Supporting Software Distributed Shared Memory with an Optimizing Compiler

Tatsushi Inagaki* Junpei Niwa Takashi Matsumoto Kei Hiraki
Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113 Japan
{ inagaki, niwa, tm, hiraki }@is.s.u-tokyo.ac.jp

Abstract

To execute a shared memory program efficiently, we have to manage memory consistency with low overheads, and have to utilize communication bandwidth of the platform as much as possible. A software distributed shared memory (DSM) can solve these problems via proper support by an optimizing compiler. The optimizing compiler can detect shared write operations, using interprocedural points-to analysis. It also coalesces shared write commitments onto contiguous regions, and removes redundant write commitments, using interprocedural redundancy elimination. A page-based target software DSM system can utilize communication bandwidth, owing to coalescing optimization. We have implemented the above optimizing compiler and a runtime software DSM on AP1000+. We have obtained a high speed-up ratio with the SPLASH-2 benchmark suite. The result shows that using an optimizing compiler to assist a software DSM is a promising approach to obtain a good performance. It also shows that the appropriate protocol selection at a write commitment is an effective optimization.

1. Introduction

Applications using software distributed shared memory (DSM) can run without troubles of unnecessary memory copy and address translation which happen with the inspector/executor mechanism[22]. Most of existing software DSM systems are designed on the assumption of using sequential compilers[23, 20, 19]. An executable object made by a sequential compiler only issues a shared memory access as the ordinary memory access(load/store). To utilize bandwidth, a runtime system has to buffer the remote memory access. There is another approach where a programmer can specify optimal granularity, protocol, and association between synchronization and shared data[3, 30]. However, with this approach, existing shared memory applications require rewriting.

Our idea is that an optimizing compiler directly analyses shared memory source programs, and optimizes communication and consistency management for software DSM execution[28]. Our target is a page-based software DSM, asymmetric distributed shared memory (ADSM)[26, 25]. ADSM uses a virtual memory mechanism for shared read, and uses explicit user-level consistency management code sequences for shared write. This enables static optimization of shared write operations. Static optimizing information about them can reduce the overhead of the runtime system. Shasta[29] is another software DSM system assuming optimizing compiler support. Since Shasta compiler analyzes objects generated by sequential compilers, it only performs limited local optimizations. Our compiler analyzes a source program directly. Therefore, it performs array data-flow analysis interprocedurally.

Here we have to solve the following three problems in order to show that our approach is effective. First, the compiler must perform sufficient optimization in reasonable compilation time. We have applied interprocedural points-to analysis[14, 31], and implemented interprocedural write set calculation, to detect and optimize shared write operations. We have found out that the above powerful analysis is done in reasonable time. Second, the runtime system also must work efficiently. We had been using a history-based runtime system of lazy release consistency[28]. But when the compiler can not optimize, the system introduces a large runtime overhead and causes the growth of synchronization costs. Therefore, we have implemented a new page-based runtime system with delayed invalidate release consistency (DIRC) model[12] to overcome these problems. We have made sure that the new system is more efficient than the history-based runtime system. Third, we have to provide an interface such that users can give information which the compiler can not extract statically. Memory access patterns of irregular applications depend on input parameters. It is

*Presently with Tokyo Research Laboratory, IBM Japan, Ltd.

difficult for a compiler to optimize copy management protocols statically. We have examined the effect of manual protocol selection on the bottleneck shared write operations of the program.

We have evaluated the performances with the SPLASH-2 benchmark suite[32]. SPLASH-2 is not only the most frequently used benchmark to evaluate shared memory systems, but also a benchmark suite with in detailed algorithmic information about each program. We have manually optimized shared write protocols using these descriptions. We do not consider SPLASH-2 as "dusty deck". Our target is to investigate what information from a user or a compiler is required for the efficient execution about shared memory programs on software DSM.

Section 2 describes a process of compilation and optimization. Section 3 describes the implementation of the runtime software DSM. Section 4 describes performance evaluation with SPLASH-2. Section 5 describes related work about a combination of optimizing compiler and software DSM. Section 6 gives a summary.

2. Compilation Process

Figure 1 describes the overall compilation process. The input is a shared memory program written in C extended with PARMACS[4]. PARMACS provides the primitives for task creation, shared memory allocation, and synchronization (barrier, lock, and pause). The consistency of shared memory follows lazy release consistency (LRC) model[20]. Our compiler inserts consistency management code sequences for software DSM into a given shared memory program. The backend sequential compiler compiles the instrumented source program and links it with a runtime library.

