processor. The explicit synchronization in between the two parts ensures that data written by one processor is ready for consumption by its neighbor in the next half-iteration.

The table in Figure 1.1(b) presents results demonstrating the effectiveness of different approaches to reducing the artifactual communication, as measured by: instructions completed, cache misses that cause communication, and most importantly, performance. The data was obtained from runs of the model code on the simulator described in Chapter 5. The results are all normalized to the performance of the standard MSI-based protocol used by the Altix and modeled by the simulator when only "global" memory instructions are used.

1.3.1 Localization

Figure 1.2(a) demonstrates a quick and dirty approach to localization, using a pragma-based mechanism to convey to the compiler which addresses should be loaded into which address space. Note that data does not actually move from address space to address space until it is referenced by a load or store; a pragma does not move data but only indicates the flavor of load or store instruction the compiler should use when referencing the indicated variable within the pragma's scope.

The idea is to keep data owned by a processor in its local address space, so at the outset each processor moves the data it owns into the local address space. At synchronization points, each processor moves data that it has written and that will be read by other processors into the global memory. This globalization occurs at the LFETCH(), where each processor touches the block with a *global* (note the lack of a localizing pragma covering this statement) prefetch instruction. The *localize* pragma before the second half iteration results in globalized data moving back into

```
    if (myid == 4 || myid == 8) {
        for (j = 0; j < n; j++) {
#pragma localize (a,b)
            for (i = first; i < last; i++) {
                a[i] = (b[i-1]+b[i]+b[i+1])/3;
            }
            SYNCH();
            LFETCH(a); // global!
            SYNCH();

#pragma localize (a,b)
            for (i = first; i < last; i++) {
                b[i] = (a[i-1]+a[i]+a[i+1])/3;
            }

            SYNCH();
            LFETCH(b); // global!
            SYNCH();
        }
    }
```

(a)

```
    if (myid == 4 || myid == 8) {
#pragma localize (a,b)
        for (j = 0; j < n; j++) {
            for (i = first; i < last; i++) {
                a[i] = (b[i-1]+b[i]+b[i+1])/3;
            }

            SYNCH();
            UPDATE(a);

            for (i = first; i < last; i++) {
                b[i] = (a[i-1]+a[i]+a[i+1])/3;
            }

            SYNCH();
            UPDATE(b);
        }
    }
```

(b)

Figure 1.2 Two approaches to reducing artifactual communication through the use of multiple address spaces: (a) localization followed by globalization, and (b) localization with an explicit update.

the local address space during the stencil computation. Finally, the block is moved back into the global address space in order to satisfy communication requirements for the next iteration.

The motion of data back and forth between address spaces can be avoided through the use of an explicit *update* instruction, as demonstrated by Figure 1.2(b). Here the localizing pragma covers all statements in the j loop, so that the block remains in the two local address spaces throughout the computation. However, after each half iteration, the explicit update instruction conveys to the protocol that each processor's version of the block should be communicated to all "sharers," and multiple writes resolved through the reduction mechanism. Note that the protocol requires synchronization on either side of the update to ensure that all protocol transactions for one computation or communication phase have completed before entering the next phase.

The second and third rows of the table in Figure 1.1(b) clearly indicate the performance benefit of keeping data localized: while *inv*, the version corresponding with Figure 1.2(a), exhibits improvement in both cache misses and cycle time over the standard protocol, *update* more than doubles that improvement in both categories.

1.3.2 Reducing Inherent Communication

As will be described in detail in Chapter 6, some applications like HYCOM take advantage of redundant computation on extended halos of non-local data to reduce communication frequency. This optimization is usually not possible for applications which take full advantage of the benefits of a global address space. When data arrays are treated as single, global entities, simultaneously operated on by multiple threads, writes associated with redundant computation cannot be performed without disturbing the data read and written by neighboring threads. Figure 1.3(a) demonstrates how localization can enable redundant computation to reduce synchronization in a Dual Address Space.

The primary modification to the fragment from Figure 1.2(a) is to strip-mine the j-loop, introducing a new k-loop. The new loop controls the bounds of the i-loops, so that they shrink on each iteration. The effect is that each processor begins computing on a localized version of the whole block but gradually shrinks the computation as it uses up "non-local" data. After all the data has been used up, the processors synchronize, exchange data via update instructions, and then start again. Note that eight elements are exchanged when the block is updated, making this a very efficient mechanism for exchange of data. On the other hand, when only one element is required, as in the non-redundant *update* case of Figure 1.2(b), the mechanism is significantly less efficient.

A more subtle change is required by the coherence protocol's mechanism for resolving multiple writers. Recall that upon issuing the update instruction, all writers send the block to the directory, which merges it with the old version of the block before sending it on to other owners. The protocol's assumption is that the writers have written distinct words in the block, and steps are taken to merge the blocks. The results are undefined if multiple processors write the same word. Thus, to avoid undefined results, processors need to selectively invalidate words they have written but do not own, by simply writing a neutral element, before the update. Since the block is in cache, these writes do not affect performance, but they do complicate the code. Here we have hidden them in the UPDATE() call.

The fourth and fifth rows of the table in Figure 1.1(b) again clearly indicate the performance benefit of redundant computation to minimize synchronization. Despite a nearly 20% increase in

```
   if (myid == 4 || myid == 8) {
#pragma localize (a,b)
       for (j = 0; j < n; ) {
           for (k = 7; k >= 0 && j < n; k--, j++) {
               if (myid == 4) {
                   upper=last+k;
                   lower=first;
               }
               else if (myid == 8) {
                   lower=first-k;
                   upper=last;
               }
               for (i = lower; i < upper; i++) {
                   a[i] = (b[i-1]+b[i]+b[i+1])/3;
               }
               k--;
               for (i = lower; i < upper; i++) {
                   b[i] = (a[i-1]+a[i]+a[i+1])/3;
               }
           }

           SYNCH();
           UPDATE(b);
       }
   }
```

(a)

```
#define SIZE 16  // doubles per cache block
double a[MAX_THR][SIZE] __attribute__ ((aligned(128)));
double b[MAX_THR][SIZE] __attribute__ ((aligned(128)));

        ...

    if (myid == 4 || myid == 8) {
        for (j = 0; j < n; ) {
            for (k = 7; k >= 0 && j < n; k--, j++) {
                if (myid == 4) {
                    upper=last+k;
                    lower=first;
                }
                else if (myid == 8) {
                    lower=first-k;
                    upper=last;
                }
                for (i = lower; i < upper; i++) {
                    a[myid][i] =
                        (b[myid][i-1]+
                         b[myid][i]+
                         b[myid][i+1])/3;
                }
                k--;
                for (i = lower; i < upper; i++) {
                    b[myid][i] =
                        (a[myid][i-1]+
                         a[myid][i]+
                         a[myid][i+1])/3;
                }
            }

            SYNCH();
            if (myid == 4) {
                for (i = last; i < SIZE-1; i++) {
                    b[myid][i] = b[8][i];
                }
            }
            else if (myid == 8) {
                for (i = 1; i < first; i++) {
                    b[myid][i] = b[4][i];
                }
            }
            SYNCH();
        }
    }
```

(b)

Figure 1.3 Using redundant computation to reduce synchronization: (a) in a dual address space, and (b) using multidimensional arrays under a normal protocol.

the dynamic instruction count, *red-inv* (the implementation of which we haven't shown but which corresponds with *inv* from Figure 1.2(a)) significantly improves the performance of *inv*. However, again, the update version performs substantially better by keeping the block in local memory.

Finally, Figure 1.3(b) demonstrates a method for achieving the same redundant computation in software under a normal coherence protocol. While this version uses multidimensional arrays to provide each processor an isolated, though still globally accessible, copy of its data and the halo, thus avoiding false sharing altogether, it gives up the advantages of the global address space. In particular, note that communication of the halo elements in the copy loop requires knowledge of both the exact processor from which the data is to be copied as well as the exact location of elements to be copied within that processor's array. While these calculations are simple in the regular case, where the data is evenly divided among processors, it can be significantly more complicated when the amount of data owned by processors differs.

Perhaps surprisingly, both the communication requirements and the performance of *multidim* are worse than those of *red-upd*, as indicated by the sixth row of the table in Figure 1.1(b). We have identified two potential explanations: first, the invalidate protocol requires two communications per block to achieve the copy while an update protocol only requires one. Second, note that there is a small window for false sharing during the copy loop: one processor is reading elements from one half of b while the other processor is writing the other half.

1.4 Example 2: AMMP

AMMP is a molecular dynamics simulator found in both the SPEC CPU2000 and the SPEC OMP2001 benchmark suites. Computation in the benchmark is dominated by non-bonded force calculations, which account for more than 96% of the execution time of the benchmark [7]. 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.

The approximation used by AMMP is a simplification of the Fast Multipole Method algorithm [28] described in detail in Chapter 7. As in the full method, force contributions of spatially co-located *clusters* of atoms a sufficient distance away from the atom under consideration are

```
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
```