To inform the runtime system that a write happened onto a contiguous shared block, we use a pair found by the initial address and the size of of the block. We call this pair a *(shared) write commitment*. Besides the start address and the size, a write commitment also requires the written contents of the block. Therefore, we place a write commitment after the corresponding shared write operations. The single write commitment can represent a lot of shared writes onto a large contiguous region. When there are succeeding write commitments with the same parameters, we can eliminate them but the last one.

2.1. Shared Write Detection

The goal of our optimizing compiler is to insert valid write commitments and to decrease the number of write commitments as much as possible. First we have to enumerate all shared memory access in a given shared memory program. Since the input program is written in C, a shared address may be contained in a pointer variable and may be passed across procedure calls.

We have applied interprocedural points-to analysis[14, 31] to shared write detection. Interprocedural points-to analysis calculates symbolic locations where variables may point to. Variables and heap locations are represented with a *location set*, a tuple of a symbolic base address, an offset, and a stride. The compiler interprocedurally calculates points-to relations among location sets using a depth-first traversal of the call graph. We track the return values of shared memory allocation primitive (G_MALLOC). We insert a write commitment after a write operation using shared address values.

We adopted interprocedural points-to analysis because of the following merits:

- succeeding optimization passes can perform code motion using pointer information, and
- precise shared pointer information can decrease the costs of the redundancy elimination pass.

Points-to analysis represents all variables as memory locations. This is a conservative assumption in C. When an input program contains unions or type-castings, they may generate false alias information, which takes many iterations to converge. We assume that an input program is type-safe about pointer values, that is pointer values are not conveyed through non-pointer locations. In points-to analysis, we only record pointer assignments into pointer type locations. This assumption prevents generating false alias relations in a program with complex structures.

2.2. Redundancy Elimination

In release consistency model, a shared write is not transmitted to other nodes until the node which had issued the shared write reaches a synchronization. Therefore, it is valid that we place a write commitment everywhere from the corresponding shared write to the first synchronization thereafter. We use this flexibility to remove redundant write commitments.

For example, let us look the following code sequence from LU:

```
a[ii][jj] = ((double) lrand48())/MAXRAND;
if (i == j)
  a[ii][jj] *= 10;
```

Suppose that `a[ii][jj]` is shared. It is valid that we insert write commitments after both assignments. However, if we delay the first write commitment after the conditional, the write commitment within the conditional is redundant. When we denote a write commitment as `WC`,

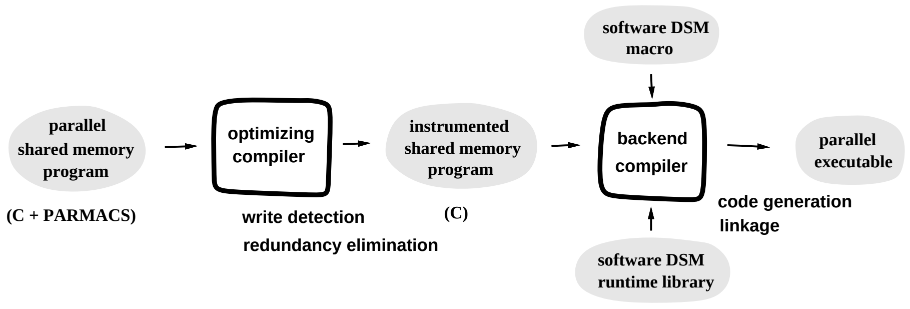


Figure 1. Overall compilation process

```
a[ii][jj] = ((double) lrand48())/MAXRAND;
if (i == j)
  a[ii][jj] *= 10;
WC (&a[ii][jj], 1);
```

Note that this holds if the order between the assignment and the conditional is opposite.

This optimization can be formalized as redundancy elimination[8, 27] of write commitment. Here we represent a statement in a procedure as i. We can consider that i is a node of a control flow graph (CFG) of the procedure. For simplicity, we fix a write commitment with the same address and the same size. From the result of points-to analysis, we obtain the following logical constants about each statement i:

$\mathrm{COMP}(i)$	the statement i issues the shared write
$\mathrm{TRANS}(i)$	the statement i propagates information about the shared write

$\mathrm{TRANS}(i)$ is false when the statement i is a synchronization primitive or the statement i modifies the parameters of the write commitment. We can calculate the following logical dataflow variables from these constants:

Availability In all paths which precede the statement i, the shared write is issued

Anticipatability In all paths which succeed the statement i, the shared write is issued

To minimize the number of write commitments, we place write commitments only where,

- the shared write is available,
- the shared write is not available in one of the succeeding paths, and
- the shared write is not anticipatable.

We represent availability before and after execution of the statement i as $\mathrm{AVIN}(i)$ and $\mathrm{AVOUT}(i)$. Similarly, we represent anticipatability as $\mathrm{ANTIN}(i)$ and $\mathrm{ANTOUT}(i)$. $\mathrm{INSERT}(i)$ is a variable which means we actually place the write commitment after the statement i. Variables are calculated under dataflow equations in Figure 2. Primitives $\mathrm{pred}(i)$ and $\mathrm{succ}(i)$ represent sets of statements preceding and succeeding the statement i.

$$
\begin{aligned}
\mathrm{AVIN}(i) &= \prod_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{AVIN}(i) \\
\mathrm{ANTOUT}(i) &= \prod_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{ANTOUT}(i) \\
\mathrm{INSERT}(i) &= \mathrm{AVOUT}(i) \cdot \neg \left(\prod_{s \in \mathrm{succ}(i)} \mathrm{AVOUT}(s) \right) \\
&\quad \cdot \neg \mathrm{ANTOUT}(i)
\end{aligned}
$$

Figure 2. Dataflow equation to remove redundant write commitments

To compute interprocedurally, we reflect AVOUT at the exit of the callee procedure to the COMP at the call site of the caller procedure. When the availability of the callee can not be propagated to the caller, we insert write commitments at the exit of the callee. We call a procedure which is called recursively or called through function pointers, as *open* procedure[7]. An open procedure does not inform availability to the call sites. Therefore, we can consider the call graph is acyclic. The compiler simply calculates interprocedural availability with bottom-up traversal of the call graph. If we want more precise elimination, the compiler also can traverse the call graph in depth-first manner, which is not implemented yet.

2.3. Merging Multiple Write Commitments

A write commitment can handle shared write operations onto a contiguous region. For example, let us look the following code sequence in LU:

```
for (i = 0; i<n; i++)
  a[i] += alpha * b[i];
```

Suppose that `a` is a shared pointer. Instead of inserting a write commitment into the innermost loop, we can generate:

```
for (i = 0; i<n; i++)
  a[i] += alpha * b[i];
WC (a, n);
```

This code generation has two merits. First, a consistency management overhead is reduced because the write commitment is hoisted out from the loop. Second, the runtime system can utilize the size information for message vectorization.

To combine multiple write commitments, it is convenient to represent a sequence of write commitments as *(shared) write set*. A write set $W = (f, s, C)$ is a tuple, such that f is a start address of a write commitment, s is a size, and C is a set of inequalities which generate write commitments. Inequalities C represent induction variables of enclosing loops around the write commitment.

A dataflow variable takes a set of write sets. The logical operations in the above dataflow equations are considered as set operations. Just after points-to analysis, each write set includes only one write commitment, i.e., $s = 1, C = \emptyset$. We use interval analysis[9, 5] to calculate dataflow equations. In interval analysis, CFG is represented hierarchically with interval (i.e. loop) structures. When a summary of interval is propagated outward, inequalities which represent induction variables are added to C.

We describe optimizing methods to combine multiple write commitments using write set.

Coalescing This is applicable when write commitments onto contiguous locations are issued in a loop. Suppose a write set $W = (f(i), s, C(i))$ and the induction variable i has a increment value c. If $f(i{+}c) - f(i) = s$, we can replace i with the initial value of i, multiply s by the number of iterations, and remove inequalities about i from C. For the above example,

$$W = (\&a[i], 1, \{0 \le i < n\}) \rightarrow W' = (a, n, \emptyset).$$

Coalescing is applicable when the index variable is only *continuous*. For example, let us look the following code sequence in Radix:

```
for (i=key_start; i<key_stop; i++) {
  this_key = key_from[i] & bb;
  this_key = this_key >> shiftnum;
  tmp = rank_ff_mynum[this_key];
  key_to[tmp] = key_from[i];
  rank_ff_mynum[this_key]++;
} /* i */
```

Suppose `key_to` points to shared addresses. Variables `rank_ff_mynum[this_key]` are incremented by one when `key_to[tmp]` is written. Therefore, we can coalesce write commitments using initial values and final values of `rank_ff_mynum[this_key]`.