(a)

```
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 interaction-list
        else
            compute forces between A and node N

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

(b)

Figure 1.4 (a) Original uniprocessor algorithm and (b) the algorithm transformed for distributed memory.

lumped together. However, rather than constructing and maintaining an octtree, where each node in the tree contains a volume and its children contains one-eighth of that volume, AMMP maintains only a single level of nodes.

Figure 1.4(a) presents a high-level representation of the uniprocessor algorithm. 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 with which A will interact are not known until immediately before the force computation.

Note also that the algorithm takes computational advantage of the symmetric nature of forces: in computing the forces exerted by atom B on atom A, most of the work necessary to compute forces exerted by atom A on atom B has also been done. The algorithm therefore computes and stores forces for both A and B at the same time.

The SPEC OpenMP implementation parallelizes the outer loop, using a simple "omp parallel for" directive. However, the work per atom is not consistent, for two reasons:

1. The symmetry optimization: since A computes forces for both A and B, B does no computation with respect to A.

2. The number of atoms contained in the nearest neighbor nodes is variable.

To address this potential load imbalance, the implementation uses 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 the cost of locality.

Additionally, the SPEC OpenMP implementation adds locks around data involved in the symmetry optimization noted above. Locks are required since the data for atom B is potentially read and updated by multiple processors at the same time. As the number of processors increases, redundant computation becomes significantly more cost-effective than the serialization caused by locking.

1.4.1 Distributed Implementation

The SPEC OpenMP implementation is not suitable for Non-Uniform Memory Access (NUMA) architectures like the SGI Altix, which require attention to data locality.

First, to take advantage of the spatial locality that exists in the data, we use a well-known *linear sorting* technique to sort atoms along a Hilbert curve [68], significantly improving locality at minimal cost. Second, we replicate the nodes, since there are relatively few of them and they are required by all processors. Third, we eliminate the *guided* scheduling parameter, which is not effective for load balancing on this platform since it does not take into account the locality of atoms. Instead we divide the algorithm into two parts as demonstrated in Figure 1.4(b).

Since the amount of work per atom in the first loop is consistent, we distribute the atom-list in a simple block distribution, allowing an even distribution of computation of the first loop across processors. The first loop both performs the atom-node calculations *and* builds an interaction list consisting of the atoms in nearest neighbor nodes for each atom. We then partition iterations of the second loop according to the amount of work per atom based on the interaction list. This repartitioning of iterations essentially results in a repartitioning of data also.

A message passing implementation is left with the additional problem of gathering non-local atom data for atom-atom interactions. Gathering non-local atoms from the gather lists proceeds in several stages. 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.

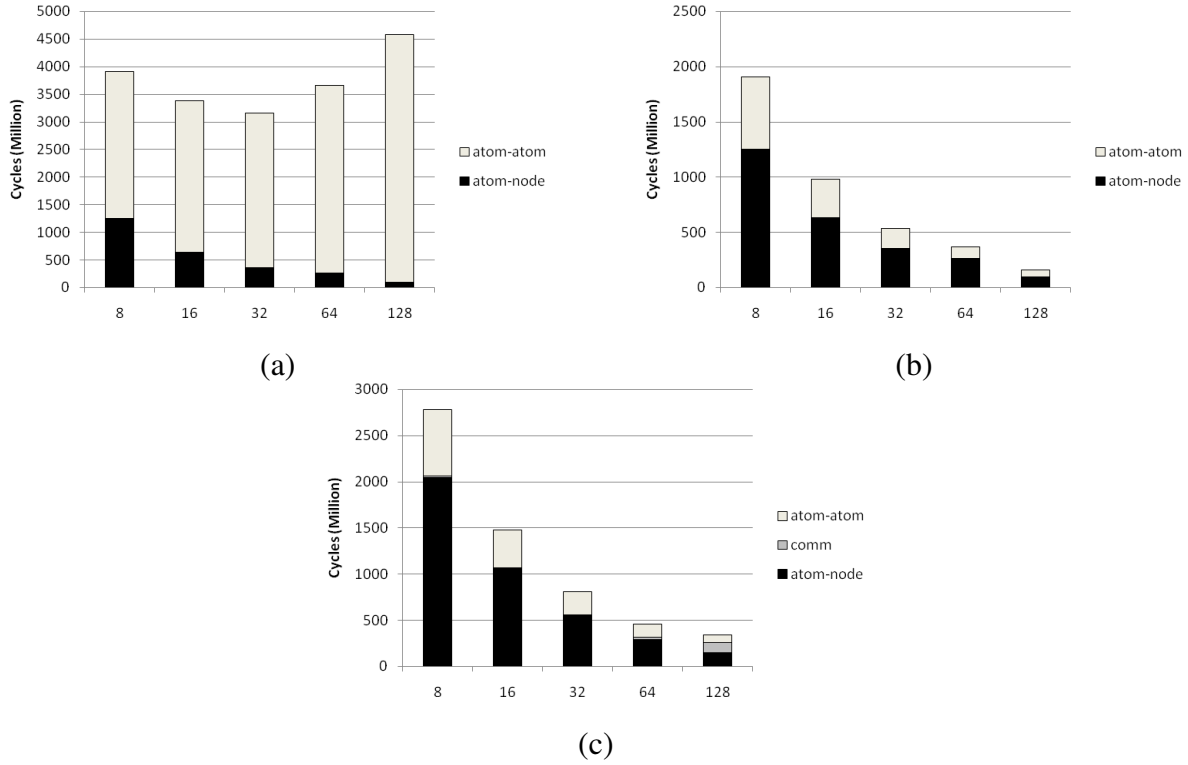
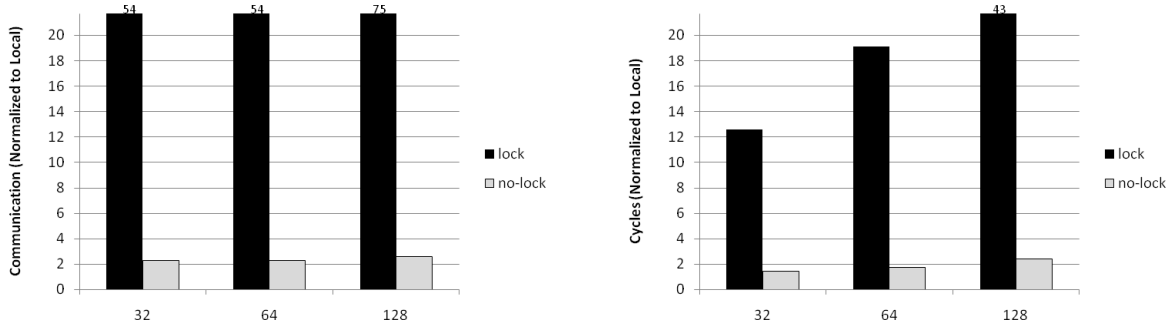


Figure 1.5 Results from runs on an SGI Altix at NASA Ames: (a) lock, (b) no-lock, (c) MPI.

Next, each processor sends requests for the indices of the atoms it requires, and then services the requests of other processors. Finally, each processor receives the data it requested. Despite the use of very high-level MPI collective communication primitives, this implementation of the gather increases the overall code size by roughly 33%.

1.4.2 Performance

Figure 1.5(a) (*lock*) and Figure 1.5(b) (*no-lock*) demonstrate the effect of the use of locks on performance and scalability. *Lock* uses locks during the atom-atom computation in the second loop, with disastrous performance consequences. *No-lock* dispenses with the symmetry optimization, and instead performs redundant computation. While performance is much improved, the scalability of the atom-atom computation is hampered by the repartitioning of data: although computation is perfectly balanced, the scaling efficiency is only 60% and shrinking as the number of processors increase.



(a) (b)

Figure 1.6 Improvement from localization, focusing on atom-atom computations, results normalized to localized version: (a) cache misses that result in communication , (b) performance.

Finally, Figure 1.5(c) demonstrates the scalability problems of the MPI implementation. At 128 processors the time to gather the interaction list dominates the time to compute the interactions.

1.4.3 Localization

Local is a localized version of the *lock* implementation, which retains the symmetry optimization but dispenses with the locks. Instead it makes use of the protocol's reduction mechanism, allowing updates to be performed on localized data.

Rather than repartitioning the atom data, the *local* implementation effects an efficient gather/scatter communication by introducing a new data structure containing only those elements of the atom data that are updated in the second loop. During the loop each processor updates its private copy of the new atom data structure. The updated data items are then globalized in a third loop (partitioned in the same manner as the first loop) which collapses them via a sum reduction and adds them to the still localized original data elements.

Figure 1.6(a) and Figure 1.6(b) demonstrate the benefit of localization as measured by communication reduction and performance improvement, focusing on the atom-atom phase of the computation. The data was obtained from runs of *lock*, *no-lock* and *local* on the simulator described in Chapter 5.

Local reduces communication of *lock* by a factor of *75* at 128 processors and improves performance by a factor of *43*, while still allowing the computation saving device. *Local* communicates

half as much data and improves performance by a factor of 2.5 over *no-lock*. Importantly, the performance gap widens as the processor count increases. In addition to reduced computation due to the symmetry optimization, *local* benefits from both the ability to overcome false ownership through the explicit placement of repartitioned data in local memory, and the efficiency of the reduction mechanism.

1.5 Contribution

The thesis of this dissertation is that:

1. User-Controllable Coherence is an efficient mechanism for enabling increased user and compiler control over data consistency and data locality.

2. The Dual Address Space provides a high-level interface for programmers to conceptualize and take advantage of the mechanism.

3. Together they provide users and compilers an efficient means, particularly appropriate to the structure of scientific applications, of keeping data consistent and local, enabling improved performance over current global address space implementations.

After a discussion of related work in the next chapter, the remainder of the dissertation evaluates the thesis in two parts. The first part demonstrates the feasibility of the ideas by describing the details of the high-level architecture, including the programming model, and its implementation via extensions to a standard directory-based cache coherence protocol. The second part evaluates the performance of Dual Address Space implementations of two applications that currently underperform on distributed address space platforms:

1. HYCOM makes use of a static though slightly irregular data decomposition, complicating the nearest-neighbor communication due to stencil computations. The use of a shared address space simplifies computation partitioning considerably, while localization significantly reduces false sharing and other artifactual communication, and preserves a communication

reducing technique through redundant computation on an extended halo of remote data otherwise not possible in a global address space implementation.

2. The Fast Multipole Method features highly irregular, dynamic computation partitioning, data decomposition and communication that significantly impair implementation in a distributed address space. The dissertation demonstrates, through indirect evaluation on existing hardware using hardware counters, the potential for the use of localization to maintain and modify data partitions, and efficiently control communication through the array reduction mechanism.

Chapter 2

Related Work

Research on Distributed Shared Memory (DSM) systems was an extremely active area in the previous decade, peaking in the late 1990's. An advantage of working in such a mature area is that one has a clear vantage point over what was successful, and what was not. However, success is not necessarily determined by merit and care must be taken not to overlook the specific merits of ideas that may have fallen short in another area or simply not had sufficient support to withstand the test of time.

At a course granularity, three approaches to research in this area have focused on different weaknesses of snoopy broadcast-based shared memory systems. Hardware-based approaches attack scalability concerns by replacing the broadcast-based snoopy cache coherence protocol with point-to-point message-based directory protocols. At the other extreme, software-based systems, generally implemented by modifying the virtual memory subsystem of an operating system, attempt to give programmers the benefits of a shared address space at a relatively low cost. Finally, hybrid systems add varying amounts of hardware in an attempt to optimize the price-performance trade-off; decreasing the cost of pure hardware controlled systems while improving on the performance of purely software-controlled systems.

The point of reference for this discussion (and the dissertation as a whole) is hardware-based CC-NUMA (Cache Coherent, Non-Uniform Memory Access), arguably the dominant system to emerge from DSM research and almost certainly the most relevant to scientific computing. CC-NUMA is currently exemplified by Altix [75] systems from Silicon Graphics Incorporated (SGI).

The Altix is the follow-on to SGI's Origin2000 [46], itself a commercialization of the prototype Dash [47] system created by researchers at Stanford University.

Several very thorough surveys of the general area already exist [17, 63]; the remainder of this chapter therefore focuses on the aspects of DSM research most relevant to User-Controllable Coherence and the Dual Address Space Architecture.

2.1 Reducing Artifactual Communication

In the introduction we distinguished between two classes of artifactual communication: communication due to false *ownership* of data and communication due to false *sharing* of data. On CC-NUMA platforms the distinction is one of granularity: ownership occurs at a page granularity while data transfer, and thus sharing, occurs at a block granularity. However, in software DSM systems, both ownership and sharing typically occur at a page granularity.

2.1.1 Addressing False Ownership

Cache-Only Memory Architectures (COMA) [82] address the issue by distributing ownership at cache-block granularity. In this architecture, there is no memory, instead memory is treated as a cache. As a consequence there is no home node for a block. The major problem for this architecture is what to do on a cache miss, that is when the desired block is not already in the local cache. Both locating the missing block, and ensuring that there is somewhere to put the replaced block if it is the last copy, require complex hardware. Another problem is determining the correct amount of memory to enable good performance.

Simple COMA [69] architectures address the complexity of COMA systems by distinguishing between coherence granularity and ownership granularity. As with a CC-NUMA architecture, data is *allocated* at a page granularity but kept coherent at a block granularity. However, unlike CC-NUMA, space is allocated in local memory for non-local data, again at a page granularity. Other blocks in the page are left invalid unless and until they are requested by the CPU. When a non-local block is evicted from the cache it can be stored in, and re-fetched from, local memory, as long as it has not been written by another processor.

Simple COMA still has a "last-copy" problem on block replacement, significantly complicating the coherence protocol. A variant from Stanford [38] addressed this problem by maintaining a fixed home for a block. However, a significant problem for all variants of the architecture is wasted memory due to fragmentation. MS-COMA [9] attempts to address *that* problem by allowing multiplexing of pages; that is non-local blocks from multiple pages are permitted to reside in the same local page.

Stache, the memory cache of the Tempest [65] architecture, preceded Simple COMA. However it requires a custom coherence protocol whereas, according to the authors, S-COMA works with a standard protocol. Reactive NUMA [24, 29] attempts to combine the best aspects of S-COMA with those of CC-NUMA platforms.

While the replication of cache blocks can improve performance for applications whose working sets for *read-only* data is larger than the last-level processor cache, it does not address false sharing.

2.1.2 Addressing False Sharing

Hardware-based cache coherence protocols are by nature eager algorithms, reacting in response to every read and write. Even so, these mechanisms are not sufficient to ensure the reasonable global ordering of instructions issued by multiple processors as formalized by *sequential consistency* [44]. The additional ordering constraints required to preserve sequential consistency are: 1) writes to the same location must be serialized, and 2) writes must be globally complete before any read returns the written value [4].

These constraints have many performance implications, not the least of which is to exacerbate the impact of false sharing. As we will discuss in Chapter 6, multiple writes by different processors to the same block, each of which must complete before the next can be issued, can result in block ownership ping-ponging from processor to processor.

Relaxed consistency models address the performance problems of sequential consistency by removing, to varying degrees, the ordering constraints imposed by the model. The most ambitious

models, Release Consistency [26] and Weak Ordering [3], relax all ordering constraints for normal memory operations, and introduce special "fence" instructions to enable explicit indication of synchronization.

While the use of these instructions allows multiple writes to the same block to be coalesced before a coherence, if the underlying coherence protocol mechanism remains unchanged, the window of opportunity for coalescing is narrow. The protocol eagerly propagates actions for write instructions regardless of their position with respect to fence instructions.

The Delayed Consistency [21] hardware protocol takes advantage of the distinction between synchronizing versus normal memory operations to delay coherence operations for normal memory operations until synchronization. However, the hardware proposed is limited in scope – the authors advocate maintaining state, and thus delaying coherence, for only between two and eight blocks at a time – so the window of opportunity for coalescing coherence transactions remains quite small.

More recent hardware-based work attempts to use speculation to reduce false sharing. For example, Coherence Decoupling [36] speculatively assumes that a block is not truly shared, and rolls back if that assumption turns out to be false. Again, the window of opportunity is determined by the amount of state that can be saved in hardware structures near the CPU. Another problem is that it can actually *increase* coherence-related traffic. Finally, it does not address the problem of true-sharing.

Since software-based distributed shared memory systems implement the coherence protocol in software, they have the advantage of perspective: coherence operations can be delayed until they are absolutely necessary. In these systems coherence is maintained at a page granularity, greatly increasing the likelihood of false sharing, so the coalescing of coherence transactions is essential to performance. Additionally, these systems can take advantage of the coarser transaction granularity to use an *update* protocol rather than the *invalidation* protocol used by most hardware implementations.

The Munin [13] software DSM system, taking full advantage of the freedom provided by the Release Consistency model, delays all coherence transactions until the end of a critical section. At the synchronization, processors send their modifications, in the form of diffs, to all sharers. The

Treadmarks system proposes a variation, known as Lazy Release Consistency [39], to the Munin protocol: rather than send diffs to all sharers at the end of a critical section, send modifications only to the processor that requires them at the beginning of the next critical section. Follow-on work by the same authors, described in the next subsection, provided mechanisms for a compiler to indicate further opportunities for reducing traffic. Home-based Lazy Release Consistency [86], simplifies the protocol substantially by supplying a per-page "home" node (as in hardware-based DSM systems), providing a single point of contact for requests for, and updates of, each shared page.

2.2 Software Control Over Coherence

A large body of work that attempted to take advantage of the structure of scientific applications by allowing software control over cache coherence protocols, centered around the Tempest interface for user-level shared memory [66]. A defining distinction between that work and User-Controllable Coherence is their respective views of the role of the coherence protocol and the consistency model. In Tempest, the coherence protocol is explicitly described and treated as a policy. The entire coherence protocol is available at the user level, and optimizations to reduce communication require modifications of the *protocol*, as well as the application, on a per-application basis [23].

A contemporaneous response to this potential embarrassment of riches, Loosely Coherent Memory [45], proposed a restricted set of mechanisms (i.e., a fixed Tempest coherence protocol), very similar in spirit to those provided by User-Controllable Coherence, that compilers could invoke to perform communication optimizations. In particular, it provided three directives which might retrospectively be viewed as moving data between local and global address spaces: *mark_modifications* created a writable block removed from the jurisdiction of the coherence protocol; *reconcile_copies* acted as a barrier, after which all modified blocks had been flushed to their home to be reconciled by a user-definable function; finally, *flush_copies* enabled a processor to return modified blocks to the home processor without the implied barrier. Perhaps the most significant difference between LCM and User-Controllable Coherence is the former's lack of a

mechanism to reconcile (i.e., "globalize") at a granularity finer than all-modified-blocks: without such a mechanism it is impossible to keep data that does not need to be communicated localized while globalizing data that requires communication.

Earlier work from the same group, Cooperative Shared Memory (CSM) [34], also proposed directives enabling software and hardware to work together to communicate data more efficiently. The emphasis of that work was on *cost*-efficiency: it proposed 1) simplified coherence hardware, leaving software to handle complex cases; and crucially, 2) *semantics-preserving* directives – inserted by users or compilers, describing producer/consumer data patterns – to help avoid the complex cases.

As noted above, the Treadmarks software DSM system also provided mechanisms that could be invoked by a compiler to improve protocol performance [22] by partially or fully removing data from the domain of the coherence protocol. In particular, it provided two directives: *Validate* was used to explicitly fetch a region of data (or if the region was indicated to be completely overwritten, *not* fetch it), thus bypassing the usual page-faulting mechanism for bringing in non-local data; *Push* enabled explicit updates of remote data at a barrier.