Fusion We can also merge write commitments originating in different statements in the program. We represent this operation as a binary operator "$\circ$". For example, let us look the following code sequence in FFT:

```
for (i = 0; i<n1; i++) {
  x[2*i] /= N; x[2*i+1] /= N;
}
```

Suppose `x` points to shared address,

$$W = (\&x[2*i], 1, \emptyset), \qquad W' = (\&x[2*i+1], 1, \emptyset),$$

$$W \circ W' \rightarrow W'' = (\&x[2*i], 2, \emptyset)$$

Redundant index elimination When the start address of a write commitment is a constant, we can delegate the write commitment with the maximal size. If we can detect the maximum, this index variable is redundant. We can eliminate redundant indexes using Fourier-Motzkin elimination[11]. Fourier-Motzkin elimination is also applicable to nonlinear but monotonous expressions. For example, in the following write set in FFT,

$$W = (x, 2 * 2^q * (N/2^q), \{1 \le q \le M\}),$$

we can eliminate q, using monotonicity of 2^q and $Q * (N/Q)$, and obtain

$$W \rightarrow W' = (x, 4 * (N/2), \emptyset).$$

The names coalescing and fusion come from the similarity to loop transformations. When a dimension of inequalities in C is decreased, the dimension of generated loop of write commitments is decreased.

When the summary of an interval is computed, we apply coalescing and redundant index elimination to write sets. Fusion is applied to the computation of set union in dataflow equations. When a write set is propagated outward from a loop without coalescing or index elimination, we add inequalities about loop indexes into C. This corresponds to *fission*(or distribution) in loop transformations. Fission does not reduce the number of issued write commitments but improves memory access locality. Along dataflow computation in interval analysis, the compiler repeatedly applies Fourier-Motzkin elimination to the expressions in innermost loops. We use *memorization*[1] technique which stores and reuses the results computed before.

3. Target Software DSM

We implemented a runtime library of ADSM on a Fujitsu AP1000+. The AP1000+ has dedicated hardware which executes remote block transfer operation (put/get interface[18]). We assume that point-to-point message order is preserved.

Formerly, we had been using a history-based runtime system of lazy release consistency[28]. This implementation stores write commitments as a write history. When a synchronization primitive is issued, the page contents are written back to the page-home. This corresponds to a software emulation of automatic update release consistency (AURC)[19]. Diff based implementation compares whole page contents[20]. History based implementation can avoid this when the compiler successfully eliminates and coalesces the write commitments. However, the following two problems exist:

- When the compiler can not optimize, history management introduces a large runtime overhead.
- We handle logical timestamps between each synchronization like LRC and AURC. Frequent synchronization causes long synchronization messages and the growth of synchronization costs.

This time, we have implemented a new page-based runtime system. The basic design is similar to that of SoftFLASH[15] with delayed invalidate release consistency (DIRC) model[12]. We use a write commitment for message vectorization.

3.1. Basic Design

Shared memory is managed by pages. Each page has a page-home node and the user can specify which it is. Each node manages the following bit tables with the size of the number of shared pages.

Valid bit table indicates that the page contents are valid.

Dirty bit table indicates that the node has written into the page with the current synchronization interval[20].

Each node also manages the following bit table with the size of the number of nodes.

Acknowledge table indicates that the node had written into the page of the corresponding page-home node.

Synchronization tags of locks and pauses are handled by specified synchronization-home(i.e., lock-home or pause-home) nodes. Each lock and pause has its own dirty bit table. We describe the behaviors of the runtime system for each primitive.

When a write commitment is issued, the written memory contents are sent to the page-home node with a put operation. The size parameter of the write commitment corresponds to the length of the block transfer. The page-home node is recorded in the acknowledge table.

At an acquire operation, the node receives the dirty bit table from the lock-home processor. The obtained dirty bit table is applied to the valid bit table. The size of synchronization messages are limited by the dirty bit table size because the time information is not utilized at synchronization. However, if a node acquires the same lock again, a page may be invalidated even when the page is not written between lock acquisitions.

In a release operation, the node sends the nodes recorded in the acknowledge table and confirms that all sent messages have arrived to the destinations. Then, the node sends the dirty bit table to the lock-home node.

When a page fault occurs, the page contents are copied from the page-home by a get operation.

At a barrier operation, the following steps are executed:

1. Each node confirms whether all the preceding page-home updates are completed.
2. All nodes send their own dirt bit tables to the master node.
3. The master merges the sent dirty bit tables and broadcasts the merged one.
4. All nodes invalidate their copies using the sent dirty bit table.
5. Each node clears its dirty bit table and the dirty bit table of synchronization tags which it manages.

Communications at page faults and write commitments are handled asynchronously. Acquire and release operations are serialized by sending explicit messages to the synchronization-home nodes. Currently we use CellOS on AP1000+. CellOS does not provide a signal mechanism to users. Therefore, shared memory accesses are not handled by the virtual memory mechanism. But they are executed by code sequences which check valid bit tables. The optimizing compiler inserts this code sequence before each shared memory access. The compiler also inserts message polling[29].

3.2. Protocol Selection at Write Commitment

The above runtime system provides a write-invalidate protocol. We can simulate two other protocols

By modifying behavior at a write commitment, we can select two other protocols[26, 25] at each write commitment.

Broadcast At a write commitment, the writing node sends written contents to all nodes. The node does not set the dirty bit table entry.

Home Only The writer updates the page-home without making a copy. This is achieved by omitting the valid bit table checking of the corresponding shared write.

The broadcast protocol can reduce the communication latency and alleviate false sharing. Broadcast is also useful to efficiently execute a program which is not properly labeled[16]. At the release operation after broadcasting, the sender node must wait for acknowledgments from all nodes. Home only protocol can reduce page fault traffic at fetch-on-write. The contents of the page and the state of the valid bit table entry are temporarily inconsistent until the succeeding synchronization. When a home only write and ordinary page accesses occur in the same page, this may cause incorrect page contents. We introduce the *home only acknowledge table* which records the page-home node for home only write commitments. When a page fault occurs, the node checks this table and waits for an acknowledgment from the page-home node.

To perform the protocol optimization, we have manually specified the type of write commitments in the bottleneck part of a generated source program. When we implement the home only protocol using a virtual memory mechanism, we have to explicitly check the valid bit table at conflicting writes to avoid frequent page faults.

4. Performance Study with SPLASH-2

We used three kernels (LU-Contig, Radix, FFT) and five applications (Barnes, Raytrace, Water-Nsq, Water-Sp, Ocean) from SPLASH-2.

4.1. Compilation Time

At redundancy elimination, we calculated availability with bottom up traversal of the call graph, and calculated ancitipatability intraprocedurally. We show the compilation time of each program in Figure 1. The compiler is run on Sun SPARCstation 20 (with 50MHz SuperSPARC) + SunOS4.1.3. "Scalar dataflow" represents the time to detect induction variables. Without type-safe assumption, points-to analysis takes from 1.4 to 4.2 times longer time for programs with structures containing pointers (Barnes, Raytrace, and Water-Sp) and for a program with pointer casting (Ocean).

Table 2. Input problem size and sequential execution time (in seconds)

program	problem size	sequential
LU-Contig	1024^2 doubles	115.67
Radix	1M integer keys	4.32
FFT	64K complex doubles	2.10
Barnes	16K bodies	54.68
Raytrace	balls4, 128^2 pixels	349.38
Water-Nsq	4096 molecules	800.08
Water-Sp	4096 molecules	88.37
Ocean	130^2 ocean	7.09

4.2. Runtime System

We show the problem size of each program and the sequential execution time on one node. Each node of the AP1000+ consists of 50MHz SuperSPARC (20KB I-cache and 16KB D-cache) and 16MB memory. The nodes are linked by 2D torus network whose bandwidth is 25MB/s per link. The small problem size of Ocean is caused because of the limit of physical memory size.

The page table checking is implemented by software. If we use a virtual memory mechanism, there is no checking overhead when the page is valid. Coalescing and redundancy elimination are also applicable to the software page table checking. We manually applied redundancy elimination to checking codes using a similar interprocedural algorithm to that of write commitments. We selected 4KB page size for kernels, and 1KB for applications. We used gcc 2.7.2 (the optimizing level is -O2) as the backend compiler.