Other researchers have tried to circumvent the inefficiency of update protocols when combined with eager hardware-based coherence by providing mechanisms for the indication of specific opportunities for updates: the *data forwarding* mechanism [41] from Illinois, the *deliver* and *update-write* operations on the DASH architecture [47], and the *poststore* instruction on the Kendel Square Research KSR1 [67]. In the same vein, several groups of researchers have also looked at integrating into DSM systems more explicit message passing mechanisms that bypass the coherence protocol [42, 85, 25, 70].

Finally, other groups have looked into programmable coherence protocols, most notably the FLASH [43] project at Stanford; however, on that system the protocol was not accessible at user-level, only at system level.

2.3 Transactional Memory

Transactional memory [32] is currently enjoying enormous research interest. The goal of transactional memory is to simplify parallel programming by providing mechanisms, in hardware or software, for implementing implementing critical sections as *transactions*. The idea is that code within a transaction can be executed in parallel speculatively and rolled back if a conflict is detected. References to data that is truly shared and *contended* are not helped by this technique.

Speculative Lock Elision[64] *automatically* creates transactions from lock-based critical sections by speculatively assuming that locks are not necessary, executing as though the locks are not present, and rolling back if the speculation is incorrect.

The work in this area closest to User-Controllable Coherence in spirit is Transactional Coherence and Consistency [31] since it is also based on a contract between hardware and software: in this case, software partitions computation into atomic transactions and hardware ensures that they execute atomically. The drawback of this everything-is-a-transaction view is that fine-grained transactions are not an efficient means of exploiting the kind of coarse-grained parallelism exhibited by most scientific applications.

2.4 Address Spaces

Hardware-based distributed shared memory systems provide a shared virtual address space on top of a shared *physical* address space. Software-based DSM systems, on the other hand, provide a shared *virtual* address space on top of distributed physical address spaces. In both systems, a parallel program is typically implemented as a single process with multiple threads of control, all running in the same global address space.

Several Cray systems, including the T3E [72] and the more recent Blackwidow [71], implement a *distributed* virtual address space on top of a shared *physical* address space. On these platforms it is possible to directly access remote data with special load and store instructions, but not possible to cache it. As in the message passing paradigm, a parallel program is implemented as multiple *processes*, each with its own complete address space; however, each process also has the ability to

directly access data in the address space of other running processes (so long as it knows where to find it).

For completeness, we must mention the existence of a "Dual Address Space Operating System," patented by researchers at Hewlett Packard [54]. Under this operating system, an *application developer* can decide whether to run in either a shared address space or a private address space. In other words, the choice is made at a program granularity.

2.5 OpenMP Extensions

Finally, another source of related work stems from attempts in the OpenMP community to extend the standard with specific support for distributed shared memory platforms. For example, work by DEC describes HPF-like extensions to partition data and computation [10], while other work advocates runtime page migration [58]. One sub-branch of particular relevance to our work advocates privatization of memory references in software [80, 14]. That work suffers from the 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.

Chapter 3

Architecture and Programming Model

This chapter describes the Dual Address Space, the software interface to the Dual Address Space Architecture. Throughout the chapter, the discussion focuses on the high-level interface, abstracting away from the hardware implementation. This abstraction is reflected by our use of the term *threads*, as opposed to the more hardware-oriented *processors*. The discussion also abstracts away the other side of the interface, using *software* to denote the output of compilers as well as that of systems programmers and application developers.

The chapter opens with a description of the Dual Address Space. It proceeds by introducing new instructions for accessing data in the two address spaces, and moving data between them. It then describes the memory model, in particular how sequential consistency is preserved. The chapter closes with a discussion of the programming model, demonstrating how software can explicitly use the Dual Address Space to reduce artifactual communication.

3.1 Dual Address Space

A thread in a Dual Address Space Architecture has access to two address spaces: the *global* address space normally associated with threads, plus a *local* address space.

As Figure 3.1 indicates, in size and structure, the local address space of each thread is a logical duplicate of the global address space. If the global address space is addressable by 64 bits, then a thread's local address space is also 64-bit. Data in the local address space is subject to the same access controls as in the global address space. Storage set aside for data at a particular address in

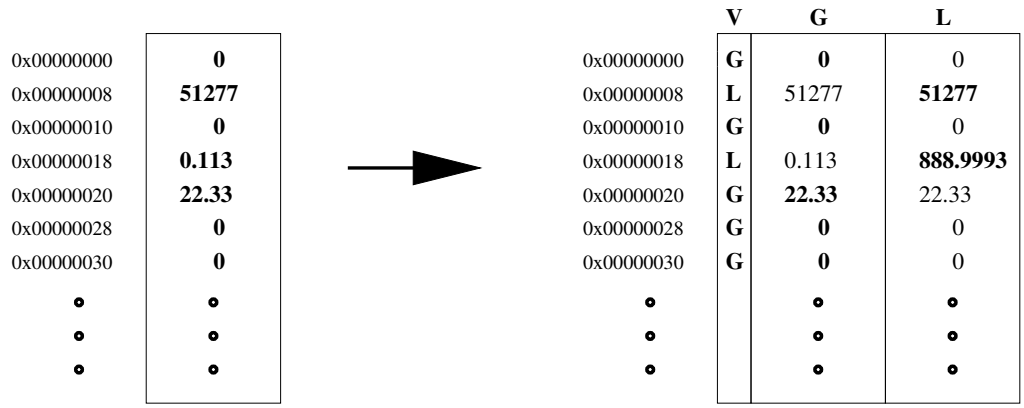


Figure 3.1 A thread's-eye view of the standard global address space (left), and of the Dual Address Space (right).

the global address space, is implicitly also set aside at that address in each thread's local address space. However, the *values* in the storage may be different.

Data Validity. At a given moment in time, data is only *valid*, at a word granularity, in either the global address space *or* the local address space of one or more threads. Data in the global address space is valid for all threads. Provided synchronization operations are marked, data in the global address space is kept sequentially consistent for data-race-free programs. On the other hand, data values in a thread's local address space are only valid for that thread. Localized data is not kept consistent and multiple local copies, with different values, may exist in parallel.

Data Movement. Threads move data between address spaces at a word granularity. Data movement is instantaneous and affects all threads. When one thread moves a word from the global address space to its local address space, the word is invalidated for all other threads until they explicitly either 1) move it into their own local address space, or 2) move it back into the global address space.

When any thread moves a word from its local address space to the global address space, the action results in the collapse of *all* the local values. If a reduction operator has been defined for the address then all the local values *and* the value from the global address space reduce to one according to the operator. Otherwise, if multiple local data values exist, the result is undefined.

3.2 New Memory Instructions

The Dual Address Space Architecture introduces two new memory instructions that enable access to the local address space, ld_L and st_L. Additionally, to facilitate data motion between the two address spaces, *localizing* semantics are added to local memory instructions, and *globalizing* semantics are added to existing, global, ld and st instructions.

3.2.1 Local Memory Instructions

Local memory instructions serve two purposes: in addition to loading and storing data in the local address space of the issuing thread, they also *localize* data – move from the global to the local address space – that is not already present.

To enable reductions, the general form for local loads is $ld_L[op]$, where op is either nop, or one of eight reduction operators. The reduction operator is associated with the *load* of data targeted for a reduction because the global value is *not* loaded; instead, since the local accumulation will ultimately accumulate onto the old global value, the thread initializes the data with the identity element particular to the operator.

The precise semantics for local instructions are as follows (where ld_L indicates that *op == nop* and $ld_L op$ that *op != nop*):

- ld_L r1 = [r2]:
    1. If the data at the address in $r2$ is not valid in the local address space of the issuing thread, then localize it: move the data from the global address space and mark the address localized.
    2. Fetch the data at the address in $r2$ from the local address space and store it in $r1$.
- $ld_L op$ r1 = [r2]:
    1. If the data at the address in $r2$ is not valid in the local address space of the issuing thread, then localize it: fill the local memory at the address with the identity elements

particular to the operator op (i.e., 0 for "+" and 1 for "*"), and mark the address localized.

2. Fetch the data at the address in $r2$ from the local address space and store it in $r1$.

- st_L [r2] = r1:

  1. If the data at the address in $r2$ is not valid in the local address space of the issuing thread, then localize it: mark the address localized.
  2. Store the data in $r1$ into the address in $r2$ in the local address space.

3.2.2 Global Memory Instructions

Global ld and st instructions also serve two purposes: in addition to loading and storing data in the global address space, they also *globalize* data – move data from the local address spaces of any threads that have the data localized to the global address space – that is not already present.

The precise semantics for global memory instructions are as follows:

- ld r1 = [r2]:

  1. If the data at the address in $r2$ is not valid in the global address space of the issuing thread, then globalize it: move the data from the local address space and mark the address globalized.
  2. Fetch the data at the address in $r2$ from the global address space and store it in $r1$.

- st [r2] = r1:

  1. If the data at the address in $r2$ is not valid in the global address space of the issuing thread, then globalize it: mark the address globalized.
  2. Store the data in $r1$ into the address in $r2$ in the global address space.

3.2.3 Update Instruction

Finally, a new $update$ memory instruction serves to effectively globalize and re-localize, in a single step, data currently valid in the executing thread's local address space.

The precise semantics are as follows:

- $update$ [r1]:

  1. After all threads that have the data in $r1$ valid in their local address space have executed the $update$, each will have the same new value, updated according to the presiding reduction operator, valid in their local address space.

Note that, to ensure forward progress, all threads that have the data localized *must* execute the update.

3.2.4 IA-64 Extension

As evidence that the instruction-set extensions we propose are not onerous, we demonstrate that the new instructions easily fit within the opcode-space of the existing IA-64 instruction-set architecture. IA-64 memory instructions contain a two-bit *hint-bit* field, yielding a maximum of four possible hints, of which only three are currently used. We use the fourth combination to indicate local instructions. Other bits, normally used to store *load/store-type* information not relevant to local operations, describe the reduction operator for loads, or indicate the *update* property for a prefetch. We have also extended Intel's Open Research Compiler for IA-64 to produce IA-64 binaries with localizing extensions.

3.3 Memory Model

We define the Dual Address Space memory model in terms of programmer-centric *weak ordering* synchronization models [3], later renamed *sequential consistency normal form* (SCNF) models in Adve's thesis [2]. Recent work has used SCNF models to define the memory models of both the C++ and Java languages [12, 50].

SCNF models define contracts between software and hardware. The idea behind the contracts is to enable faster hardware by reducing the burden of maintaining sequential consistency, while simultaneously relieving software of the need to understand the requirements imposed by those optimizations.

3.3.1 Data-Race-Free-0

The simplest synchronization model is called Data-Race-Free-0. Informally, software promises hardware that it will explicitly label all memory operations for which ordering matters as synchronization operations, so that hardware can focus on those operations and not worry about maintaining order for other operations. In turn, hardware promises that it *will* maintain order for synchronization operations.

A more formal description, based on the presentation in Adve's thesis, follows (though taking a cue from the Lazy Release Consistency authors [39], we have condensed several definitions):

Definition 3.1 Two memory operations *conflict* if they access the same location and at least one is a write [73].

Definition 3.2 *Happens-before-0* ($\xrightarrow{hb0}$):

- Given two memory operations op_1 and op_2 that are executed by the same thread, if op_1 occurs before op_2 in program order, then $op_1 \xrightarrow{hb0} op_2$.
- Given two synchronization operations op_1 and op_2 that are executed by different threads but access the same location, if op_1 completes before op_2 (i.e., op_2 reads the value written by op_1), then $op_1 \xrightarrow{hb0} op_2$.
- If $op_1 \xrightarrow{hb0} op_2$ and $op_2 \xrightarrow{hb0} op_3$, then $op_1 \xrightarrow{hb0} op_3$.

Definition 3.3 *Data-Race-Free-0*:

1. Synchronization operations are marked.
2. All conflicts are ordered by the happens-before-0 relation.

3.3.2 Localized-Data-Race-Free-0

In order to preserve sequentially consistent semantics for a data-race-free-0 program with local accesses, software must ensure that localization does not undermine the happens-before-0 relation. For example, Figure 3.2 depicts a situation in which a read *does not* return the value of the last

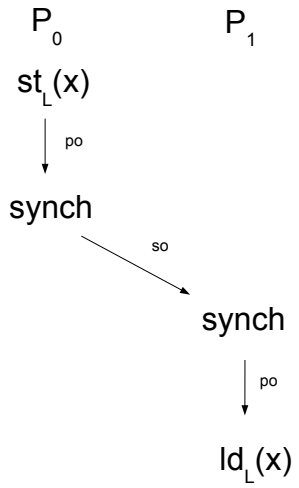


Figure 3.2 Example of a cross-thread local-to-local causal conflict.

write, even though the happens-before-0 relation *does* hold between the two operations, because both are local operations. (In the figure, *PO* labels indicate happens-before-0 ordering due to *program-order*, while the *SO* label indicates happens-before-0 ordering due to *synchronization-order*.)

Informally, to be guaranteed a sequentially consistent execution, software must ensure that no attempt is made by one thread to read data localized and written by another thread until: 1) after the two threads have synchronized, *and* 2) some thread has globalized the data.

We formally extend the definition of Data-Race-Free-0 as follows:

Definition 3.4 Two memory operations op_1 and op_2 are in *causal conflict* if op_1 is a write and op_2 is a read that returns the value written by op_1 [2].

Definition 3.5 Two memory operations op_1 and op_2 are in *local-to-local causal conflict* if op_1 is a st_L and op_2 is a ld_L that returns the value written by op_1.

Definition 3.6 A *cross-thread local-to-local causal conflict* occurs when op_1 and op_2 are executed by different threads, op_1 is a st_L, and op_2 is a ld_L that *would* return the value written by op_1 if either operation were global.

Definition 3.7 *Localized-Data-Race-Free-0*

1. Synchronization operations are marked **and global**.
2. All conflicts are ordered by the happens-before-0 relation.
3. **There are no cross-thread local-to-local causal conflicts**.

Note that software may legitimately choose *not* to obey Localized-Data-Race-Free-0. For example, looking again at Figure 3.2, it is possible that software *intends* that the load should read data from a previous local store to *x*, and not the store by the thread running on P_0, despite the synchronization event. Similar situations arise frequently in reduction computations.

However, to hold up its end of the contract hardware must ensure that when software does obey Localized-Data-Race-Free-0, it is guaranteed sequential consistency; thus if the local read by the thread on P_1 in Figure 3.2 is replaced by a global read, or by an *update* matched by *updates* from all other threads with localized copies of *x*, the value returned must be the value written by the thread on P_0.

3.3.3 Extending Other SCNF Models

Adve's thesis goes on to describe more complicated SCNF models designed to permit more complex hardware optimizations. For example, by refining the requirements for marking synchronization operations slightly, the *Data-Race-Free-1* model permits *release consistent* (RC_{SC}) hardware. As long as the synchronization operations remain global, the extension of Data-Race-Free-1 to allow local operations is trivial.

Two more synchronization models described in Adve's thesis, *PLpc1* and *PLpc2*, represent stages in the realization of a model that supports more complex release consistent (RC_{PC}) hardware.

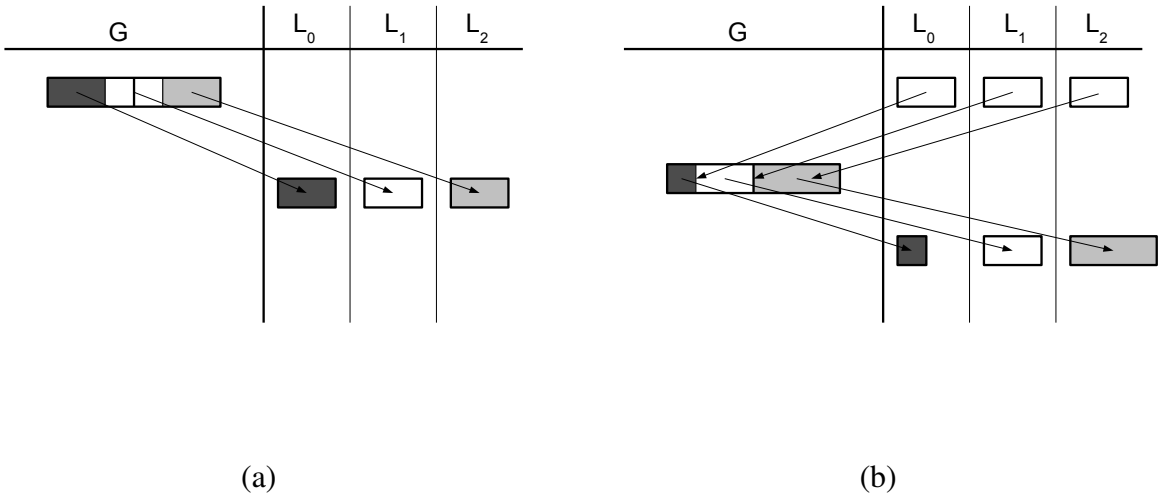


Figure 3.3 Using a Dual Address Space to: (a) create a data partition, (b) repartition data.

Again, PLpc1 requires *further* distinguishing between different varieties of synchronization operations. PLpc2, on the other hand, requires software to distinguish *data* operations that must occur atomically from those that can be executed non-atomically. A non-atomic write is visible on one processor before it is visible on all processors. A non-atomic read returns the value of a write before it is visible on all other processors. Since *all* local reads and writes are by definition non-atomic, they may be labeled as such.

Thus, so long as software marks synchronization operations according to the SCNF model requirements, *and* avoids cross-thread local-to-local causal conflicts, it will be guaranteed sequentially consistent execution of a PLpc2 program *with local memory operations* on release consistent (RC_{PC}) hardware.

3.4 Programming Model

Software uses the Dual Address Space to avoid artifactual communication by 1) explicitly placing data near threads that use it, and 2) explicitly delaying communication until it is required.

3.4.1 Data Partitioning

Defining a data partitioning using a Dual Address Space is similar to using the existing *first-touch* policy on the SGI Altix. Under first-touch the first thread to touch a piece of data owns that data; more precisely stated, the thread owns *page* that the data resides in.

The benefit of the first-touch policy over more static policies, for example round-robin at a page or block granularity, or even the programmer-controlled distribution offered by languages like High Performance Fortran, is that it allows the data distribution to be dynamically determined. Thus the data partition can follow the computation partition, and need not be determined or even expressible at compile time.

However, the policy has two obvious drawbacks. First, the granularity of ownership is determined by an architectural parameter, the page size, not by the needs of the application. As described in the introduction, this can lead to significant performance degradation due to false-ownership, particularly for computations that require multi-dimensional data distributions. Second, since only the first touch matters, a data distribution is static after it has been initially determined. There is no mechanism for modifying the distribution once it has been defined.

The following shows how local operations enable the data partition to follow the computation partition at a much finer granularity, since data can be localized at a word granularity. It then goes on to describe how a data partition thus defined can be modified.

Partitioning Data. To create a data partition, as shown in Figure 3.3(a):

1. Owners localize their portion of the data and continue to operate on it using local operations.

The example supposes that computation partitioning divides loops that operate on the pictured array into three equal parts (three colors), but because the data straddles a single page boundary (solid line), the Altix first-touch policy divides ownership into two parts. Proper data distribution is achieved by using local memory operations for the duration of the computation rather than global operations, as follows (where a st_L operating on a 4-byte value translates to the $st4.loc$ assembler mnemonic):