We modified the source codes of FFT and Raytrace. The transpose operation of the original FFT is written so that a receiver reads the parts of the array. But their page-home nodes are not receivers but senders. This causes a severe false sharing. We rewrote the procedure `Transpose` so that a sender writes to the page-home of receivers. In the original Raytrace, lock acquisition for ray ID is a bottleneck for the execution. This ID is not used for any actual computation. We removed this lock operation. For each program, we specified a page-home and a synchronization-home according to optimization hints of SPLASH-2. We applied protocol optimization to Radix, FFT, Barnes, and Raytrace.

In Figure 3, we show effects of compiler optimization on 32 nodes execution. The left bar of each program is the

Table 1. Compilation time of SPLASH-2 (in seconds)

program	number of lines	type checking	points-to analysis w/ assumption	points-to analysis conservative	scalar dataflow	write set calculation
LU-Contig	980	0.77	4.77	4.78	0.90	0.85
Radix	816	0.56	2.78	3.47	0.66	0.33
FFT	992	0.69	1.23	1.27	0.68	2.26
Barnes	3,052	6.44	16.69	25.31	3.28	1.31
Raytrace	10,910	33.23	18.38	71.26	4.06	2.84
Water-Nsq	2,080	1.64	6.05	6.17	2.44	1.04
Water-Sp	2,748	2.41	30.14	42.70	3.91	3.22
Ocean	4,847	10.70	207.46	897.04	22.84	21.20

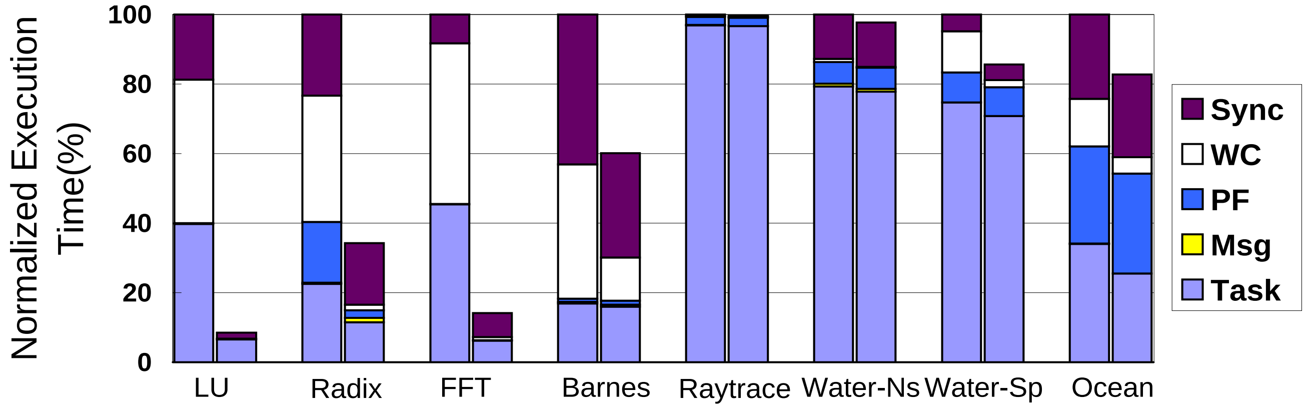
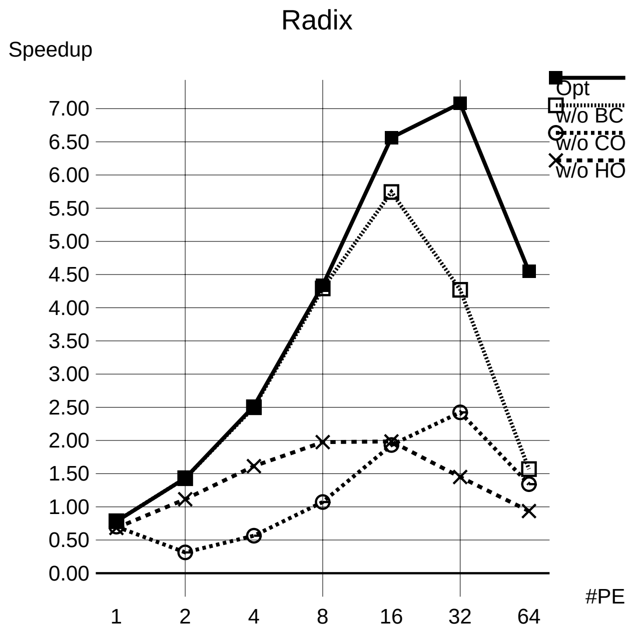