```
loop:                                          loop:
        st4     [r2]=r3,4                              st4.loc [r2]=r3,4
        ...                          =>                ...
        add     r3=1,r3                                add     r3=1,r3
        br.cloop.sptk   loop                           br.cloop.sptk   loop
```

Note that the lifetime of the distribution is not limited to the loop shown; rather it is determined by the next use of globalizing instructions.

Repartitioning Data. Figure 3.3(b) illustrates the steps required to modify an existing data partition:

1. Threads (current owners, new owners or others) globalize data to be repartitioned.
2. New owners localize their portion of the data and continue to operate on it using local operations.

This example continues the previous example, further supposing that computational requirements dictate a redistribution of the array from three equal parts to three unequal parts.

```
loop:
        st4.loc [r2]=r3,4
        ...
        add     r3=1,r3
        br.cloop.sptk   loop

        ... ;; set up for loop2
loop2:
        lfetch  [r2],4
        br.cloop.sptk   loop2

        ... ;; set up for loop3
loop3:
        st4.loc [r2]=r3,4
        ...
        add     r3=1,r3
        br.cloop.sptk   loop3
```

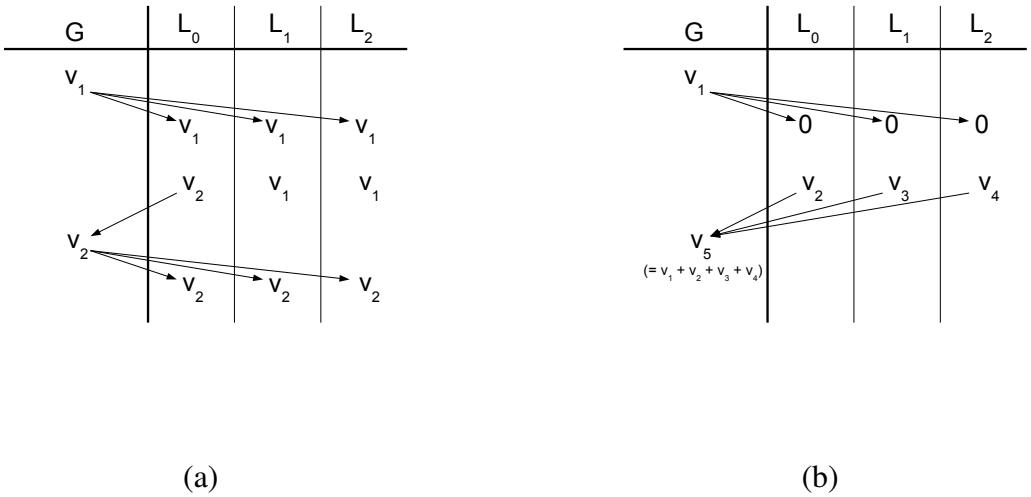


Figure 3.4 Using a Dual Address Space to implement communication patterns: (a) producer/consumer, (b) reduction.

The redistribution is achieved by having each thread globalize its data (in loop2, using global prefetch instructions), then relocalizing according to the new computation partition in loop3, and continuing to use local memory operations for the duration of the computation.

3.4.2 Communication

Since localized data is not kept consistent, modifying (already) localized data does not cause communication. Since globalizing localized data causes it to become consistent, and communication is required for consistency, communication is made explicit by moving data into the global address space. Thus, to explicitly delay communication, software localizes data and leaves it localized for as long as possible, moving it to global memory only when required to satisfy data-dependencies.

The following describes how the appropriate use of local and global operations can ensure that common communication patterns are implemented efficiently by making their communication explicit.

Producer/Consumer. Figure 3.4(a) demonstrates the steps required to explicitly implement producer/consumer communication of a single word:

1. Each thread, producer and consumer(s), localizes the word and continues to operate on it with local operations.

2. Threads synchronize.

3. One thread, producer or consumer, globalizes the word.

Note that either producer or consumer may globalize the data, and that the other may continue to use local operations. However, if the data is being globalized only to be immediately localized again by all threads, a more efficient implementation would use the explicit update mechanism to communicate the updated word from producer to consumers:

1. Each thread, producer and consumer(s), localizes the word and continues to operate on it with local operations.

2. All participating threads, producers and consumers, execute the explicit *update* instruction.

Note that in addition to removing the globalization, the explicit update approach may also remove the need for synchronization. Synchronization is implicit since all producing and consuming threads participate in the update. However, in general, synchronization may still be necessary to ensure other global values are consistent.

For an example of producer/consumer communication see the false sharing model discussion in Chapter 1.

Reductions and Gather/Scatter. Figure 3.4(b) illustrates the steps for implementing the reduction communication for a single word:

1. Writers localize the word using *operator*-load and continue to operate on it with local operations.

2. Threads synchronize.

3. Some thread globalizes the word, causing a reduction.

Again, an alternate implementation might leave the data localized and ensure that all participating threads, writers and any readers, execute the update instruction. However, it must be noted that after doing so, all threads are left with the new *global* value, albeit localized; the operator property is lost, and the data is no longer fit for future reduction.

To implement a more general gather/scatter communication for non-local writes to an array of data:

1. Writers localize the portion of the array that they will update, using *operator*-loads and compute using local operations.

2. Threads synchronize.

3. Threads, owners or otherwise, globalize the array, causing multiple reductions.

For an example of gather/scatter communication, see the localized implementation of AMMP in Chapter 1.

3.4.3 Higher-level Interface

Currently the only high-level interface for distinguishing between local and global memory operations is a simple pragma-based mechanism for C/C++ programs, and its directive-based equivalent for Fortran programs. The syntax for the *localize* pragma is as follows:

#pragma localize ([*optional-operator*,] *name-list*)

The pragma indicates to a complying compiler that, for the duration of the following statement (potentially a statement block containing multiple statements), operations accessing data in *name-list* should be local operations. If the *optional-operator* parameter is present, then loads should be implemented as *operator*-loads.

A more elegant, high-level language approach to expressing the location and motion of data in the Dual Address Space is an open research question.

3.5 Summary

This chapter described the Dual Address Space, the programmer's interface to the Dual Address Space Architecture, beginning with a thread's-eye view of the Dual Address Space. In size and structure, the local address space is a logical duplicate of the global address space, but it may contain different data values. Data is only valid in either the local or global address space at a given time. The chapter then introduced new instructions for accessing data in the local address space, ld_L and st_L, and presented the precise semantics of the local and global instructions.

Next, the chapter provided a formal definition of the memory model, in terms of well-established programmer-centric synchronization models. It demonstrated that as long as software follows established guidelines (indicating synchronization operations and making them global; ensuring no races exist between data operations; ensuring localized data is globalized appropriately) it will be guaranteed sequentially consistent execution. Finally, the chapter described the programming model, explaining how software uses the Dual Address Space to explicitly reduce artifactual communication.

Chapter 4

Implementation

This chapter describes an implementation of the Dual Address Space Architecture. The implementation extends the SGI Altix design, itself based on the SGI Origin [46], with roots in the Stanford Dash [47]. The chapter opens by describing extensions to the design of a *node*, the basic building block in all the aforementioned systems.

The chapter continues with a detailed description of the User-Controllable Coherence [51] cache coherence protocol, initially assuming single-word cache blocks, but then considering the ramifications of multi-word blocks. Next the chapter describes the support for the memory consistency model presented in the previous chapter. Finally, the chapter closes with a discussion of the implementation of local memory.

4.1 Node Design

Figure 4.1 presents a block diagram of a *node*, the basic building block in a Dual Address Space implementation. The design is based on the node of an SGI Altix system, though we also refer to its predecessor the Origin 2000, about which more detailed information is publicly available [46].

The Hub chip, renamed the Super-Hub (SHub) in the Altix, is literally at the center of the design. It connects one or more processors with 1) memory and its associated directory, 2) the network and 3) I/O. The Hub hides the DSM implementation from the processor by handling all memory requests, determining whether requested blocks are available locally or require communication, performing any communication necessary to obtain requested blocks, and finally supplying the blocks to the processor.

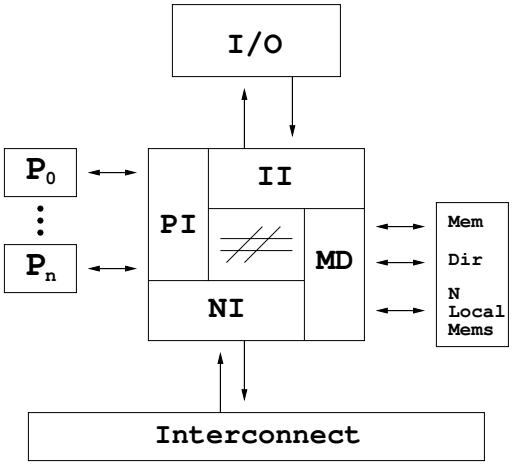


Figure 4.1 Block diagram of a node in a Dual Address Space implementation, based on that of the SGI Altix and its predecessor, the Origin 2000.

In current Altix implementations, the SHub is a separate ASIC. As core counts increase with chip density, we would expect it to become more integrated with the processor as in current AMD Opteron designs [15].

4.1.1 Processor Interface

The primary requirement on the processor, in addition to the instruction set and coherence protocol extensions required to support a Dual Address Space, is that it supports a relaxed consistency model. We assume an Itanium-based processor since 1) the current Altix is Itanium-based, 2) Itanium uses a release consistent memory model, and, 3) as we have demonstrated in Chapter 3, the IA64 instruction set can easily be extended to support localizing instructions.

Each processor is shown with a point-to-point connection to the Hub rather than connected via a single common bus. The distinction is purposeful and significant, since local memory requests should not be subject to a bus-based protocol. However, the implementation could be realized as a multiplexed single bus, as in the Origin implementation [46].

Hardware to perform reductions, inspired by the Tree Module in IBM's Blue Gene/L system [1], is located in the Processor Interface. As the result of the reduction transaction, the requesting processor receives *multiple* responses, one from memory, and one from all processors

that have updates. The requesting processor's interface is responsible for combining results and delivering them to the processor.

4.1.2 Memory and Directory Interface

The directory and memory interface is extended with connections to local memory. The implementation of local memory is discussed in detail in Section 4.5.

4.1.3 Network and I/O Interfaces

The network and I/O components of the interface remain unmodified.

4.2 Cache Coherence Protocol

This section describes the implementation of the User-Controllable Coherence protocol. For ease of presentation we divide the description of the protocol into two parts. The first part describes the processor side of the protocol, abstracting away the mechanism by which coherence events are propagated between processors. The second part focuses on that mechanism, describing extensions to a directory-based protocol.

4.2.1 Abstract Protocol

The traditional role of a cache-coherence protocol is to help maintain a global address space in the presence of caching – that is, to ensure that one processor's modifications to global memory are visible to all processors. State machines for every block in every processor cache, maintain the following invariant: *multiple processors can read a cached block at the same time, but only a single processor can write a cached block at a time.* This system of state machines preserves the invariant by producing for each input – a read or write operation on a block by some processor – the *coherence events*, if any, necessary to maintain the invariant for that block.

Figure 4.2 presents the state machine for a standard *MSI* cache coherence protocol. It consists of three states, representing the state of a line in the cache – *Modified*, *Shared*, or *Invalid*. *Modified* indicates that the block has been, or will be, written and is held exclusively in that cache. *Shared*

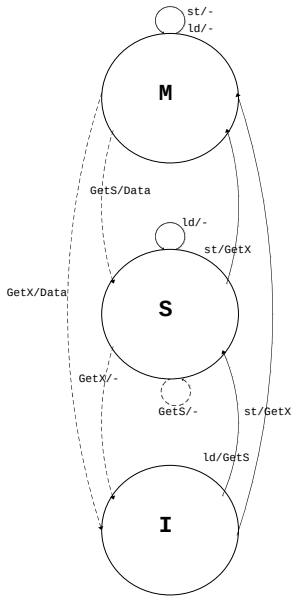


Figure 4.2 Basic MSI protocol state diagram (based on Culler and Singh [17]).

indicates that the data is read-only and may be found in other caches. *Invalid* indicates that the block is either not in the cache or is present but has been invalidated.

There are two kinds of transitions between states. Solid lines indicate *internal* transitions, transitions caused by the execution of load and store instructions. Labels on the internal transitions indicate the instruction that caused the transition and the resulting coherence events generated. For instance, a store to a shared block results in a transition to state modified, in addition to a $GetX$ coherence event.

Dashed lines indicate *external* transitions, transitions caused by coherence events generated by other processors. Labels on the externally caused transitions indicate the coherence event that caused the transition and whether or not a data response is required.

Extended Protocol. In addition to maintaining the global address space, the User-Controllable Coherence protocol also maintains a local address space for every processor, *and* facilitates the transitions between the two address spaces. There are two significant ways in which supporting a local address space is different from supporting a global address space. First, uncached localized

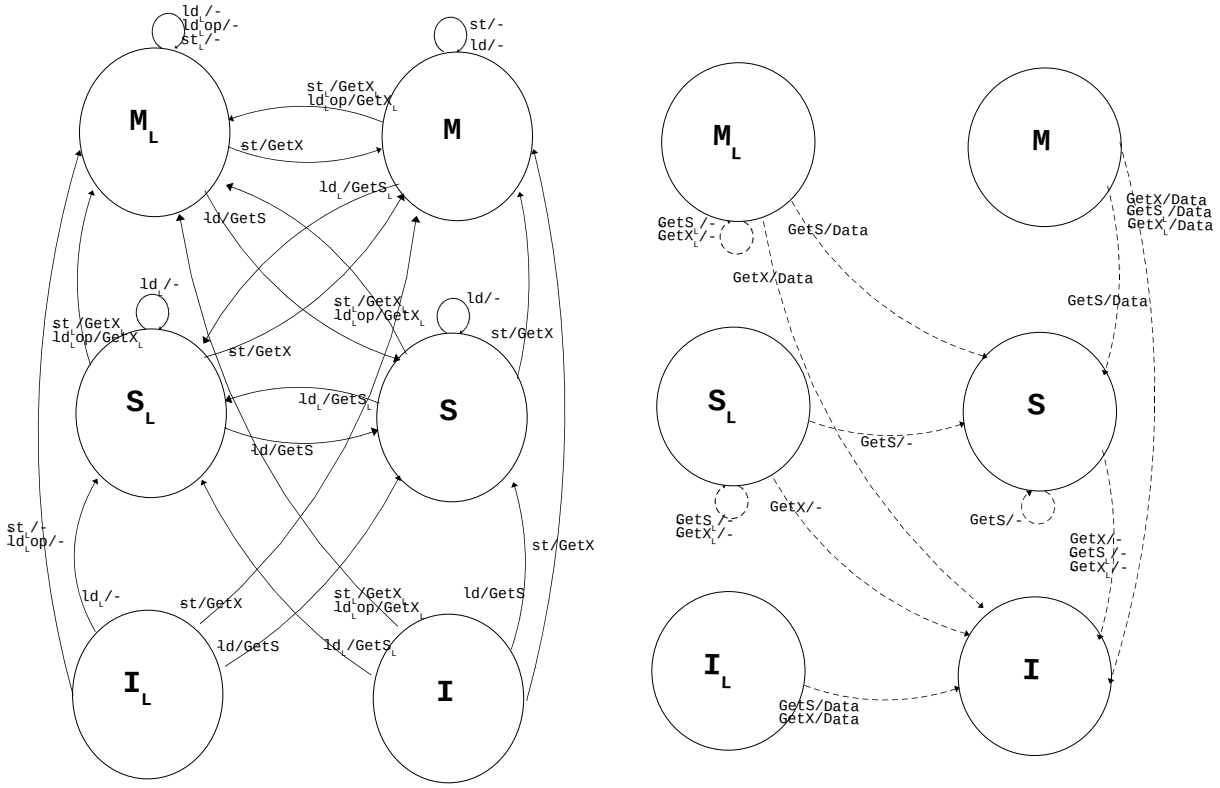


Figure 4.3 Extensions to the MSI protocol to enable transparent use of local memory.

blocks reside in *local* memory: localized blocks purged from the cache reside in local memory, and local memory must be searched first for blocks not found in cache. Second, localized blocks can be both read *and* written by multiple processors, requiring a mechanism that resolves both accumulations to the same address and multiple writes to different parts of the same block.

Figure 4.3 illustrates changes to the protocol required to support a Dual Address Space. For clarity, the state diagram is broken into two parts with internal transitions on the left (with solid lines), and external transitions on the right (with dashed lines). Inspection of the two halves reveals that the basic MSI protocol described above remains intact; the transitions between the M, S and I states are exactly the same.

The protocol adds three new states – *Local Modified* (M_L), *Local Shared* (S_L), and *Local Invalid* (I_L). Like its counterpart in the standard protocol, M_L indicates that the block is localized

and has been, or will be written; however, it does *not* imply exclusive access to the block. Blocks in S_L state are localized read only, and therefore do not require communication upon globalization. I_L represents localized data that has been evicted from the cache and moved to local memory.

Note that there is an ambiguity between I and I_L since both represent data that is not cached. Processors determine whether to look for data in local or global memory based on the type of the memory instruction that caused the miss. If the instruction was local load or store, the issuing processor looks for the data in local memory first. If the data is not found, then the request is forwarded to global memory. Similarly, if the instruction that caused the miss was a global load or store, then the processor goes straight to global memory, though it is possible the processor has the data in its local memory. The coherence mechanism must be able to make the distinction and inform the processor.

Internal transitions between the new set of states closely mirror the transitions between the old states. Internal transitions between global states and local states and back, which reflect *localization* and *globalization* events respectively, are also symmetric.

External transitions, on the other hand, are not symmetric since no coherence events result in a transition to a new local state. Two new coherence events, $GetS_L$ and $GetX_L$, indicate that some other processor has localized a block. Processors that have the global data cached must invalidate it upon receiving a $GetS_L$ or $GetX_L$ to ensure they do not find data if they issue a normal load or store to the address. However, processors that already have the block localized are not interrupted.

At a high level, a typical localization/globalization cycle for a block proceeds as follows:

1. Processors execute global operations on the block, resulting in the global coherence events necessary to maintain the invariant.

2. Some processor executes a localing memory operation on the block, resulting in a localizing coherence event, which invalidates the block in all other processor caches.

3. The processor, and potentially others, continues to execute local operations. While for the most part these operations do not generate coherence events, the coherence mechanism continues to monitor the address, waiting for a globalizing event.

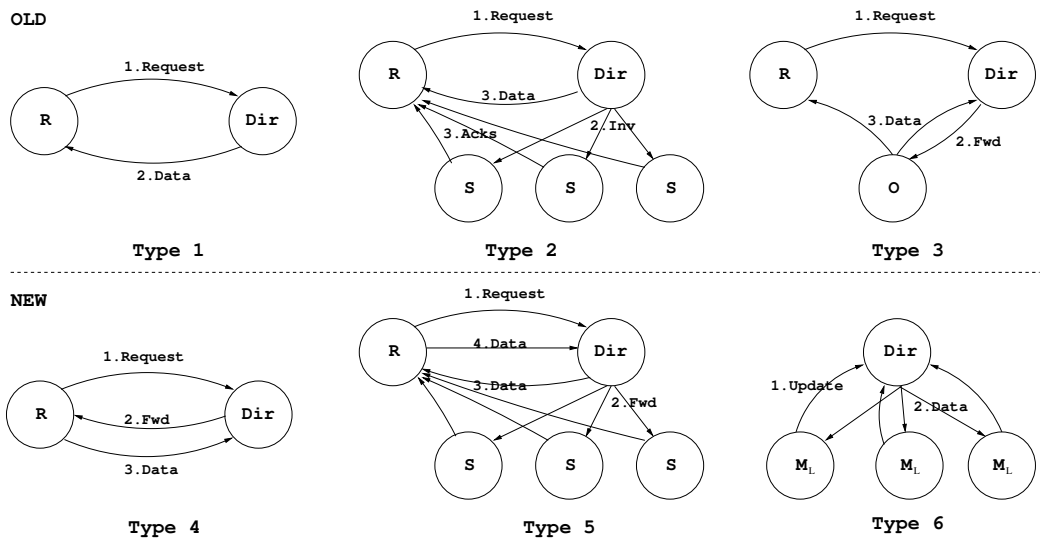


Figure 4.4 Pictorial descriptions of the complete set of transactions necessary to maintain coherence in the four protocol phases described in the text. Types 1, 2 and 3 may be familiar as the transactions in a standard protocol, while types 4, 5 and 6 are new.

4. Some processor executes a globalizing memory operation, resulting in a globalizing coherence event which potentially leads to a reduction if it has been modified.

4.2.2 Directory Protocol

In a directory protocol, the coherence mechanism is enhanced with state machines for every block in memory, collectively called the *directory*. The directory in conjunction with the processors maintains the coherence invariant for each block via sequences of messages, collectively referred to as *transactions*. Figure 4.4 presents the full arsenal of transactions necessary to maintain coherence in a Dual Address Space. The first three, types 1, 2 and 3, may be familiar as transactions from a standard protocol, while types 4, 5 and 6 are new.

A state machine at the directory in a standard MSI protocol has three states: *Modified* means a processor has the block modified. The directory keeps track of which processor. *Shared* means at least one, possibly more than one processor, has the block read-only. Again, the directory keeps track of the processors in its *sharing list*. Finally, *Idle* means that no processor has the block cached. Directory state machines are extended with two new states, M_L and S_L.

The high-level localization/globalization cycle for a block in a directory-based implementation proceeds as follows:

1. *Global.* A standard directory-based MSI protocol, very similar to that presented in [61], maintains global coherence. No transaction is necessary when a processor executes a load and finds the block already cached. Similarly, if a processor has already modified a block in its cache, no transaction is necessary to perform further reads or writes.

   When a transaction *is* necessary, its type is determined by two factors: whether the instruction modifies the block, and whether the block is already modified in another cache. Simple request-response transactions (Type 1 in Figure 4.4) are possible when data has not already been, and will not be, written. Type 2 transactions are required when data is going to be written: other processors that have cached the data must be informed, through an invalidation message, so that they do not read stale data. Finally, Type 3 transactions are necessary when data that has been written is going to be read or written by another processor: the directory forwards the request to the writer, which replies to the requester with the data.

2. *Localizing.* The first dynamic instance of a *localizing* load or store to a block results in a message to the directory for that block, which sends invalidation messages to any current sharers, adds the requesting processor to the sharing list, and forwards the block.

3. *Local.* Most subsequent *localizing* loads and stores to the block do not result in any further coherence transactions. Two exceptions: 1) When a read-only block is written, resulting in a transaction to inform the directory that a reduction may be necessary. 2) When owners of a localized block perform an update transaction.

4. *Globalizing.* The first instance of a *globalizing* load or store instruction results in the appropriate re-globalizing actions at the directory. Specifically if the state of the block is Local-Shared the directory ensures that sharers are downgraded. On the other hand if it is in Local-Modified state the directory must ensure that all writers – noted on the sharing list – send their contributions to the requesting processor, and that the requesting processor knows the reduction operator and how many messages to expect.

4.2.2.1 Transaction Details

The following describes the details of the directory protocol. Actions taken by the directory depend on the type of message received from a requesting processor, and the state of the block at the directory.

$GETS$**:** Equivalent to standard protocol if block is in a global (or idle) state at the directory. Otherwise, this is a globalizing transaction.

Global

- *Idle* or *Shared*: Type 1 transaction: directory adds requester to the sharing list for the block and sends requester the data.
- *Modified*: Type 3 transaction: directory adds requester to the sharing list for the block, and forwards the request to the current owner. Owner downgrades its copy of the block to *shared* and forwards the data to the requester *and* to the directory. The directory needs a valid copy of the block to satisfy future requests for the block.

Globalizing

- *Local Shared*: Type 2 transaction: directory adds requester to sharing list, sends requester the data, and sends sharers a *downgrade* message: they may keep the data, but they should move it from state S_L to state S. Sharers send acks to requester to complete the transaction.
- *Local Modified*: Type 5 (reduction) transaction: directory removes sharers from sharing list, and adds requester. Directory sends data to requester (along with reduction operator and operand type and size) and sends a *reduction* message to sharers. In response, each sharer sends its copy of the block to requester, which performs a reduction according to the algorithm presented in Section 4.3.2.

$GETX$**:** Again, equivalent to standard protocol if block is in a global (or idle) state at the directory. Otherwise, this is a globalizing transaction.

Global

- *Idle* Type 1 transaction: directory adds requester to the sharing list for the block and sends requester the data.

- *Shared*: Type 2 transaction: directory adds requester to sharing list, sends requester the data, and sends sharers an *invalidate* message. Sharers invalidate their data and send acks to requester to complete the transaction.

- *Modified*: Type 3 transaction: directory removes current owner from, and adds requester to, the sharing list for the block, and forwards the request to the owner. Owner invalidates the block and forwards the data to the requester *and* to the directory. The directory needs a valid copy of the block to satisfy future requests for the block.

Globalizing

- *Local Shared*: Type 2 transaction: directory adds requester to sharing list, sends requester the data, and sends sharers a *local-invalidate* message. The message type is distinct from a normal *invalidate* in order to remove potential ambiguity in the event of a race, as described in Section 4.2.2.2. Sharers send acks to requester to complete the transaction.

- *Local Modified*: Type 5 (reduction) transaction: directory removes sharers from sharing list, and adds requester. Directory sends data to requester (along with reduction operator and operand type and size) and sends *reduction* message to sharers. In response, each sharer sends its copy of the block to requester, which performs a reduction according to the algorithm presented in Section 4.3.2.

$GETS_L$**:** If this is a localizing request, the actions taken by the directory are very similar to those for a $GETX$ in a standard protocol, since sharers must be invalidated.

Localizing

- *Idle*: Type 1 transaction: directory adds requester to the sharing list for the block and sends requester the data.

- *Shared*: Type 2 transaction: directory sends invalidations to all sharers (since subsequent global references must *re-globalize* the block), removes them from the sharing list, adds requester to the new sharing list, and sends requester the data. This transaction is very similar to that associated with a read-exclusive request when there are sharers in a standard directory protocol. The difference is that the requester is not made owner since other processors may also now request localized versions of the block.

- *Modified*: Type 3 (or 4) transaction: directory forwards the request to the current owner, which subsequently invalidates its copy of the block and forwards the data to the requester *and* to the directory. The directory needs a valid copy of the block in case the requester, or some other processor, writes a portion of it while the address is localized.

  A Type 4 transaction results when the requester is also the owner. While it might seem possible to simplify the transaction by sending the data with the initial request, as we will explain in some detail in Section 4.2.2.2, doing so would lead to an explosion in the number of protocol states. In the interest of keeping the number of states to a minimum, the requester sends a simple request and awaits the forward of its own request.

<u>*Local*</u>

- *Local Shared* or *Local Modified*: Type 1 transaction: directory adds requester to the sharing list for the block and sends requester the data.

$GETX_L$**:** The actions are nearly identical to those for $GETS_L$. The primary difference is that the reduction operator (possibly nop) and the type and size of the operands are included in the request message and stored at the directory. If the reduction operator is not nop, the directory simply acks the requesting processor, since the requester will initialize its local copy of the block with values appropriate to the reduction operator; for instance, in the case of a sum reduction, the block is initialized to zero.

$UPDATE$**:** The block *must* be in state *Local Modified*. Furthermore, every processor on the sharing list must send its copy of the block.

<u>*Local*</u>

- *Local Modified*: Type 6 transaction: after a message has been received from each processor on the sharing list, directory reduces the block according to the stored reduction operator and operand type and size. Directory then sends the new block to each processor on the sharing list.

4.2.2.2 Transient States

To this point we have conveniently ignored race conditions. Race conditions arise in a request/response protocol when there are requests for blocks for which requests are already outstanding. Transient states – intermediate states in which a request has been made, but a response has not yet been received – allow a protocol to explicitly handle race conditions, since intermediate states can be backed out of if necessary.

However, because extra states complicate both the verification of a protocol design and testing once it is implemented [17], it is important to keep the number of transient states to a minimum. We have taken great pains to limit the total number of states in our protocol to less than twice the total number in the original protocol.

We have already noted one instance where given a choice between adding transient states or adding latency to a transaction, we decided to avoid adding states. Recall that in the situation where a processor had a block modified and was making a localizing request, we observed that sending the data to the directory at that point would result in a protocol state explosion.

The problem with this situation is the potential for a race if some other processor makes a request for the block. That potential requires the use of a transient state to enable the localizing cache to back out of the localization if it loses the race. Since there are two potential destination states, S_L and M_L, two transients would be required, call them MS_L and MM_L. In addition, each of those two states requires two more transient states to handle the situation when the localizing processor loses the race. In that event, it must await two messages from the directory in any order: a busy-ack acknowledging (and refusing) the original request, and the forwarded request from the

other processor. The odd-looking transaction type 4 in Figure 4.4, then, is the result of avoiding the addition of six new protocol states.

Another example arises from a similar situation going in the other direction, globalizing a localized block. In this case a processor has a block localized, in either state S_L or M_L, and issues a global operation. Again, the potential for race conditions requires the use of transient states. Rather than adding four new transients – i.e., S_LS, S_LM, M_LS, M_LM – we have chosen to reuse existing transients from the global protocol: IS and IM. Fortunately, most requests are not ambiguous since they can be divided between those that are only applicable to localized data and those that are only applicable to global data. Other saved state serves to disambiguate the few remaining requests.

4.3 Block-based Storage

The definition of a Dual Address Space states that data is localized and globalized at a word granularity. However, standard cache coherence protocols operate at a multi-word, or *block*, granularity; multi-word blocks have the potential to impact both performance and correctness of the User-Controllable Coherence protocol.

Beginning with performance impact, global operations on data co-located with localized data can result in significant performance degradation. Just as each global operation globalizes the block, each subsequent local operation re-localize the block. To avoid such ping-ponging between address spaces, software will want to avoid to the extent possible co-locating data that will be localized with data that will not. This involves optimizations already standard practice on shared-memory platforms, such as aligning data on cache-line boundaries and padding data structures to ensure they fill an entire block.

More troublesome are two potential correctness problems:

- Writes to different parts of the same localized block by multiple processors must be resolved correctly when the block is globalized.

```
 1  wrk-blk = blk-in-my-cache;
 2
 3  foreach incoming block blk
 4      if OP == NOP then
 5          if is-from-dir(blk) then
 6              dir-blk = blk;
 7          else if is-invalid(wrk-blk) then
 8              wrk-blk = blk;
 9          else
10              xor-blk = xor-blk | (wrk-blk ^ blk);
11          endif
12
13          if is-last-blk(blk) then
14              if is-zero(xor-blk) != true then
15                  wrk-blk = xor-blk ^ dir-blk;
16              endif
17          endif
18      endif
19
20      else if OP == ADD_INT_1 then
21          foreach byte in blk and wrk-blk
22              wrk-blk += blk;
23          end
24      endif
25
26      else if OP == ADD_INT_2 then
27          foreach double-byte in blk and wrk-blk
28              wrk-blk += blk;
29          end
30      endif
31
32      [etc...]
33
34  end
```

Figure 4.5 Reduction algorithm.

- Depending on how operator reductions are implemented, mixing of reduction data with data not involved in the reduction could lead to incorrect results.

Both potential problems are handled through the reduction mechanism, invoked when a localized block that has been modified by more than one processor is globalized.

The cache on the receiving end of a reduction transaction, the type 5 transaction in Figure 4.4, executes the algorithm listed in Figure 4.5. Inputs to the algorithm come from three sources: 1) the processor's own cache or local memory (*blk-in-my-cache*), 2) the directory, which has the original

block (*dir-blk*), and 3) other processors that have modified the block. The action taken by the cache depends on the reduction operator, as well as the type and size of the operands, information carried as part of the data message in which the block is sent. No matter what the operator, the overall goal is to take the contribution from each processor and accumulate it onto the block that had been stored at the directory.

4.3.1 Multiple Local Writers

When the operator is nop, multiple processors may have written different words in the block. The code in lines 4-18 merges the writes into a single block. In common with several page-based software DSM implementations that handle multiple writers [70, 16], the code exploits the fact that $((A \text{ xor } B) \text{ xor } A) == B$.

Since it is a violation of Localized-Data-Race-Free-0 for multiple processors to write at the same address of a localized block, we can assume that if two versions of block B – call them B_1 and B_2 – are both modified, the modifications do not overlap. Further, we know that in locations *not* modified, each is equivalent to the original block A stored at the directory. Therefore $(B_1 \text{ xor } B_2)$ is equivalent to $(B \text{ xor } A)$ – where B contains the modifications of both B_1 and B_2 – *and* B can be recovered by *XOR*'ing the result of $(B_1 \text{ xor } B_2)$ with the directory entry A. The algorithm works by accumulating a series of these $(B_1 \text{ xor } B_2)$ computations in *xor-blk* and then recovering B through a final *XOR* with *dir-blk*.

4.3.2 Operator Reductions

Assuming a single reduction operator per block, and no intermixing of reduction data with non-reduction data, the implementation of the accumulation for other operators is straightforward. The only complication is ensuring that correctly sized operands are used in the computations. For example, byte-sized operands are formed in line 21 of the algorithm, while double-byte-sized operands are formed at line 27.

We have identified three levels of support for the intermixing of reduction data with data not involved:

- *Do not allow intermixing of reduction data with non-reduction data.* Under this approach it is software's responsibility to ensure that reduction data is not co-located in a block with data that is not involved. The current implementation takes this approach; thus, ld_Lop initializes the entire block and a globalizing reduction operates on the entire block.

- *Allow intermixing of reduction data for one operator with non-reduction data.* The addition of per-word bits to the per-block state would enable an arbitrary association between words and a single operator: ld_Lop would initialize only the affected word and set the per-word bit. A globalizing reduction would operate only on words with the bit set.

- *Full generality: allow multiple reduction operators per block.* A straightforward, but costly, implementation might provide storage for an operator per word. A more realistic implementation might instead permit multiple reduction operators, but with a severe performance penalty. Under this approach, a ld_Lop on a block already localized with a *different* operator would result in a reduction of words linked with the current operator, enabling the block to be re-localized with the new operator. Future operations on words linked with the old operator would again require a reduction to reinstate the old operator.

4.3.2.1 Atomic Updates

A subtle impact of the *multiple-operators-with-reduced-performance* approach is its requirement for an atomic update mechanism. Consider the following scenario: P_0 has executed a ld_Lop, executed another instruction to update the value in register, and is about to st_L the value. However, before it executes the store, the block is globalized and re-localized because P_1 has executed a different reduction operator on another word in the block. When P_0 executes the store, it does so in an unexpected context; rather than writing a value that will be reduced along with local values from other processors, it is potentially creating what could become the new, incorrect, *global* value.

One way to provide atomic updates would be to introduce a localized compare-and-swap instruction (called *cmpxchg* in the IA-64 instruction set), and mandate that software use the instruction when performing updates to words involved in reductions. In this scenario, an update sequence

might look as follows (where we have taken significant liberties with IA-64 instruction syntax for ease of presentation):