Figure 3. Effects of compiler optimization (executed on 32 nodes)

execution time without compiler optimization (base time). The right bar is that with the optimization. The base run of Radix and FFT use the protocol optimization. Execution time is normalized by the base time. “Sync” means waiting time for synchronization. “WC” means time for write commitment. “PF” means waiting time for page fault. “Msg” is message handling time for synchronization. “Task” is time for the original computation. In LU-Contig, shared writes into each block are coalesced into one write commitment. In Radix, we can apply the previously described coalescing for a continuous variable. In FFT, a write commitment is issued for each column of the block decomposed array. Since LU-Contig, Radix, and FFT contain regular memory accesses in the innermost loops, coalescing reduces from 60% to 90% the total execution time. A write commitment in a innermost loop introduces an overhead of procedure call and reduces memory access locality. Since these overheads are included in task time, optimization also reduces task time. In Radix, the response time of page fault is improved because the traffic of network is reduced by coalescing. In Barnes, the compiler coalesces the write commitments for each record of the structure. Raytrace and Water-Ns have high task ratio and the reduction of execution time is less than 3%. In Ocean, overheads of synchronization and page fault are dominant because of the small problem size. The effect of optimization is confined to 17%.

FFT and Radix are challenging applications for a shared memory system because they potentially require high communication bandwidth because of false sharing[15]. We show effects of protocol optimization to Radix and FFT in Figure 4. In Radix, write operations for the sorted array cause severe false sharing. Since all of the contents of the target array are written at the previous iteration, this traffic can not be reduced by the diff mechanism of LRC. We selected a home only protocol for this write operation. We also selected a broadcast protocol for the array `radix` which is used by all nodes. The left figure shows speedup ratio of Radix. “w/o BC”, “w/o CO”, and “w/o HO” re-



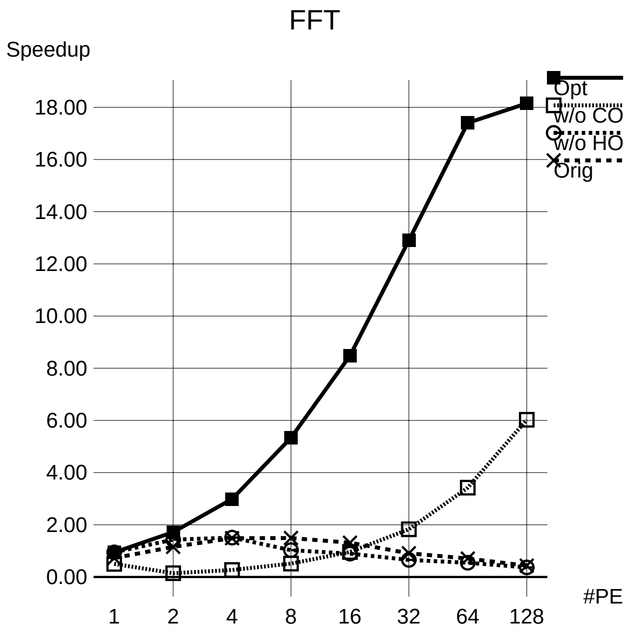


Figure 4. Effects of protocol optimization to Radix and FFT

spectively mean executions without the broadcast protocol, coalescing, and the home only protocol. Though write commitments in the innermost loop cause a large overhead, this part can be parallelized. Without the home only protocol, the performance is saturated over 16 nodes because of heavy traffic. The broadcast protocol is effective also over 16 nodes. The right figure shows the speedup ratio of FFT. "Orig" means the execution of the original SPLASH-2 code. In FFT, the code restructuring of `Transpose` and protocol selection raise the maximal speedup ratio from 1.49 to 18.1.