```
                                  label:
ld4.loc.+  r1=[x]                    ld4.loc.+       r1=[x]
add        r2=r1,r2    =>            add             r2=r1,r2
st4.loc    [x]=r2                    cmpxchg4.loc    r3=[x],r2,r1
                                     if (r3 != r1) goto label
```

The *cmpxchg* only succeeds if the value at x has not been modified since the load; if it fails, the $ld_L op$ is reperformed to ensure that conditions are appropriate for a future operator reduction. An alternative option, assuming support for transactional memory, is to require software to perform updates within a transaction.

We will revisit the need for atomic updates in Section 4.5.2.

4.4 Memory Consistency

In the Dual Address Space memory model, data is either in the global address space, and kept consistent according to the global consistency model, *or* it is localized and *not* kept consistent. An implementation must take care to ensure that there is no in-between: that there is no time in the transition between global to local, or back, that an address is in an undetermined state relative to the consistency model.

4.4.1 Global Consistency Model and Enforcement

As noted earlier, we are assuming a processor based on the Intel Itanium implementation, which uses a relaxed memory ordering model [37]. The model is based on *release consistency*, introduced with the Stanford Dash system [26].

Under release consistency there are two types of synchronizing operations, *acquires* and *releases*. Data operations are allowed to occur in any order, subject to two constraints: 1) all operations after an acquire, in program order, must wait for the acquire to complete; 2) all operations before a release, in program order, must complete before the release completes.

The Dash work identified two flavors of release consistency, determined by the ordering imposed on synchronization operations: RC_{SC} imposes sequential consistency on synchronization operations, while RC_{PC} relaxes the ordering to *processor consistency* (PC) [27, 26]. Itanium introduces a third variety, RC_{TSO} [57], under which hardware ensures that synchronization operations occur in *total store order* (TSO) [78].

No information is publicly available as to how the Altix enforces its consistency model, but we assume the enforcement mechanism is based on the Dash implementation. In the Dash, memory ordering constraints imposed by release consistency were enforced via *fences* [26]. Fences were essentially counters, located in the Dash equivalent of the Hub chip, that kept track of outstanding accesses. The counter was incremented upon issue of the transaction for an access, and decremented upon completion of the transaction as identified by the receipt of either data or the final acknowledgement.

An acquire implies a *full* fence, meaning all future operations must await its performance: the processor stalls until the *acquire* counter is zero. On the other hand, a release implies only an *immediate* fence, meaning that it does not block future instructions. However, it cannot complete until all previous outstanding operations have completed. In an aggressive implementation each release requires its own counter, set initially to the current number of outstanding requests and decremented upon completion of each of (only) those requests.

4.4.2 Localization

According to the contract in the memory model discussion of Chapter 3, in order to be guaranteed sequential consistency, software must ensure that all synchronization operations are global.

A Dual Address Space implementation upholds its end of the contract by ensuring that all operations, *global and local*, are treated the same way with respect to synchronizing operations.

The same counters that keep track of outstanding global data operations also keep track of local operations. Thus, all data operations, local and global, are guaranteed to have been performed after a *release*, and no data operation, local or global, can perform before an acquire performs.

Thus hardware ensures that a globalizing operation immediately after a release receives the latest value(s) from the processors that had the data localized. Similarly, a localizing operation immediately after a release is guaranteed to receive the latest global value. No such guarantees are offered for unsynchronized globalizations and localizations.

4.5 Local Memory

The nomenclature we have adopted for local storage, *local memory*, strongly suggests that it be implemented as a memory (i.e., data that enters does not leave unless it is explicitly invalidated), and the next subsection describes two such implementations. In fact, however, local memory need *not* be implemented as a memory[1]; Section 4.5.2 describes an alternative, local memory implemented as a cache.

4.5.1 Local Memory as *Memory*.

A literal-minded implementation of the Dual Address Space would extend conventional virtual memory: every page in the virtual address space would map to two physical pages, a *global* physical page and a *local* physical page. The advantage of such an approach would be that both local and global memory would be served by the same physical memory pool, allowing per-application balancing of memory resources between global and local. Additionally, it leaves open the possibility of extending both local and global memory by swapping pages to disk.

One drawback to this approach is its requirement for significant modifications to the microprocessor virtual memory system implementation: page-tables, the TLB, and the TLB-filling mechanism. However, the major flaw in the design is that, because of the location of the coherence protocol handlers in the memory hierarchy, they only receive *physical* addresses from processors and directories. Any action required by an operation on a global address that affects a

[1] This insight due to David Wood.

local address, for example a local invalidation due to a global load, would require a mechanism for translation between *local* and *global* physical addresses. Translations in the other direction would also be necessary. The efficient implementation of such mechanisms is difficult to envision.

Our current implementation avoids this problem by using *global* physical addresses for both global and local memory. Local data is stored associatively, as it would be in a hardware cache, on a per-block basis. Block-based storage is a good match for the targeted applications: both multi-dimensional and irregular distributions tend to fragment pages. Block granularity also simplifies the handling of per-block state information required for the coherence protocol, since data may be returned from local memory already modified.

The trade-off between size and speed tends to favor low-associativity designs for large caches. However, to maximize the use of space, our implementation leverages work on software-managed caches that improves the scalability of fully associative memory structures via a hashing scheme [30].

The implementation does have drawbacks. First, while the software-managed cache implementation provides good performance for the average case, a potential limitation is the lack of a maximum time bound for negative results. Another problem is that it puts a fixed limit on the size of local memory. On the other hand, using virtual memory to extend physical memory via page swapping can be slow and Cray implementations, for example, have historically never used it.

4.5.2 Local Memory as *Cache*.

A cache-based implementation of local memory retains the advantages of block-based storage. It also enables faster positive and negative searches, and reduces concerns about size limitations. However, a problem that must be solved is the handling of replacements of localized blocks. The directory's response depends on the state of the replaced block:

Block in state S_L: Silent replacements are not possible because S_L in the cache does not imply S_L at the directory.

1. Cache sends replacement message (no data) to directory.

2. Directory removes processor from sharing list.

The consequence of the replacement, namely that a future ld_L by the same processor is slowed by the refetch, is limited to the replacing processor.

Block in state M_L, *op != nop*: If the (entire) block is involved in an operator reduction computation, i.e., *op != nop*, then the block need not be invalidated in sharers' caches, the directory can update its copy of the block according to the reduction operator and operands.

1. Cache sends replacement message, *with data*, to directory.
2. Directory takes processor off sharing list.
3. Directory reduces block.

Again, the consequence of the replacement is limited to the replacing processor.

Block in state M_L, *op == nop*: If the block is not involved in an operator reduction, then it must be invalidated in all sharers' caches, since the *XOR* reduction algorithm depends on all caches starting from the original block stored at the directory.

1. Cache sends replacement message, *with data*, to directory.
2. Directory takes processor off sharing list.
3. Directory sends invalidation message to all sharers.
4. Sharers send block back to directory.
5. Directory reduces block.
6. Directory removes all sharers from sharing list.

Here the consequences are more widespread, future ld_L/st_L of any sharer is slowed by the refetch. The cache's replacement policy should strongly favor keeping modified blocks cached.

The invalidation of sharers could be avoided if the directory were able to save modified blocks until they were 1) required for an *XOR* reduction, *or* 2) refetched by the modifying processor. The

optimization can be implemented as a fully associative table indexed by block address. The table cannot be too large because it needs to be searched in parallel with every directory access. If a block is found for the requesting processor it is returned, otherwise the directory returns its copy of the data. A block is inserted into the table upon replacement if it is in state M_L, *op* == *nop*, and there is room in the table. If the table is full, the directory falls back on the invalidation approach. Upon reduction, if there are multiple blocks associated with the address in the table, the directory reduces them to one, and sends the reduced block *and* the original to the requesting processor.

4.5.2.1 Atomic Updates Revisited

Note that the scenario presented in 4.3.2, where there was an interruption before the st_L of an updated value involved in a reduction, can also arise if local memory is cache-based. Suppose the block is replaced (and thus reduced at the directory) before the st_L; again, the store operates in the wrong context. Thus, cache-based local memory also requires atomic updates of reduction data. The solutions noted in that section, an additional local *cmpxchg* instruction or updates-within-transaction, also apply here.

4.6 Summary

This chapter has described a full-scale implementation of the Dual Address Space Architecture, an extension of the SGI Altix design, itself based on the SGI Origin. The chapter began by describing the layout of a *node*, the basic building block in all the aforementioned systems. Modifications are largely limited to the coherence protocol implementation in the processor and directory interfaces, along with the addition of local memory to the memory interface.

The chapter continued with a detailed description of a directory-based implementation of the User-Controllable Coherence cache coherence protocol, initially assuming single-word cache blocks, and then describing in detail the ramifications of multi-word cache blocks. The chapter then described support for the consistency model introduced in Chapter 3. Finally, the chapter closed with a discussion of the implementation of local memory.

Chapter 5

Simulation Platform

This chapter introduces the simulation platform we used to perform the experiments described in Chapter 1 and Chapter 6. The simulator both accurately models an existing CC-NUMA platform, the SGI Altix 4700, and incorporates the protocol modifications described in Chapter 4.

The chapter is divided into three parts. The first part describes a mechanism for directing a multi-threaded reference stream to a single back-end memory system simulator – a modified version of the Slicc/Ruby portion of the Gems simulator from the Multifacet group at the University of Wisconsin [77] – while ensuring that the references are simulated in the same global order that they would have occurred in a multiprocessor. In Section 5.1 we introduce the algorithm by describing the implementation of a simulator prototype which blocks on every reference. Since the Itanium2 [53] processor on which the Altix architecture is based does not block on most memory references, the actual algorithm used by our simulator is more complicated. Section 5.2 describes in detail how we model those elements of the Itanium2 micro-architecture essential for the accurate simulation of memory references.

The second part, Section 5.3, describes how we have used this mechanism, in conjunction with a custom parallelization of the Ruby back-end, to implement a parallel simulator that models a 1024 processor Altix 4700 system. Finally, the third part, Section 5.4, presents an evaluation of the accuracy and performance of the parallel simulator.

```
void
reference(UINT64 tid, UINT64 pred,
          UINT64 iaddr, UINT64 addr)
{
    if (pred == 0) {
        cycle[tid]++;
        return;
    }
    sem_up(&sim_sem);
    sem_down(&thread_sem[tid]);
}

void
incr_cycle(UINT64 tid, UINT64 iaddr)
{
    cycle[tid]++;
}

void
Instruction(INS ins, void *v)
{
    switch(INS_Category(ins)) {
    case TYPE_CAT_STORE:
    case TYPE_CAT_LOAD:
        PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR)reference, ...);
        break;
    default:
        PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR)incr_cycle, ...);
    }
}
```

```
void
sim_loop(void* unused)
{
    int i;
    sem_down_cnt(&sim_sem, max_threads);

    while (!done) {
        UINT64 min = MAX_UINT64, min_tid = 0;

        for (i = 0; i < max_threads; i++) {
            if (go[i] && cycle[i] < min) {
                min = cycle[i];
                min_tid = i;
            }
        }
        cycle[min_tid] += 1;
        sem_up(&thread_sem[min_tid]);
        sem_down(&sim_sem);
    }
}
```

Figure 5.1 Prototype code.

5.1 Simulator Prototype

When a multi-threaded front-end is used to generate memory references for a single-threaded back-end memory system simulator, the accurate modeling of synchronization and cache coherency protocol state requires that care be taken to ensure that threads execute instructions in an order that is consistent with a multiprocessor execution. This constraint imposes a synchronization requirement between the threads and the simulator. The prototype simulator uses semaphores to implement that synchronization.

Introduced by Dijkstra [20], semaphores are essentially counters that control access to resources. The interface to semaphores consists of two components: *wait* and *signal* (originally *P* and *V* respectively). A typical implementation of *wait* causes the initiating thread to sleep until it is *signal*'d by another thread.

Semaphores provide a simple mechanism to synchronize the producer/consumer relationship between the application threads and the simulator:

1. Consumer *wait*s for producer

2. Producer produces, then *signal*s the consumer and *wait*s for consumer.

Unfortunately, in the absence of a lightweight thread library, the efficient implementation of semaphores is still an open problem: because *wait* and *signal* require atomicity and an efficient sleep mechanism, they are typically handled in the operating system kernel, implying a costly context switch on every call.

Figure 5.1 demonstrates the prototype algorithm stripped down to its bare essentials. We make use of Pin [49], a dynamic instrumentation tool from Intel, to instrument a multi-threaded program such that every program reference causes a synchronization with a "simulator" running in a separate thread. Application threads run through a Pin tool based on the code on the left-hand side of Figure 5.1. The instrumentation routine *Instruction()* associates the *reference()* analysis routine with every memory reference. As a result, all dynamic *ld* and *st* instructions will call *reference()* before they execute. All other instructions will simply call the *incr_cycle()* analysis routine.

Not shown in the figure, the Pin tool creates a simulator thread before starting the application threads. The simulator executes the code on the right-hand side of Figure 5.1. The algorithm then proceeds as follows:

- The simulator immediately *wait*s on a "multiple count" semaphore: the semaphore must be *signal*'d multiple times – in this case once by each thread – before the simulator can continue.

- All application threads are then started as normal by Pin. When an application thread hits its first memory instruction it *signal*s the semaphore that the the simulator is waiting on (modeled by the dotted arrow in the figure) and then *wait*s on its own semaphore.

- After all the threads have *signal*'d and *wait*'d, the simulator thread wakes up and "simulates". The prototype simulator does nothing, but the real simulator can now safely simulate time up till the next completion of a reference.

- The prototype simulator then releases the thread with the oldest reference which executes until it reaches a memory instruction, wakes up the simulator, and so on.

5.2 Full-Scale Simulator

As noted in the introduction to this chapter, Itanium2 instruction issue does not block on most memory references, but rather on instructions that depend on the results of memory references. In this section we describe how we modified the prototype to model micro-architectural features required to more accurately simulate the timing of Itanium2 reference generation.

5.2.1 Handling Multiple Outstanding Loads

Since reads do not necessarily block, the placement of semaphores to synchronize instrumented application threads with the simulator is no longer obvious: blocking is not associated with the *ld* instruction that initiated the request, but rather with another, arbitrary, instruction that uses the register into which the value is read from memory.

Our approach to this problem is to use indirection. We associate semaphores with a table of outstanding requests, and precede each register use by a look-up into the table to determine whether the register is associated with an entry in the table; if it is then the thread blocks on the associated semaphore. Our Pin tool's analysis routines maintain two structures to help implement the task:

1. *MemQueue*, with an entry for each outstanding request, keeps track of outstanding requests; each request is given a slot in the *MemQueue* and each slot in the *MemQueue* has an associated semaphore.

2. *RegReady*, with an entry for each CPU register, keeps track of when that register is ready to be read. Instructions that write registers use the structure to indicate the cycle at which the register will be ready for reading. For example, arithmetic operations that write a register would update the *RegReady* entry with the current cycle plus 1 for integer operations or 4 for floating point operations. *Ld* instructions, on the other hand, insert a pointer to a reserved *MemQueue* slot into the *RegReady* entry. The thread then continues processing instructions.

If an issuing instruction depends on a register whose *RegReady* entry contains a pointer to an entry in the *MemQueue*, then the thread returns control to the simulator and blocks on the

semaphore associated with the *MemQueue* entry. When time has been simulated up till the reference completes, the simulator replaces the pointer in the *RegReady* entry with the current cycle value and returns control to the thread.

Virtual Registers. The discussion above has so far left out an element of the Itanium2 architecture that significantly complicates our efforts: Itanium instructions refer to virtual registers rather than physical registers. Virtual registers are a mechanism useful for both implementing a register stack at procedure calls and generating more efficient software pipelining code.

At procedure calls, Itanium compilers allocate a set of fresh local registers for use in the procedure and to pass parameters. Local registers in the new procedure always begin at register r32. Under the covers hardware maps the virtual registers to new physical registers, sliding a window of virtual register names forward over the physical register file. Upon return from the procedure, hardware pops the stack, sliding the window back so that the logical registers map back to the physical registers they mapped to before the call.

Virtual registers are also used to implement a rotating register mechanism, which enables compilers to generate extremely compact software pipelining code, as in the following example:

```
loop:
      (p17) st4 [r11]=r33,4
            ...
      (p16) adds r32=100,r36
            ...
            br.ctop.sptk.few loop
```

The value written to r32 by the *adds* instruction in one iteration of the loop is read by the *st4* instruction on the next iteration, though the store lexically precedes the add. Predicate registers also rotate, facilitating pipeline start-up and wind-down.

Fortunately, both of these cases are marked in the instruction stream by the use of specialized branch instructions: *call* and *ret* mark procedure calls, while *ctop* and *wtop* indicate rotation points.

In our implementation, the *RegReady* structure is actually associated with an "infinite" array of physical registers, and we keep track of where virtual registers point in the physical registers after

pushes, pops, and rotations, via pointers. We can then use a *FindReg()* routine to map a virtual register to physical register in order to set or read the ready time for the physical register. Since we assume an infinite number of physical registers we do not model the spilling of register contents to memory necessary when hardware runs out of physical registers.

5.2.2 Other Micro-architectural Considerations

Instruction issue and counting cycles. The Itanium architecture has strict limitations on the number of instructions that may issue each cycle and the order in which they may issue. The compiler groups instructions into "bundles" that obey these constraints. One of the runtime values that an analysis routine can receive as a parameter from the instrumentation routine is the value of a stop bit, which indicates whether the current instruction is the last instruction of a bundle. With that information our analysis routines can ensure that at most two bundles are issued before incrementing a cycle counter.

The cycle counter is also incremented 1) when an issuing instruction depends on a register whose *RegReady* value is greater than the current cycle, and 2) when control returns to a thread after a *st* instruction completes.

Features not modeled. Our model of the Itanium2 micro-architecture is far from complete. Our assertion that it is sufficient makes some assumptions about our workload:

- We do not model the instruction stream coming from memory, so we are assuming that: 1) L1 instruction cache misses are rare and do not affect performance, and 2) instructions do not conflict with data in the L2 and L3 caches.

- We do not model branch prediction hardware so we are assuming that most branches are predicted correctly.

- Finally, we do not model finite functional units so we are assuming that in our workload, structural hazards are rare and do not affect performance.

5.3 Simulator Parallelization

As described in the previous sections, the full simulator consists of two parts. The front-end performs cycle-accurate simulation of Intel Itanium2 cpus and feeds memory references to the back-end memory system simulator. The back-end determines the number of cycles it would take to satisfy the references and returns control back to the front-end. The previous sections described how a multi-threaded front-end can be used to produce references for a monolithic back-end. In this section we describe how we have parallelized the back-end, resulting in a fully parallel multi-processor simulator.

We have parallelized the simulation of the back-end at the granularity of a node, where a node consists of the hardware associated with a network endpoint. On the Altix, this endpoint is referred to as a SHUB, and consists of a memory controller and associated memory, a network interface, and two dual-core Itanium processors. Thus, each thread of a multi-threaded application runs on its own front-end, and four front-ends feed each back-end.

In our original parallel implementation, the front-end was a Pin tool operating on a multi-threaded application as described in the previous sections. Each thread group, consisting of one back-end and its front-ends, was pinned to an Altix processor. The semaphore calls described in the previous sections were implemented as "spin_yield" semaphores: a thread implicitly blocked on a semaphore by continuously calling the *sched_yield* system call until the condition it was waiting for was met. In this fashion, scheduling and descheduling were left to the operating system, resulting in significant overheads.

However, the ultimate cause for our abandonment of this implementation was a simple artifact of the way Pin managed application memory which prevented the simulator from simulating more than 64 threads. While this problem likely would have been easily fixed had we had access to the source code, Pin is closed-source and the original Itanium version we were using was no longer supported, so we were forced to change platforms.

The current implementation uses Hewlett Packard's recently *open*-sourced Itanium simulator *Ski* as a front end. In this implementation only the back-ends are true system threads, while the

front-ends are essentially user-level threads. We have implemented the semaphores using calls to *make_context()* and *swap_context()*, POSIX mechanisms for creating user-level thread libraries. This allows much more efficient explicit passing of control between simulated front-end threads and their back-end.

While the move to Ski had many advantages – chief among them allowing us to simulate an order of magnitude more threads – there were at least two significant disadvantages:

1. The front-end simulator now must explicitly manage the simulated application's memory, rather than leaving the task to Pin. Every memory access must go through a translation from virtual page address to physical page address. The page table is implemented as a large, *parallelized* (since it is accessed by multiple front-end threads simultaneously), bucket hash table. As we are only concerned with additions to the table, we enable parallel insertions with a simple lock-free algorithm: we use the test-and-set atomic instruction to add the current element to the front of the list *only* if the front of the list has not been changed by another processor.

2. The front-end is required to handle thread-management issues, which again, had been handled implicitly by Pin. Unfortunately Ski does not contain support for system calls used by the *pthreads* library. Rather than attempting to implement such support, we chose to implement a handful of calls typically found in a threading library as system calls, allowing them to be handled directly in the front end. Among the calls we support are: *parallel* which spawns threads; *barrier* which implements a barrier; *lock* and *unlock* which enable point to point synchronization; and *get_thread_num* and *get_num_threads* which return the calling thread's id and the total number of threads respectively.

5.3.1 Network Parallelization

The back-end models the network of a 256 node Altix system, including: the dual plane, fat-tree topology; NUMA-link routers; and the SHUB network interface chips. The major problem facing parallel network simulators is the maintenance of a global clock: in order to avoid timing

violations (i.e., to ensure that packets are not received later than they should be), simulation threads must be synchronized.

There are two approaches to parallelizing network simulators based on discrete event simulation: the conservative approach and the optimistic approach [55]. Under the conservative approach, in order to ensure no message is received later than it should be, simulator threads synchronize a minimum of every *network hop latency* cycles. Under the optimistic approach, simulators synchronize less often but must save state in order to allow roll-back if a violation is detected. We take the conservative approach, performing a global barrier every *network latency* cycles.

In the current implementation, we only model the outermost routers of the network. At each barrier, *all* back-end threads synchronize, and each back-end thread copies packets destined for processors in its domain from every other thread. Here, as everywhere, we have taken extreme care to ensure that remote memory is only *read* not *written*. Nevertheless, this approach is a significant impediment to scalability due to load imbalance: every thread is affected by every other thread at every barrier. In retrospect, a better approach would distribute the barriers according to the physical routers in the network. This would also allow more accurate simulation of contention at internal routers.

5.3.2 User-Controllable Coherence

Applications that do not use localizing instructions run as they would on unmodified hardware. Applications with localizing instructions use exactly the same simulated hardware, but with the additional states, transitions, and actions described in Chapter 4. Both the standard MSI protocol and the extensions are described using the *Slicc* protocol description language *Slicc*. The *Slicc* compiler automatically generates the protocol state-machine component of Ruby, the back-end memory system simulator. Ruby also provides a test-bed for rigorously stressing the coherence protocol to find race-conditions and other errors, of which we have made extensive use.

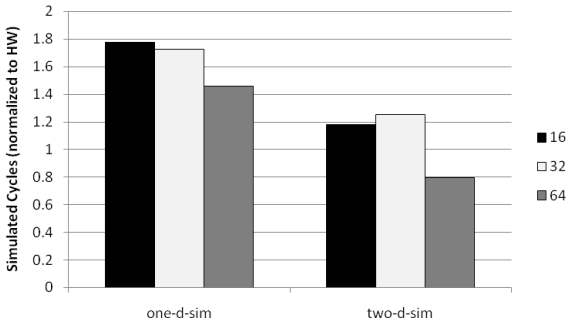
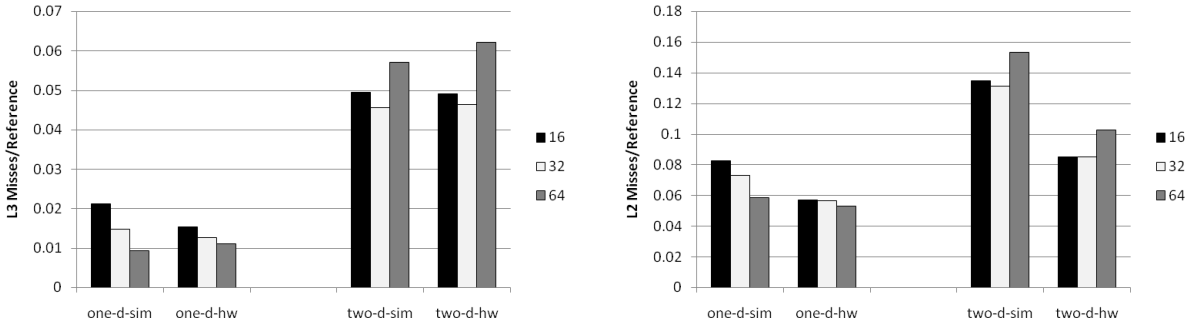


Figure 5.2 Per-processor cycle counts predicted by the simulator vs. those measured by performance counters on hardware.



(a) (b)

Figure 5.3 Per-processor cache miss rates predicted by simulator vs. those measured on hardware: (a) L3 cache, (b) L2 cache. Note that the simulated L3 miss rates are much more accurate, and that, as discussed in the text, low-latency L2 inaccuracies have more effect on the simulated performance of *one-d*.

5.4 Simulator Validation and Performance

The first part of this section describes our efforts to validate the simulator against actual hardware implementations. The second part briefly discusses the performance of the parallelized simulator. We performed the hardware (and simulator) runs on a 2048 processor SGI Altix 4700, an experimental element of the Columbia cluster located at the NASA Advanced Supercomputing facility [56]. We present results from runs of the *one-d* and *two-d* variants of the application described in next chapter.

5.4.1 Validation

Tying the simulation platform to an existing system is a double-edged sword. On the one hand, demonstrating that the simulator matches existing hardware can add credence to the results of simulations of *extensions* to the hardware. However, doing so adds the significant burden of trying to exactly match the behavior of often fickle hardware.

Microprocessor simulators are notoriously difficult to validate [19]. Validating a *system* composed of tightly interconnected microprocessors, in a production environment, significantly adds to the challenge. In addition to the vagaries of hardware, there are difficulties imposed by the environment, including: long queue waits, uncertain processor distribution in the network once the job runs, and noise due to traffic from other processors.

Figure 5.2 compares cycle counts predicted by the simulator with those measured on hardware using performance counters; the results indicate two problems:

- The simulator significantly over-predicts the number of cycles required to complete an iteration. Moreover, the difference in performance for the *one-d* variant is substantially greater than the difference for the *two-d* variant, potentially calling into question results in the next chapter that demonstrate improvement over the *one-d* variant.

- The *two-d* variant experiences a drop-off in performance at 64 processors not predicted by the simulator.

The two problems indicate two different sources of inaccuracy: the first is a result of inaccuracies in the uniprocessor simulation while the second is due to inaccuracy in the network simulation.

Since *two-d* suffers more L3 misses than *one-d* (see Figure 5.3(b)) and thus more memory and potentially network traffic, performance is dominated by long latency events. Therefore, short-latency inaccuracies in the uniprocessor simulation will have more effect on the performance of the *one-d* variant.

As for the unpredicted performance drop for *two-d* at 64 processors, setting aside potential inefficiencies in the hardware run (i.e., assume the run was not affected by suboptimal processor

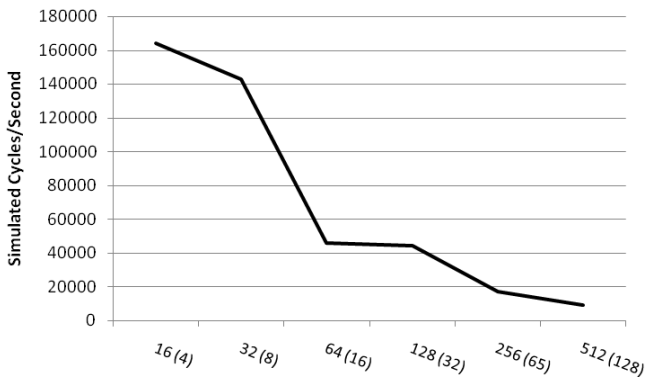


Figure 5.4 Performance of the parallel simulator, as measured in simulated cycles per second (per processor). The x-axis labels indicate the number of simulated threads followed, in parentheses, by the number of processors performing the simulation.

placement and/or network contention from other jobs), the simulator's contribution to the discrepancy is likely due to inaccuracies in the simulated network model. As noted in Section 5.3, the simulator does not model the internal routers of the network, but instead computes a fixed latency for packets based on the number of hops to the destination. Figure 5.3(b) indicates very high L3 miss rates for *two-d* and, based on data we present in the next chapter, many of the misses are to remote data. These factors indicate pressure on internal network routers potentially leading to high queuing delays not currently modeled.

5.4.2 Performance

Finally, we turn our attention to the parallel performance of the simulator. Figure 5.4, which presents performance as measured by simulated cycles per second per processor, appears to demonstrate an unfortunate lack of scalability as the number of simulated processors increases.

Note, however, that simulated *cycles* is not necessarily a fair measure since cycles do not necessarily require equal amounts of time to simulate. Consider, for example, the difference in simulation time for cycles in which a simulated thread makes no memory references, versus the simulation time for cycles in which thread references hit in a cache, versus cycles in which thread

references require injection into the memory system back-end. Moreover, as the number of simulated processors increases the likelihood that a reference will need to be injected into the memory system increases (for this workload at any rate).

A more fair measure would be simulated time on a uniprocessor divided by simulated time on a multiprocessor. However, given the size of the simulations we are performing, both in terms of number of processors and data per processor, it is unlikely that the simulation would even run on a uniprocessor.

Nevertheless, there is certainly room for improvement in the simulator's parallel performance. We have already noted one problem: load imbalance due to the need for global synchronization every *network latency* cycles. Another problem is the potential for all-to-all communication of data after each barrier, as back-end threads exchange packets. Both of these problems would be addressed by modeling the internal routers of the network, though thought and care would be required to ensure that processors that perform the router simulation are kept busy and are appropriately placed in the network of the simulator's host.

5.5 Summary

This chapter has described the implementation of a cycle-accurate simulator, which models a 1024 processor SGI Altix 4700 system. The simulator consists of two parts: a reference generating front-end and the memory system simulator back-end. The chapter described the parallelization of both parts, resulting in a fully parallelized system simulator, demonstrated capable of accurately simulating 512 threads running a real application across 128 host processors. Finally, the chapter provided validation and performance data for the simulator, noting several opportunities for improvement in both areas.

Chapter 6

Application One: HYCOM

In this chapter we evaluate User-Controllable Coherence in the context of an ocean modeling code, HYCOM [11], the **Hy**brid **Co**ordinate **M**odel, in production use at the Naval Oceanographic Office (NAVOCEANO).

As the name suggests, HYCOM uses a hybrid approach to represent ocean coordinates because no single coordinate system is optimal everywhere in the ocean. In HYCOM, the choice of a vertical (depth) coordinate system for a discretized point depends on where the point is located in the ocean. In particular, *isopycnal* (constant density) coordinates are used in the open, stratified ocean (where density is dependent on depth). However, a "continuity equation" transitions to a second coordinate system (theta-levels) for points in shallow coastal regions, or a third coordinate system (z-levels) for points in the mixed layer or unstratified seas, where the density is not as dependent on depth.

The model solves the partial differential equations defining ocean behavior *explicitly*, using finite differencing via a nine-point stencil computation. As noted in the introduction, a general problem with explicit solution methods is that they tend to require a small time-step when modeling fast changing physical processes [18]. To address this problem, HYCOM divides the model into two parts: the *baratropic* phase of the computation models faster moving surface physical processes, while the *baroclinic* phase models the slower moving physical processes operating on the entire volume. This division enables the use of multiple time-step resolutions: by solving multiple iterations of the faster-moving baratropic equations per iteration of the baroclinic equations, HYCOM avoids the need for a small time-step throughout.

```
do 76 k=1,kk
  margin = mbdy - 1
  do j=1-margin,jj+margin
    do l=1,isu(j)
      do i=max(1-margin,ifu(j,l)),min(ii+margin,ilu(j,l))

        ...

      enddo
    enddo
   enddo
 enddo
```

Figure 6.1 Typical loop-nest from the MPI version of HYCOM.

6.1 Parallelization

This approach also permits the use of a single, simple parallelization strategy throughout, which should be suitable for distributed memory platforms. The three dimensional volume of the ocean is divided via a two dimensional domain-decomposition for longitudinal and latitudinal axes, while depth is divided into a fixed number of layers. Thus each processor is responsible for a column of ocean. Column dimensions, and therefore processor data and computational partitions, are determined *statically*, before the program is compiled.

However, the desire to avoid useless computation involving land elements results in both a slightly irregular data distribution and irregular loop bounds.

Figure 6.1 presents a typical loop-nest from the MPI version of HYCOM. The k loop iterates over ocean layers, while i and j iterate over the two-dimensional partition of the k'th layer owned by a given processor. The l loop, along with the isu, ifu, and ilu arrays, accounts for holes in a partition due to land mass. The $margin$ variable controls how much of the six-element-wide halo of non-local data (explicitly communicated previously) is used by a given loop-nest.

Another interesting aspect of HYCOM's parallelization strategy is its use of an extended halo of non-local data to minimize synchronization. HYCOM uses an *operator splitting* approach to solving differential equations: in essence, rather than using the same right hand side for an entire time-step, this method uses multiple intermediate left hand side results within a time-step[81]. This

use of intermediate results leads to intermediate cross-processor data dependencies, which in turn result in more communication per time-step.

While it is not possible to change the *volume* of communication required to honor these dependencies, HYCOM makes use of redundant computation to minimize the number of synchronization events required during a time-step.

Comparison with POP POP [76], the Parallel Ocean Program, is a competing ocean model originally developed at Los Alamos National Laboratory. POP comprises one quarter of the Community Climate System Model. Like HYCOM, it is an MPI-based code, written in Fortran90.

The most significant differences between the two models are:

- POP uses a fixed depth coordinate system, z-level. However, this difference is not fundamental; there is an ongoing effort to incorporate HYCOM-like hybrid coordinates into a new version of the code [48].

- Like HYCOM, POP uses an explicit method for most computation phases. However, for the faster moving baratropic equations, POP uses the implicit conjugate gradient method. While this choice enables a constant time-step, it has significant implications on the parallelization strategy. In particular, the conjugate gradient method requires a significant number of collective communication events, which tend to have a severe impact on scalability. Significant efforts have been undertaken to minimize the impact, including redistribution of data for different computation phases [40].

- Unlike HYCOM, POP does not feature irregular loop bound: while tiles that consist only of land are discarded, there is no mechanism for avoiding computation on land elements within a partition. POP2 permits multiple tiles per processor, enabling finer grained data distribution at the expense of more complicated communication.

```
do 76 k=1,kk
  do j=myfirstj,mylastj
    do l=1,isu(j)
      do i=max(myfirsti,ifu(j,l)),min(mylasti,ilu(j,l))

        ...

      enddo
    enddo
   enddo
 enddo
```

Figure 6.2 The sample loop-nest transformed for SPMD-style OpenMP.

6.2 Artifactual and Inherent Communication in HYCOM

In this section we illustrate the perils of artifactual communication, and the impossibility of performing an explicit communication optimization in a global address space, using HYCOM, a well-known, production ocean modeling code.

6.2.1 SPMD HYCOM

We transform the loop-bounds for SPMD-style OpenMP as shown in Figure 6.2. The values for $myfirstj$, $mylastj$, $myfirsti$, and $mylasti$, are computed by each processor during the initialization phase of HYCOM. We are also forced to introduce extra synchronization between loop-nests, but again leave details for discussion in Section 6.2.3. Aside from these explicit synchronizations, all communication of remote data is implicit, initiated when a processor touches an element owned by another processor (that has not already been cached).

Finally, we note that the existence of holes in the data partition owned by a processor, and the provision for them evidenced by the l loop, significantly complicates efforts to achieve load-balance in HYCOM. A major advantage of programming in a global address space is that modifying a processor's partition simply requires modifying loop-bounds variables such as $myfirstj$. Changing partitions is much more complicated in MPI, where remote data must be explicitly found, named, and moved.

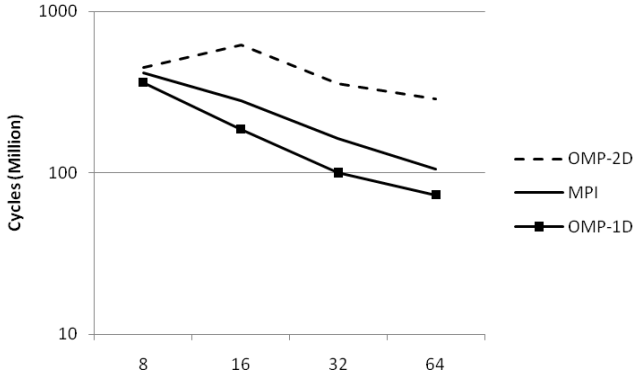


Figure 6.3 Results from HYCOM runs using the GLBA0.72 data set on an SGI Altix BX2. Here, as in all figures in this chapter, "cycles" indicates the cycle count measured by performance counters on the last processor to complete one time-step (the second).

6.2.2 Artifactual: Multi-dimensional Domain Decompositions

HYCOM's two-dimensional partitioning of each layer of ocean across processors, described above, is an example of a multi-dimensional *domain decomposition*. Domain decompositions co-locate all data associated with a point from a discretized representation of a physical domain on a single processor, and distribute all such points evenly across processors while, to the extent possible, keeping points close in physical space on the same processor.

Multi-dimensional domain decompositions distribute more than one dimension across processors for two reasons. First, doing so reduces communication volume as the number of processors increases, so long as data-dependencies exist only between neighboring elements. Second, distributing more than one dimension increases available parallelism: if data is distributed along a single dimension then the maximum number of processors is determined by the number of elements in that dimension. If it is distributed across two dimensions then the maximum parallelism is determined by the multiple of the elements in the first dimension and the number of elements in the second.

Far from reducing volume, artifactual communication effects from multi-dimensional distributions can result in significantly increased communication for DSM implementations, like the Altix, where data ownership is determined at a page granularity. The problem is that multi-dimensional

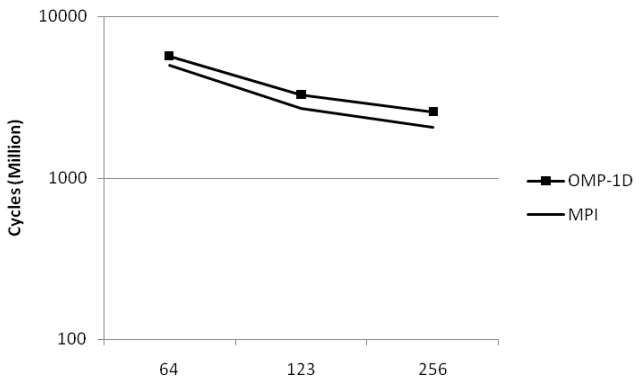


Figure 6.4 Results from runs using the GLBA0.08 data set, a larger data set which enables the use of more processors.

arrays are stored linearly in memory. This means that when a global array is distributed in more than one dimension, the portions of that array owned by one processor are not contiguous.

If these non-contiguous chunks are smaller than a page, a mismatch between the "owner" as defined by the coherence protocol and the "owner" as defined by the data decomposition, can result in a severe increase in communication. If the data decomposition owner does not match the protocol owner, then *every cache miss results in communication*, as opposed to a trip to local memory.

Figure 6.3 demonstrates that artifactual communication resulting from a two-dimensional distribution of the arrays in $cnuity()$ can result in significant performance degradation. The OMP-1D and OMP-2D variants described in the figure are different only in the values assigned to the loop-bound variables from Figure 6.1 ($myfirstj$, $mylastj$, $myfirsti$, and $mylasti$). In OMP-1D, only the j loop is parallelized (implicitly distributing the data referenced by the loop in that dimension) while OMP-2D parallelizes both i and j loops.

6.2.3 Inherent: Communication Frequency Optimization

HYCOM uses an "operator-splitting" approach in its solution of the differential equations that define ocean behavior. In this approach, results computed within a time-step are used in the same time-step, increasing data dependencies within a time-step relative to other approaches [81]. This increase in data dependencies in turn leads to increased communication frequency.

Distributed address space implementations can mitigate this effect by performing redundant computation on an extended halo of remote data. Note that the halo is gradually used up as computed results are used. The purpose of the $margin$ variable, described above in Section 6.2.1, is to ensure that only valid portions of the halo are used.

Implementations that take full advantage of the global address space – by treating each data array as single, global entity, simultaneously operated on by multiple threads – cannot perform such redundant computation without disturbing neighboring threads. Instead they must obey the data dependencies, leading to increased synchronization, as described in Section 6.2.1.

Specifically, the SPMD OpenMP versions of $cnuity()$ execute *more than 250* dynamic barriers per time-step. Since the purpose of the barriers is to ensure that data written by one processor is read by another, the number of data communication events is similar. The MPI version of $cnuity()$, on the other hand, communicates twice: once at function entry and once midway through.

While Figure 6.3 indicates that this increase in synchronization and communication frequency is not an issue at low processor counts, Figure 6.4 supports the intuition that synchronization and communication overhead increase significantly as the number of processors increases.

6.3 Experimental Evaluation

This section describes the results of simulating runs of HYCOM on up to 512 processors of a Dual Address Space Architecture implementation using the simulator described in Chapter 5.

The full distribution of HYCOM is too large for simulation experiments. Rather than using a smaller benchmark, our approach was to find a representative subset of the actual application for experiments. To that end we have transformed one module of HYCOM – $cnuity()$, which computes the continuity equation described above – to use SPMD-style OpenMP exclusively for both single- and two-dimensioned data distributions. $cnuity()$ contributes roughly 15% to the total runtime of HYCOM. The Fortran90 file which implements $cnuity()$ contains over 1200 lines of code, comprising some 20 loop-nests that read and write more than 40 arrays.

The runs use two of the three data sets provided with the *GLBLA0* global ocean benchmark [79]. The three data sets provided with the benchmark are:

- GLBA0-72, referred to hereafter as *small*, discretizes the global ocean at 72-degree resolution, resulting in a 500x382 element grid per layer of ocean. The benchmark distribution advises using up to 64p for the MPI version of the code. We used 16, 32 and 64 processors in our simulations.

- GLBA0-24, which we refer to as the *medium* data-set, discretizes at a 24-degree resolution, resulting in a 1500x1100 grid. While the benchmark distribution only provides for up to 128 processors, we also used 256 and 512 configurations in our simulation experiments.

- Finally, GLBA0-08, discretizes at 8-degree resolution, providing sufficient data for thousands of processors. This data-set is too large for our simulator to manage efficiently at this point.

Each simulation experiment ran two iterations of $cnuity()$, the HYCOM module that computes continuity equations. We used the first iteration to warm up caches and local memories, and we present results for the second. *Small* data set runs use the full benchmark; however, to keep simulation time manageable for the larger *medium* runs, we truncated the outer k-loop to run only six iterations rather than 26.

6.3.1 Communication

Figure 6.5 presents data for communication due to last-level cache misses. Each bar represents the total number of last-level cache misses suffered during the second iteration of $cnuity()$. The bars are further divided into four categories:

1. *local*: misses that were satisfied in local memory, and did not cause communication.

2. *compulsory*: non-local misses due to the initial reference to the data. Since we present results from the second iteration, the number of compulsory misses is unsurprisingly very low.

3. *capacity/conflict*: non-local misses due to limited cache capacity or due to low associativity.

4. *coherence*: non-local misses resulting from invalidations by other processors, or upgrading from shared to writable.

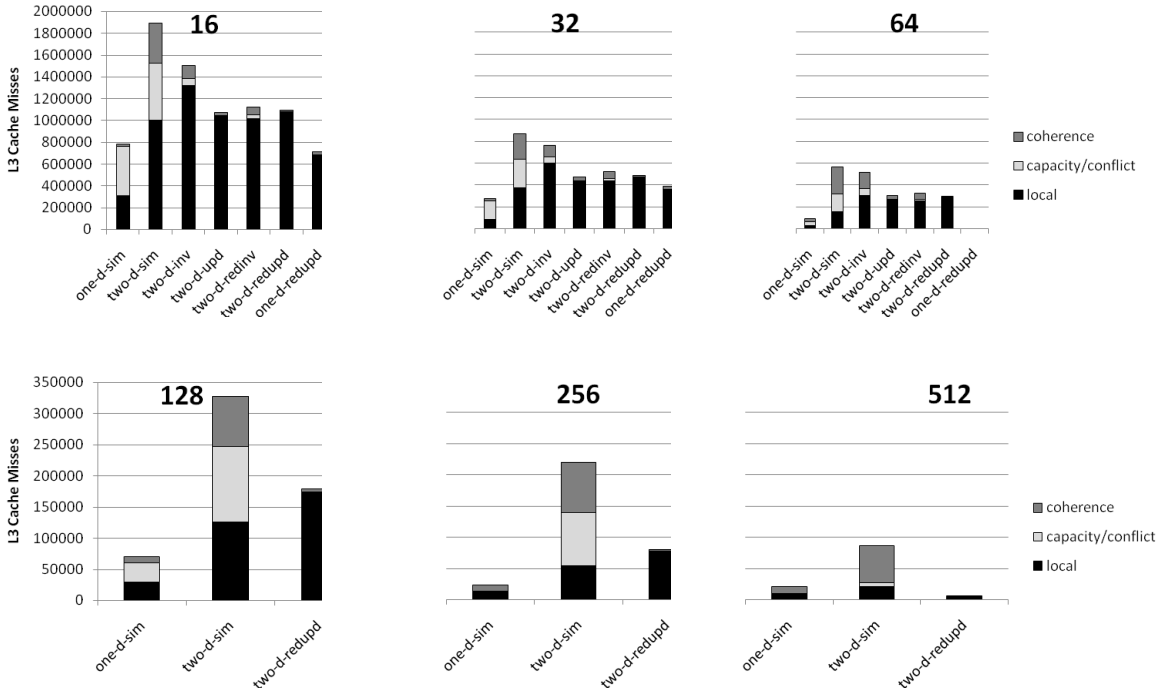


Figure 6.5 Last-level cache miss categorization for small (top) and medium (bottom) data set.

We turn our attention first to the *one-d-sim* and *two-d-sim* results, taken from runs using the standard coherence protocol of a one-dimensional distribution of data and a two-dimensional distribution respectively. As expected, due to artifactual communication *two-d-sim* suffers significantly more non-local cache misses than *one-d-sim*. However, the number of misses due to coherence increases for both *two-d-sim* and *one-d-sim*, as the number of processors increases, again not surprisingly since the computation-to-communication ratio is shrinking.

Turning next to the variants that make use of the dual address space to reduce artifactual communication for the two-dimensional distribution (shown only for *small* distribution), we see that all variants reduce the total number of cache misses, and dramatically reduce the number of non-local misses. However, as the results of the previous section would indicate, the *update* versions suffer significantly less communication than the *invalidate* versions. Finally, the use of redundant computation further reduces the number of non-local misses from the update-only version.

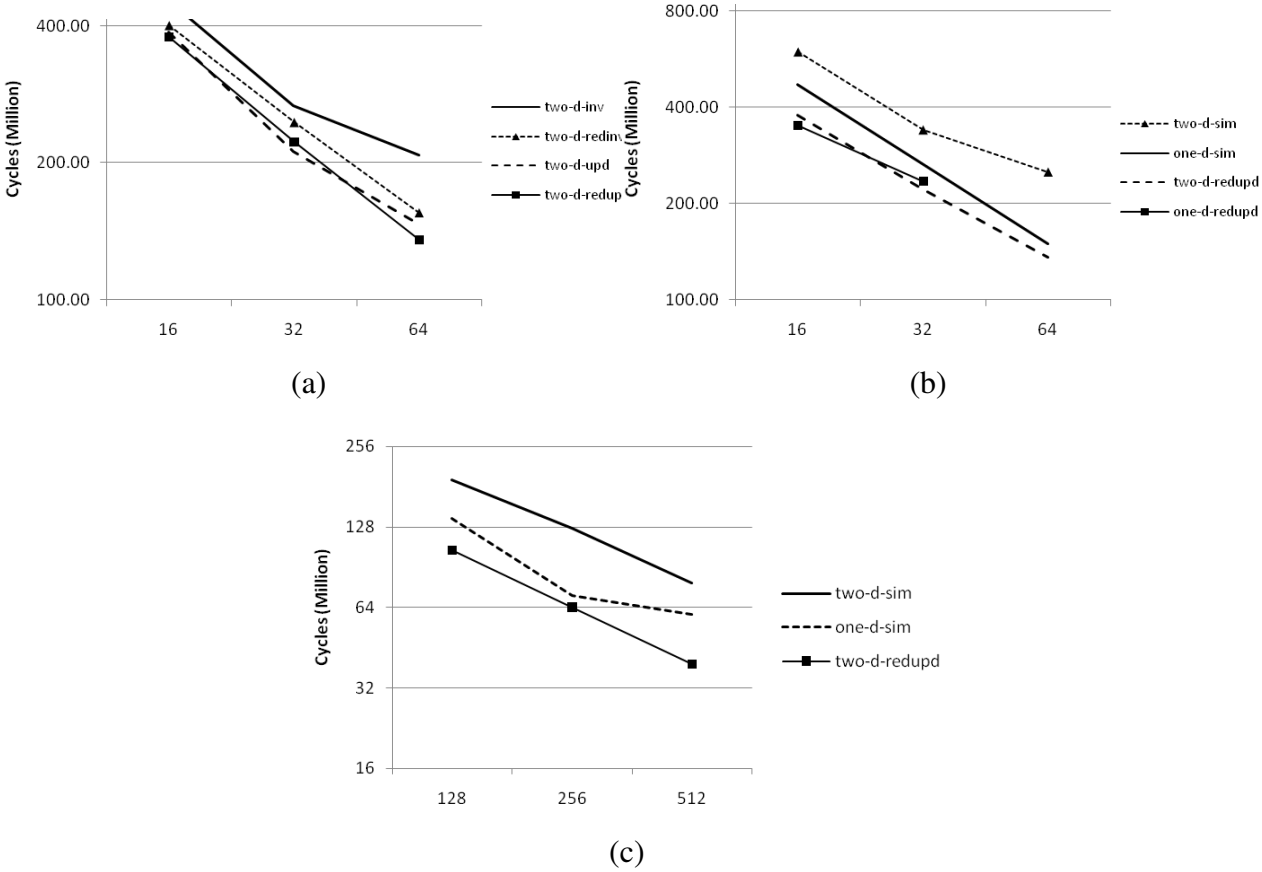


Figure 6.6 (a-b) Performance for small data set, comparing (a) different localization strategies, and (b) the best strategy with the standard protocol. (c) Performance for medium data set, versus the standard protocol

However, in almost all cases the total number of misses is significantly higher for *two-d-redupd* than for *one-d-sim*, implying there is still a performance hit for using a two-dimensional distribution. We can safely attribute these misses to conflicts: they are definitely not coherence or compulsory misses, and since *one-d-sim* references the same amount of data, they are not due to capacity. Therefore the two-d variants would all benefit from optimizations to reduce conflicts.

6.3.2 Performance

Figure 6.6 presents performance data in the form of (log-based) scaling curves. Figure 6.6(a) compares the performance of the four localized two-dimensional variants introduced in the previous chapter. Again, as predicted by the performance of the model, the *update* variants, which leave

data localized, perform significantly better than the *invalidate* variants, which move data from the local to the global address space for communication. Unsurprisingly, using redundant computation to reduce communication helps more at higher processor counts.

Figure 6.6(b) demonstrates that the best performing localize two-dimensional variant consistently outperforms even the one-dimensional variant. It also introduces another variant, *one-d-redupd*, which uses the same redundant computation technique for the one-dimensional distribution. The problem with this variant is that the redundant computation, proportional to the full non-distributed dimension, quickly outstrips the non-redundant computation. In fact, there is no result for 64p because there is not enough data per processor to support it.

Finally 6.6(c) presents performance data for the *medium* data set, showing that the localized variant retains its performance advantage at higher processor counts. The *one-d-sim* curve is interesting because the knee at 256 processors is caused by load imbalance: there is insufficient work when the ocean is divided across processors along one dimension to support more than 256 processors. In fact, it is not even possible to generate a distribution for more than 512 processors in one dimension. The two-dimensional variant however would continue to scale to 1024 processors (in principle; a bug during program loading on the simulator currently prevents it).

6.4 Summary

This chapter has demonstrated the performance benefits of a Dual Address Space implementation of HYCOM. It demonstrated the effectiveness of User-Controllable Coherence in both: 1) removing overheads associated with artifactual communication in a global address space, and 2) permitting communication optimizations that would otherwise be impossible in an implementation that takes full advantage of the global address space. Simulation results indicated that localization of a shared memory variant of HYCOM, which uses a two-dimensional data decomposition, significantly reduces the number of last-level cache misses that cause communication. Furthermore, results demonstrated that the reduction in communication, combined with a communication optimization in which synchronizations are reduced via redundant computation, leads to performance

improvement over even a nearly artifactual communication-free one-dimensional data decomposition.

Chapter 7

Application Two: Fast Multipole Method

This chapter takes a different approach to evaluating the potential performance of the adaptive Fast Multipole Method [28] on a Dual Address Space Architecture platform. Rather than simulating a subset of the Fast Multipole Method using a simulator, we draw conclusions from a comparison of the performance of the *full* method running on two hardware platforms, one distributed address space and the other shared.

The adaptive Fast Multipole Method (FMM) is an example of a class of algorithms that uses hierarchical data structures (i.e., trees) to rapidly approximate the numerical solution of the $O(N^2)$ N-Body problem. The FMM has been shown to approximate the solution with high accuracy in $O(N)$ time, though the constant factor is quite large. As noted in the Introduction, a potential application of the FMM of particular interest to molecular biologists is the computation of long-range interactions in molecular dynamics simulations.

Unfortunately, irregular communication patterns make efficient parallelization of the algorithm difficult, if not impossible, to achieve on distributed memory architectures. While the bulk of the run-time of the algorithm is spent in interaction computations – whose communication requirements, though irregular, can be handled efficiently on distributed memory architectures – the remaining time is spent doing fine-grained tree-traversals which quickly become a bottleneck as processors are added.

The purpose of the experiments described in this chapter was to test the hypothesis that direct access to shared memory enabled by hardware-based cache coherence is a better match for the

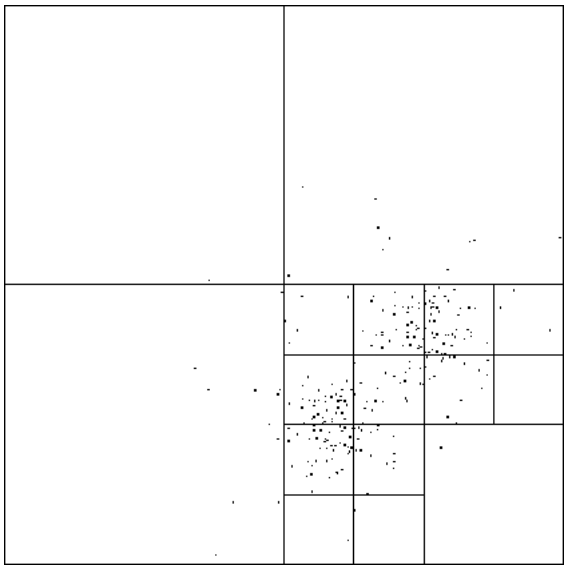


Figure 7.1 Example of an adaptive *quad*tree.

fine-grained communication requirements of the tree-code computation, while the message passing version's gather/scatter approach would be more efficient for the interaction computation.

After providing context for the discussion – with a brief explanation of the Fast Multipole Method algorithm, and our parallelization strategies – we turn our attention to the experiments, and the implications of the results for our hypothesis, and for User-Controllable Coherence.

7.1 Algorithm

While the computational kernel we use is due to Anderson [5], the algorithmic structure of code is the same as the adaptive version of Greengard and Rokhlin's Fast Multipole Method. Like other hierarchical methods, the FMM algorithm is centered around the *octtree* data-structure, which is created as follows: starting with a root box that contains all bodies, recursively subdivide boxes into eight equal-sized sub-boxes until a stopping condition is met. In an *adaptive* octtree, recursion may stop at different levels in different locations of the tree, depending on the distribution of bodies. Figure 7.1 gives an example of an adaptive quadtree, the two-dimensional version of an octtree.

In contrast with the the $O(n \log n)$ Barnes-Hut algorithm [8], which requires each body to traverse the tree individually, the FMM makes use of the observation that when two boxes are "well-separated", the far-field effect of all the bodies in one box on those in the other (and vice-versa) can be approximated as a single interaction between the *boxes*. Such interactions between well-separated boxes occur at all levels of the tree, and the savings in computation enable the FMM to compute far-field forces in $O(n)$ time.

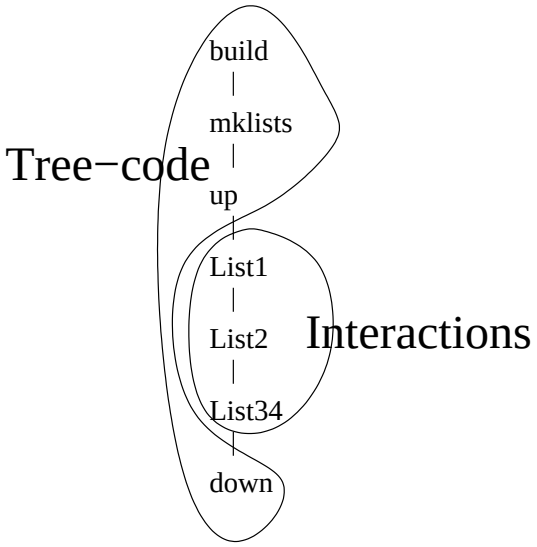
Figure 7.2(a) describes the adaptive version of the algorithm, developed for use with adaptive octtrees. After creating the octtree, the algorithm traverses the tree to compute several types of interaction lists for each box. The algorithm differentiates between three types of interaction computations and therefore defines three lists for each box in the tree (see Figure 7.2(b)): boxes in *List1* are adjacent leaf boxes (and therefore are not sufficiently distant from the source to allow approximation); boxes in *List2* are the same size and well-separated (that is, sufficiently distant from each other to allow approximation); finally, boxes in *List34* are differently sized and well-separated from the perspective of one of the boxes but not the other.

An upward pass over the boxes in the tree establishes the far-field approximations for each box: At the leaves, the approximation for a box is computed from the bodies within; at higher levels, the approximation for a box is computed from approximations for boxes it contains. Then each box computes interactions with all the other boxes on its interaction lists. Finally a downward pass propagates results, collected at each level in the form of a "local-field potential," from box to box, all the way down to the bodies at the leaves.

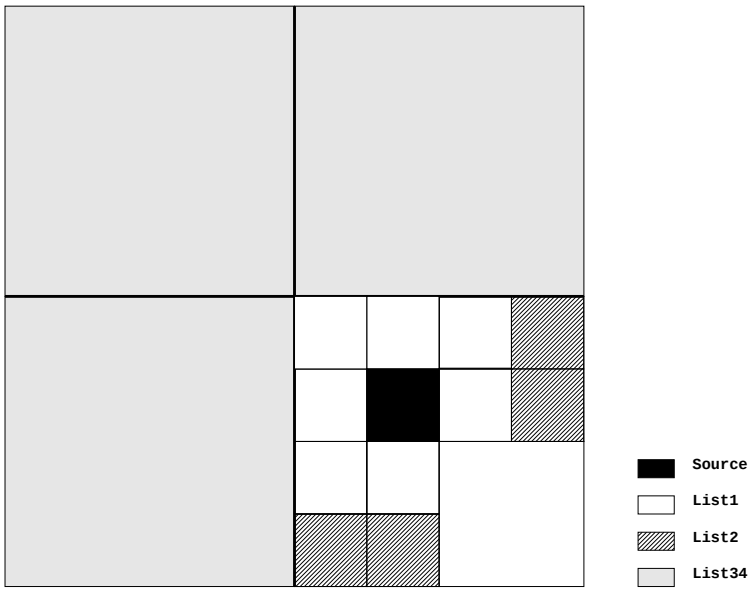
7.2 Parallelization Strategies

7.2.1 Message Passing: MPI

The MPI implementation draws on three sources: the algorithmic structure derives from the FMM benchmark in the SPLASH-2 suite from Stanford [84]; the computational kernel comes from an HPF library implementation of the algorithm [35]; finally, the data-structure derives from Warren and Salmon's "hashed-octtree" parallelization of the Barnes-Hut algorithm [83]. Performance compares favorably to the HPF version due to more efficient communication mechanisms [52].



(a)



(b)

Figure 7.2 (a) The steps of the adaptive FMM algorithm and their division into *Tree-code* and *Interactions* summary categories. (b) Interaction lists for a box in the example quad-tree from Figure 7.1.

```
foreach Node n accessed in Loop l
    if n is NOT in localtree then
        nonlocal.Insert(n)

othersNonlocalIds = Alltoall(nonlocal.ids)
found = localtree.Search(othersNonlocalIds)

myNonlocals = Alltoall(found)
localtree.Insert(myNonlocals)
```

Figure 7.3 Inspection and communication in the MPI implementation.

Non-local Data Access. The primary barrier to overcome in a message passing implementation of the FMM is the irregular access of data imposed by the adaptive octtree data structure. This implementation uses a form of inspector-executor adapted to the hash-table representation of the octtree we borrowed from Warren and Salmon.

Each node in the tree is represented by a unique identifier, derived from the node's location in the tree, which is used as a key for accessing nodes in a hash-table representing the tree. This approach simplifies management of distributed trees in two ways. First, the identifier for a node is the same on *all* processors. Second, integration of non-local nodes into a local tree is simple: data for non-local nodes is simply added to a processor's hash-table.

The pseudo-code in Figure 7.3 demonstrates how the MPI implementation inspects a typical loop-nest for non-local references and communicates required non-local data.

Body Distribution. Another significant problem that the FMM poses for a distributed memory implementation is the initial distribution of bodies across processors such that communication is minimized and load balance is maximized. We have found that sorting bodies according to their position along a Hilbert space-filling curve [68] is an excellent match for this algorithm: since both the Hilbert curve and the octtree recursively divide space in half along each dimension, all bodies in the same leaf of the octtree are contiguous after the sort. We can then partition the sorted sequence of bodies among the processors by assigning each a contiguous range of bodies and, with a little care, ensure that each processor is assigned all bodies in a given subtree of the octtree before the tree is even built.

Parallelization. Below we provide details about the parallelization of each step in the algorithm from Figure 7.2:

build – Given the partitioning strategy described above, we are able to construct octtrees locally, except for a brief communication phase in which processors exchange information about *shared boxes* (boxes at upper levels of the tree whose sub-boxes lie on more than one processor) to ensure that the representation of these boxes is globally consistent.

mklists – The fine-grained nature of the computation in the list-construction algorithm, combined with large communication requirements, can cause this step to be a major bottleneck in the parallelized application if special care is not taken. We have transformed the uniprocessor list construction algorithm into a level-by-level form which allows us to use the inspector-executor strategy noted above, at each level, to gather non-local data before computing lists for that level.

up, down – As noted earlier, we replicate information about shared nodes at the uppermost levels of the tree to all processors and ensure that all nodes in a subtree below any non-shared node are located on the same processor. This partitioning strategy avoids communication in the downward pass and requires only a single communication step in the upward pass when moving from private nodes to the shared parents.

List1, List2, List34 – As in list construction, we communicate non-local data required during the interaction computations using our variation on the inspector-executor technique. To ensure load-balance, we move data for boxes involved in each of the three computation phases into a "weighted-block" distribution immediately prior to that phase. This involves looking in the work-list for each box involved in the computation to determine the amount of work it will do, and then minimally redistributing the boxes such that each processor will have approximately the same total amount of work.

7.2.2 Shared Memory: OpenMP

The shared memory implementation represents a merging of the MPI derivative back into its SPLASH parent, with the aid of OpenMP parallelization pragmas.

Non-local Data Access. The primary difference between the OpenMP and MPI codes is the former's ability to directly access non-local memory. This ability manifests as an advantage in two ways: first it makes the code, particularly the tree-code, much easier to write and understand. As a rough measure of this advantage, the number of lines of C code in the OpenMP implementation is half that of the MPI version.

Secondly, direct access of non-local memory potentially makes the code much more efficient, since there is no need for extra computation and synchronization associated with an inspection phase for irregularly accessed data. On the other hand, there is also potential for inefficiency due to the cache-based nature of hardware coherence: reuse of non-local data that does not stay cached, whether due to conflict, capacity or coherence misses, can severely degrade performance.

Body Distribution. Despite the ability to access global memory, due to locality concerns it is still beneficial for both load balance and communication to "distribute" data appropriately, especially when the architecture exposes local memory to the user. For this reason, we use exactly the same algorithm to partition bodies, sorting them according to their position along a Hilbert space-filling curve, as in the MPI code.

Parallelization. Below we provide details about each algorithmic step:

build – As with the MPI version, the partitioning strategy described above allows most of the octtree construction to be done locally. While merging the trees still requires some synchronization through locks, it is minimal compared to the amount required in the original SPLASH code. However, unlike the MPI version, there are no shared boxes in the OpenMP version, which guarantees a certain amount of contention as processors add their boxes from the top levels of their local trees to the global tree.

mklists, up, down – The fine-grained computation in these steps, coupled with their limited reuse of non-local data, makes them ideally suited to the direct memory access capability of shared memory.

List1, List2, List34 – We are not currently load-balancing interactions. While our use of inspector-executors to communicate non-local data in the MPI interaction computations dictates that all processors do all *List1* computations, at the same time, then all *List2* and then all *List34*, shared memory allows much more flexibility in terms of overlapping computations. Not yet having determined the optimal way to order the List computations for the shared memory version, and also requiring a meaningful way to compare computation times with the MPI version, we currently simply use the same strategy as the MPI version but without the load balancing step.

7.3 Experiments

This section describes experiments with the two versions of the FMM described in the previous section, on an SGI Altix BX2. The Bx2 has 512 Intel Itanium2 processors running at 1.5GHz, each with 6MB of L3 cache. The machine features 4 GB of memory per processor for a total of 2 TB of shared memory.

The experiments study the *weak scaling* of FMM on the Altix: as the number of processors increases, the data size increases proportionately, keeping the number of bodies simulated per processor constant. Perfect scalability would result in constant run-times as the number of processors is increased, evidenced by a flat horizontal line in plots where processors increase on the X axis and times are measured on the Y axis.

We present two categories of results: 1) *summary* results, in which total time divides into "tree-code" versus "interactions", and 2) *detailed* results, in which we further divide the summary data into its constituent parts. Results are the average of per-processor cycle counts and level three cache (L3) miss counts obtained from Itanium2 performance counters. L3 misses are significant since they must be satisfied from memory, potentially non-local, in which case they cause communication.

As discussed earlier, we expect the irregular, extremely fine-grained communication of the tree-code phases to favor hardware-based cache coherence. On the other hand, the more coarse-grained communication requirements of the interaction phases – while still irregular – favor a message-based approach for two reasons:

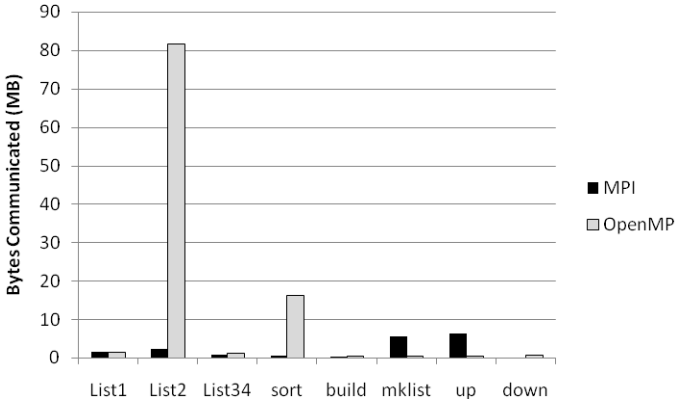


Figure 7.4 Rough comparison of bytes communicated by MPI vs OMP (128 procs).

- For maximal efficiency, non-local data should *stay* where it is after it has been communicated. In cache-based approaches, evicted non-local data must be re-fetched. One of the key observations in [52] is that there is heavy reuse of non-local data in the *List2* interactions, which would seem to translate into a high likelihood for redundant communication as non-local data is evicted from the cache.

- Again, for maximal efficiency, updates to non-local data should not interfere with one-another. In a message-based approach, updates are naturally handled as reductions. On the other hand, in the shared memory paradigm, updates are more naturally expressed using locks to synchronize access to the data, potentially resulting in both lock-contention and unnecessary communication traffic as the updated values propagate from cache to cache.

7.3.1 Communication

As a measure of relative communication efficiency, Figure 7.4 presents approximations of the communication requirements of the two implementations for each algorithmic phase.

For the OpenMP implementation, the approximation is based on L3 cache misses as measured by hardware counters. Since the transfer size for cache misses is the cache-block size, the number of bytes communicated is equal to the number of L3 misses multiplied by the block size. Since not every L3 miss results in communication, this measure overestimates communication. However, if the number of the L3 misses is low we can certainly infer that the number of misses that cause

communication is low; and, as the number of L3 misses grows, the likelihood that the misses require communication, given that some misses do require communication, also grows.

The SGI MPI runtime provides a mechanism for obtaining the number of bytes communicated by the MPI programs. This method also overestimates the amount of data communicated since it does not distinguish, for collective communication calls, between bytes sent to other processors and bytes sent to self.

The data clearly supports the hypothesis that while shared memory more efficiently communicates non-local data in the tree-code steps, message passing is more efficient for interactions. For fine-grained tree-code steps, most notably *build* and *mklists*, the MPI implementation requires significantly more communication than the OpenMP version. On the other hand, the OpenMP implementation of the interaction phase, particularly the *List2* phase, appears to transfer significantly more data than the MPI version.

7.3.2 Performance

Figure 7.5(a) presents summary results comparing performance of the two implementations for the tree-code and interactions. Figure 7.5(b) and (c) show detailed results for each algorithmic step for OpenMP, and Figure 7.5(d) and (e) show the same for MPI.

All the way up to 512 processors, the shared memory version outperforms the MPI version, even on the interaction code. Since the gap between the two is fairly steady as the number of processors grows, we might attribute this to greater indexing efficiency in the shared memory version: recall that we use a hash table to represent the octtree in the MPI version which implies a hash lookup for every index into the tree rather than the simple pointer dereference enabled by globally shared memory.

Based on the detailed results, most of the steep growth for the OpenMP curves in the summary graph can be attributed to two phases: the *List34* interaction phase and the *build* tree-code phase. Both phases make use of locks to synchronize data access.

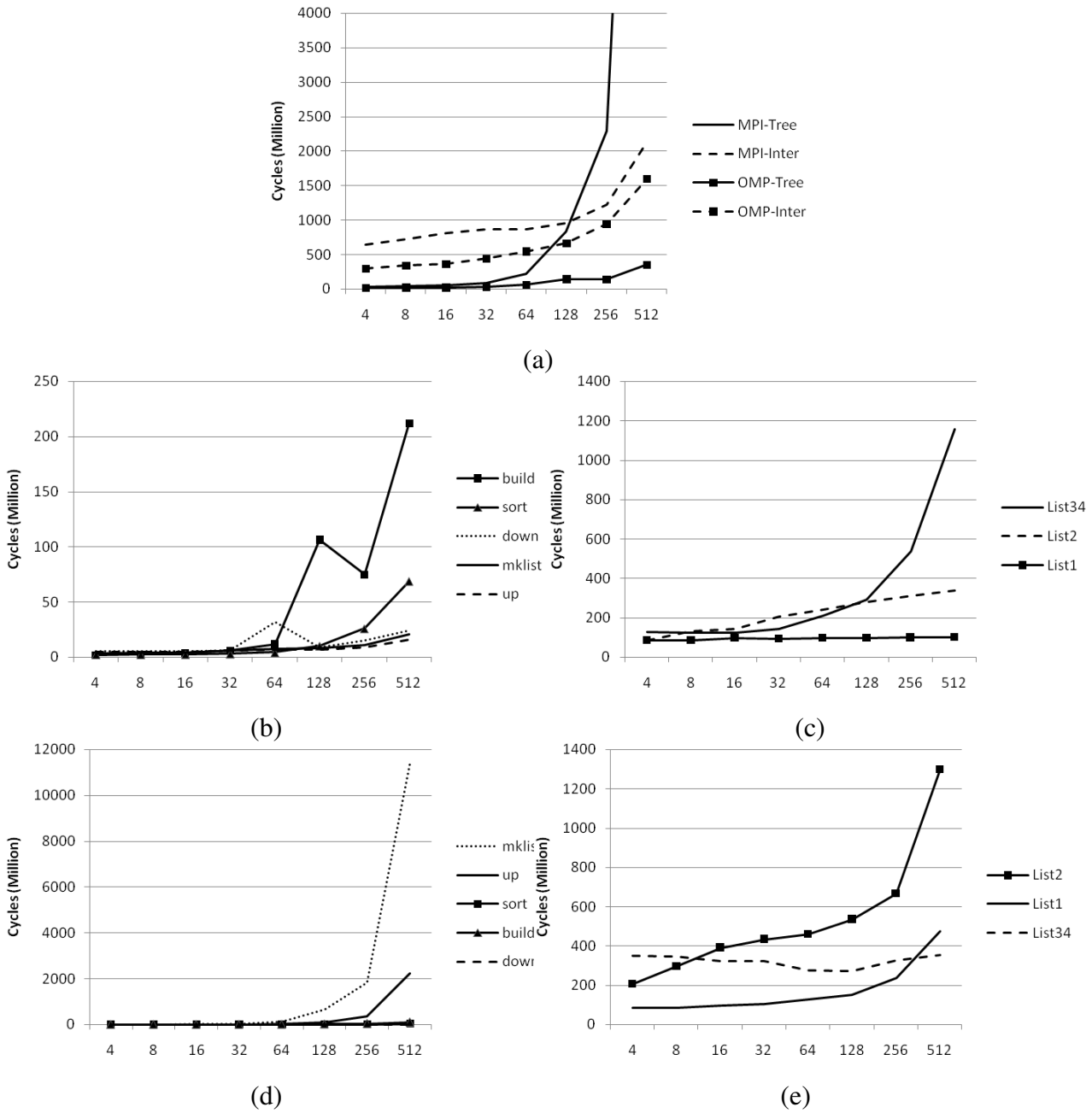


Figure 7.5 Results for 512 processor Altix Bx2 at SGI (3rd iteration): overview (a), and detailed: (b) OpenMP tree-code, (c) OpenMP interactions, (d) MPI tree-code, (e) MPI interactions.

As described in 7.2, we have already done significant work to try to reduce the effect of synchronization in the *build* phase. It appears that we need to find a more structured means of completing the global tree that requires even less synchronization.

Locks are required in the *List34* phase due to potential non-local writes: recall that List34 is actually two symmetric lists, such that if Box *A* is on Box *B*'s List3 then Box *B* is on *A*'s List4. To simplify the list construction phase, only one end of this dual transaction is recorded in a list (List34) and both computations are taken care of at the same time in the *List34* interaction phase. As a result, the computation may result in writes to non-local data since Box *A* may be located on a different processor than Box *B*.

The MPI version takes advantage of the fact that each write is only an update to a value which is not actually used until the phase is over, and turns a collection of non-local writes into a single post-computation reduction.

Strikingly, despite the OpenMP implementations problems with *List34* locks, *and* the MPI version's apparently substantial advantage in terms of communication requirements for the *List2* interaction computations, MPI interaction performance consistently lags OpenMP. The detailed results point to the *List2* interactions as the primary source of the problem.

As a potential explanation, recall that every processor gathers all the non-local requests of every other processor to determine whether it owns the data requested. This all-to-all approach clearly does not take full advantage of data locality inherent in the algorithm.

Another factor may be that there is some communication involved in redistributing data in order to load balance the interaction computations in the MPI version. As noted earlier, we are not currently load balancing the OpenMP interactions.

7.4 Implications for the Dual Address Space Architecture

With its mixture of fine-grained and coarse-grained communication, the Fast Multipole Method is a particularly good target for a Dual Address Space Architecture implementation. Experimental data presented in this chapter indicates the following for such an implementation:

1. The treecode phases *require* a global, shared address space. Furthermore, the shared address space implementation must be hardware-based; scalability is already an issue at 512 processors even on the Altix. The tree building phase in the current implementation is a particular problem, likely due to locking, and different approaches will need to be investigated for scaling to higher processor counts. One possibility is the replacement of locks with transactional memory transactions.

2. The interaction phases can benefit significantly from local memory. Though the shared address space implementation outperformed the distributed memory implementation in the experiments, there is still significant room for improvement in two areas. First, non-local cache misses can be substantially reduced in the List2 interactions phase. Second, locking in the List34 interactions phase can be removed through the use of reductions.

3. There remain several implementation issues to work out. For example, some data is used in both phases. Also, since particles move from timestep to timestep, repartitioning of localized data between steps will be necessary.

7.5 Summary

This chapter described experiments with two implementations of the Fast Multipole Method application – one using the shared memory programming model (OpenMP), the other the message passing model (MPI) – on two different memory architectures. The experiments demonstrated that, as expected, the tree-code phases of the algorithm are much better suited to the shared memory programming paradigm, both from a communication and an actual performance standpoint. The OpenMP version also outperforms the message passing version for the coarse-grained interaction phases despite significantly unfavorable communication requirements. Overheads in the inspector-executor paradigm, or at least in our particular implementation of it, appear to outweigh the benefits of reduced communication.

The experiments expose weaknesses in both models that can be addressed with a Dual Address Space implementation of the algorithm: 1) The treecode phases *require* a global, shared address

space. 2) The interaction phases can benefit significantly from local memory: First, non-local cache misses can be substantially reduced in the List2 interactions phase. Second, locking in the List34 interactions phase can be removed through the use of reductions.

Chapter 8

Conclusions

Chapter 1 introduced the three-pronged thesis of this dissertation:

1. User-Controllable Coherence is an efficient mechanism for enabling increased user and compiler control over data consistency and data locality.

2. The Dual Address Space provides a high-level interface for programmers to conceptualize and take advantage of the mechanism.

3. Together they provide users and compilers an efficient means, particularly appropriate to the structure of scientific applications, of keeping data consistent and local, enabling improved performance over current global address space implementations.

The remainder of the dissertation evaluated the thesis in two parts. The first part evaluated the first two clauses of the thesis by construction: Chapter 3 described the architecture and programming model, while Chapter 4 described a full-scale directory-based implementation of the Dual Address Space Architecture.

The second part of the dissertation evaluated the third clause of the thesis through a performance evaluation of Dual Address Space implementations two applications that currently underperform on distributed address space hardware. Chapter 6 introduced HYCOM, an ocean modeling application which makes use of a static though slightly irregular data decomposition, complicating the nearest-neighbor communication due to stencil computations. Chapter 7 introduced the Fast

Multipole Method, which features highly irregular, dynamic computation partitioning, data decomposition and communication that significantly impair implementation in a distributed address space.

Detailed highlights of the dissertation:

- **Chapter 3 described the Dual Address Space, the programmer's interface to the Dual Address Space Architecture.** It began with a thread's-eye view of the Dual Address Space. In size and structure, the local address space is a logical duplicate of the global address space, but it may contain different data values. Data is only valid in either the local or global address space at a given time. The chapter then introduced new instructions for accessing data in the local address space, ld_L and st_L, and presented the precise semantics of the local and global instructions.

  Next, the chapter provided a formal definition of the memory model, in terms of well-established programmer-centric synchronization models. It demonstrated that as long as software follows established guidelines (indicating synchronization operations and making them global; ensuring no races exist between data operations; ensuring localized data is globalized appropriately) it will be guaranteed sequentially consistent execution. Finally, the chapter described the programming model, explaining how software uses the Dual Address Space to explicitly reduce artifactual communication.

- **Chapter 4 described a full-scale implementation of the Dual Address Space Architecture, an extension of the SGI Altix design, itself based on the SGI Origin.** The chapter began by describing the layout of a *node*, the basic building block in all the aforementioned systems. Modifications are largely limited to the coherence protocol implementation in the processor and directory interfaces, along with the addition of local memory to the memory interface.

  The chapter continued with a detailed description of a directory-based implementation of the User-Controllable Coherence cache coherence protocol, initially assuming single-word cache blocks, and then describing in detail the ramifications of multi-word cache blocks. The

chapter then described support for the consistency model introduced in Chapter 3. Finally, the chapter closed with a discussion of the implementation of local memory.

- **Chapter 5 described the implementation of the cycle-accurate simulator, modeling a 1024 processor SGI Altix 4700 system.** The simulator, used in the evaluations of Chapters 1 and 6, consists of two parts: a reference generating front-end and the memory system simulator back-end. The chapter described the parallelization of both parts, resulting in a fully parallelized system simulator, demonstrated capable of accurately simulating 512 threads running a real, full-scale, application across 128 host processors. Finally, the chapter provided validation and performance data from runs of the simulator on a full-scale application, noting several opportunities for improvement in both areas.

- **Chapter 6 demonstrated the performance benefits of a Dual Address Space implementation of HYCOM.** It demonstrated the effectiveness of User-Controllable Coherence in both: 1) removing overheads associated with artifactual communication in a global address space, and 2) permitting communication optimizations that would otherwise be impossible in an implementation that takes full advantage of the global address space. Simulation results indicated that localization of a shared memory variant of HYCOM, which uses a two-dimensional data decomposition, significantly reduces the number of last-level cache misses that cause communication. Furthermore, results demonstrated that the reduction in communication, combined with a communication optimization in which synchronizations are reduced via redundant computation, leads to performance improvement over even a nearly artifactual communication-free one-dimensional data decomposition.

- **Finally, Chapter 7 described a different approach to the evaluation of a Dual Address Space version of the Fast Multipole Method: experiments with two implementations of the Fast Multipole Method application – one using the shared memory programming model (OpenMP), the other the message passing model (MPI) – on two different memory architectures.** The experiments demonstrated that, as expected, the tree-code phases of the algorithm are much better suited to the shared memory programming paradigm, both

from a communication and an actual performance standpoint. The OpenMP version also outperforms the message passing version for the coarse-grained interaction phases despite significantly unfavorable communication requirements. Overheads in the inspector-executor paradigm, or at least in our particular implementation of it, appear to outweigh the benefits of reduced communication.

The experiments exposed weaknesses in both models that can be addressed with a Dual Address Space implementation of the algorithm: 1) The treecode phases *require* a hardware supported shared address space. 2) The interaction phases can benefit significantly from local memory: First, non-local cache misses can be substantially reduced in the List2 interactions phase. Second, locking in the List34 interactions phase can be removed through the use of reductions.

Future Work The complete fruition of this work would be a hardware implementation of the Dual Address Space Architecture, along with a Dual Address Space implementation of a molecular dynamics code that uses the Fast Multipole Method to solve long-range force computations. In addition to that long-term goal, there are many other possible directions for future work, including the following questions:

- What impact could a Dual Address Space Architecture have on the elusive goal of automatic parallelization?
- Could a Dual Address Space Architecture, on a smaller scale, be useful in addressing problems with coherence traffic on the multicore building blocks of distributed memory platforms?
- Is local memory with selective globalization the way to implement message passing on shared address space hardware?
- Could User-Controllable Coherence be the basis for a hardware implementation of Transactional Memory (ala Speculative Lock Elision [64])?

Finally, independent of the thesis, the pursuit of this dissertation has resulted in an insight that could be a particularly fruitful area for future work: there is remarkably little to distinguish applications that succeed on the largest-scale platforms, such as S3D, from others like HYCOM that are far less successful. Understanding the difference may be the first step towards fixing the problem.

LIST OF REFERENCES

[1] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, et al. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Baltimore, MD, November 2002.

[2] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[3] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, WA, June 1990.

[4] SV Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, December 1996.

[5] C.R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13(4):923–947, July 1992.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[7] V. Aslot and R. Eigenmann. Performance characteristics of the spec omp2001 benchmarks. In *EWOMP '01: Proceedings of the Third European Workshop on OpenMP*, Barcelona, Spain, September 2001.

[8] J. Barnes and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[9] S. Basu and J. Torrellas. Enhancing memory use in simple coma: Multiplexed simple coma. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 152–161, Las Vegas, NV, February 1998.

[10] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending openmp for numa machines. *Scientific Programming*, 8(3):163–181, 2000.

[11] R. Bleck. An oceanic general circulation model framed in hybrid isopycnic-Cartesian coordinates. *Ocean Modeling*, 4(1):55–88, 2002.

[12] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, Tucson, AZ, June 2008.

[13] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems (TOCS)*, 13(3):205–243, August 1995.

[14] Barbara M. Chapman, Amit Patil, and Achal Prabhakar. Performance oriented programming for numa architectures. In *WOMPAT '01: Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 137–154, West Lafayette, IN, July 2001.

[15] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, March 2007.

[16] J.A. Crawford and C.M. Mobarry. Hrunting: a distributed shared memory system for the beowulf parallel workstation. In *"Proceedings of the 1998 IEEE Aerospace Conference"*, pages 337–344 vol.4, Aspen, CO, March 1998.

[17] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, San Francisco, CA, 1999.

[18] James Demmel. Applications of parallel computers (class notes). `http://www.cs.berkeley.edu/~demmel/cs267`, 1996.

[19] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, Göteborg, Sweden, June 2001.

[20] E. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[21] Michel Dubois, Jin Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 197–206, Albuquerque, NM, November 1991.

[22] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, Cambridge, MA, October 1996.

[23] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing '94: Proceedings of the 1994 Conference on Supercomputing*, pages 380–389, Washington, D.C., November 1994.

[24] Babak Falsafi and David A. Wood. Reactive numa: a design for unifying s-coma and cc-numa. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, Denver, CO, June 1997.

[25] Matthew I. Frank and Mary K. Vernon. A hybrid shared memory/message passing parallel machine. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 232–236, Syracuse, NY, August 1993.

[26] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, June 1990.

[27] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[28] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, December 1987.

[29] Erik Hagersten and Michael Koster. Wildfire: A scalable path for smps. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 172, Orlando, FL, January 1999.

[30] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 107–116, Vancouver, Canada, June 2000.

[31] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Boston, MA, October 2004.

[32] Tim Harris, Adrian Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May 2007.

[33] E.R. Hawkes, R. Sankaran, J.C. Sutherland, and J.H. Chen. Direct numerical simulation of turbulent combustion: Fundamental insights towards predictive models. *Journal of Physics: Conference Series*, 16(1):65–79, June 2005.

[34] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: software and hardware for scalable multiprocessor. In *ASPLOS-V: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, Boston, MA, October 1992.

[35] Y. Charlie Hu, S. Lennart Johnsson, and Shang-Hua Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[36] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. Coherence decoupling: making use of incoherence. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–106, Boston, MA, October 2004.

[37] Intel. *A Formal Specification of Intel Itanium Processor Family Memory Ordering, No. 251429-001*, October 2002.

[38] T. Joe and J. L. Hennessy. Evaluating the memory overhead required for coma architectures. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 82–93, Chicago, IL, April 1994.

[39] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, Queensland, Australia, May 1992.

[40] D.J. Kerbyson and P.W. Jones. A Performance Model of the Parallel Ocean Program. *International Journal of High Performance Computing Applications*, 19(3):261, August 2005.

[41] David A. Koufaty, Xiangfeng Chen, David K. Poulsen, and Josep Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1250–1264, December 1996.

[42] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. In *PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, San Diego, CA, July 1993.

[43] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[44] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[45] James R. Larus, Brad Richards, and Guhan Viswanathan. Lcm: memory system support for parallel language implementation. In *ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 208–218, San Jose, CA, October 1994.

[46] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, CO, June 1997.

[47] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, March 1992.

[48] Los Alamos National Laboratory. A hybrid coordinate parallel ocean program (hypop). `http://climate.lanl.gov/Models/HYPOP/index.shtml`, 2008.

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, Chicago, IL, June 2005.

[50] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, Long Beach, California, January 2005.

[51] Collin McCurdy and Charles Fischer. User-Controllable Coherence for High Performance Shared Memory Multiprocessors. In *PPoPP '03: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, San Diego, CA, June 2003.

[52] Collin McCurdy and John Mellor-Crummey. An evaluation of computing paradigms for n-body simulations on distributed memory architectures. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, Atlanta, GA, May 1999.

[53] Cameron McNairy and Don Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, March 2003.

[54] Saleem Mohideen and Manish Ahluwalia. Methods and apparatus for a dual address space operating system. United States Patent, 2006.

[55] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Mark D. Hill, David A. Wood, Steven Huss-Lederman, and James R. Larus. Wisconsin wind tunnel ii: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October 2000.

[56] National Aeronautics and Space Administration. NAS projects: Columbia. `http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html`, 2008.

[57] Gil Neiger. A taxonomy of memory ordering models. `http://www.cs.utah.edu/mpv/papers/neiger/fmcad2001.pdf`, October 2001. Invited Talk for Tutorial and Workshop on Formal Specification and Verification Methods for Shared Memory Systems.

[58] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. Is data distribution necessary in openmp? In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 47, Dallas, TX, November 2000.

[59] D.A. Pearlman, D.A. Case, J.W. Caldwell, W.S. Ross, T.E. Cheatham, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman. AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Computer Physics Communications*, 91(1-3):1–41, September 1995.

[60] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, December 2005.

[61] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport clocks: verifying a directory cache-coherence protocol. In *SPAA '98: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, Puerto Vallarta, Mexico, June 1998.

[62] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[63] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *Parallel and Distributed Technology: Systems and Applications, IEEE*, 4(2):63–71, Summer 1996.

[64] R. Rajwar and J.R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34: Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, Austin, TX, December 2001.

[65] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: user-level shared memory. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

[66] S.K. Reinhardt. *Mechanisms for Distributed Shared Memory*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, 1996.

[67] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The ksr1: experimentation and modeling of poststore. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 74–85, Santa Clara, CA, May 1993.

[68] Hans Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[69] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 276, Raleigh, NC, January 1995.

[70] F. Schon. The logo structuring scheme for distributed shared memory systems. In *"Proceedings of the First International Conference on Massively Parallel Computing Systems"*, pages 408–412, Ischia, Italy, May 1994.

[71] Steve Scott, Dennis Abts, John Kim, and William J. Dally. The blackwidow high-radix clos network. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 16–28, Boston, MA, June 2006.

[72] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, Cambridge, MA, October 1996.

[73] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[74] A.G. Shet, W.R. Elwasif, R.J. Harrison, and D.E. Bernholdt. Programmability of the hpcs languages: A case study with a quantum chemistry kernel. In *"IPDPS 2008: Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing"*, pages 1–8, Miami, FL, April 2008.

[75] Silicon Graphics Inc. Sgi products: High performance computing: Altix 4700. `http://www.sgi.com/products/servers/altix/4000`, 2008.

[76] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60(1-4):38–61, November 1992.

[77] D. Sorin, M. Plakal, A. Condon, M. Hill, M. Martin, and D. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[78] Sun Microsystems Inc. *The SPARC Architecture Manual, No. 800-199-12, Version 8*, January 1991.

[79] A. Wallcraft, H. Hurlburt, EJ Metzger, E. Chassignet, J. Cummings, and OM Smedstad. Global ocean prediction using hycom. In *HPCMP-UGC '07: Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*, pages 259–262, Pittsburgh, PA, June 2007.

[80] A.J. Wallcraft. SPMD OpenMP versus MPI for ocean models. *Concurrency - Practice and Experience*, 12(12):1155–1164, October 2000.

[81] Alan Wallcraft. personal communication, 2006.

[82] D.H.D. Warren and S. Haridi. The data diffusion machine–a scalable shared virtual memory multiprocessor. In *ICOT '88: Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 943–952, November 1988.

[83] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21, Portland, OR, November 1993.

[84] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[85] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–229, San Jose, CA, October 1994.

[86] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 75–88, Seattle, WA, October 1996.