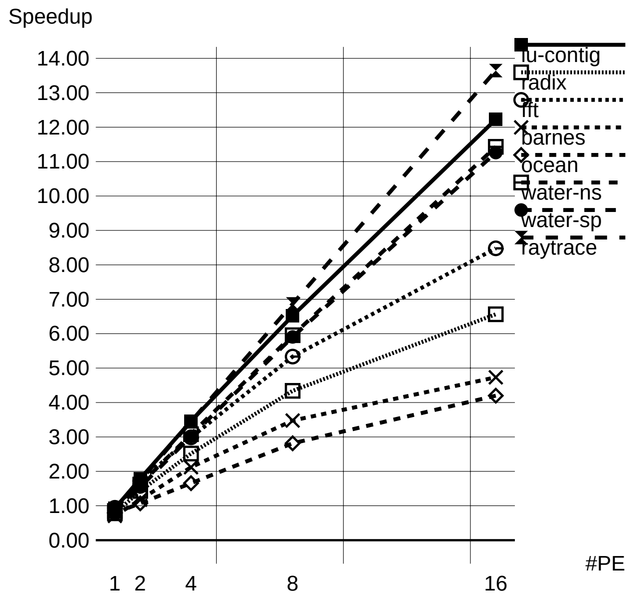
In Figure 5, we show speedup ratio of the programs with compiler optimization and protocol selection. Because of the low overheads of our runtime system and the utilization of the communication bandwidth, Raytrace, LU-Contig, Water-Ns, and Water-Sp show high speedup ratios and a good scalability. Both in Radix and FFT, an appropriate protocol selection is crucial for scalability. The performance of Barnes is saturated over 32 nodes. In Radix and Barnes, the principal overhead is synchronization because of the problem decomposition. Only Ocean slows down owing to the page fault handling which is an overhead of the runtime system. This is mainly because of the small size of the problem. As a whole, both compiler optimization and appropriate protocol specification are essential for scalability of the input problem.

5. Related Work

The computation power of recent machines enables the application of interprocedural analysis to practical problems (e.g. interprocedural points-to analysis[14, 31], interprocedural array dataflow analysis[17], and interprocedural partial redundancy elimination[2]). So far, these advanced analyses have not been used for explicit parallel shared memory programs.

Existing research about cooperation between optimizing compilers and software DSM can be divided in three kinds. The first is that a parallelizing compiler targets software DSM[21, 13, 24]. For parallelizable programs, the compiler can use precise communication information. Message vectorization is applicable to regular communication. The compiler can use code generation techniques for inspector/executor mechanism. Software DSM does not require complex code generation for multi-level indirection. The runtime library has the benefit of message vectorization, synchronization messages, and support for sender initiated communication. However, this policy is only applicable to automatically parallelizable programs.

The second is that a programmer declares shared data and association between data and synchronization[3, 10, 30, 6]. The programmer can select appropriate protocols for each data. The runtime system can utilize application specific information. Since this model hides a memory model from users, the system does not suffer from false sharing.



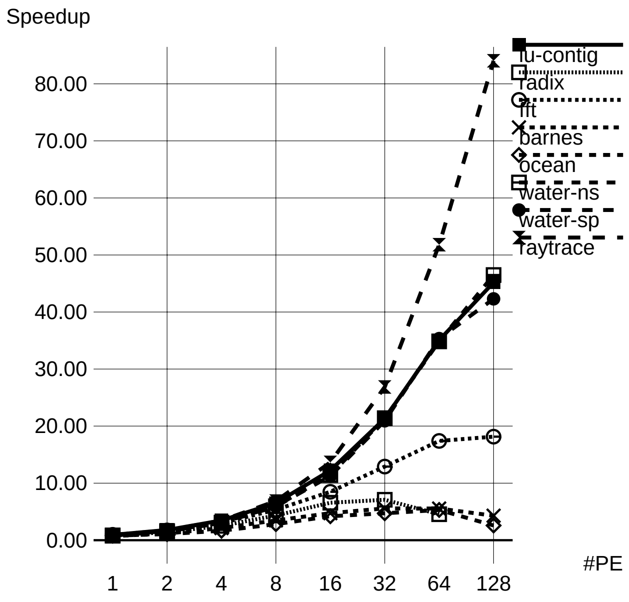


Figure 5. Speedup ratio up to 16 nodes (left) and up to 128 nodes (right)

However, the message packing/unpacking mechanism must be implemented efficiently. Users also have to adjust parallel programs to the provided programming model.

The third is that a compiler directly analyzes a shared memory program. Our system and Shasta[29] are classified in this kind. The Shasta compiler uses two optimizing techniques to reduce software overheads. One is a special flag value which indicates that the content is invalid. If the loaded value is not equal to the flag value, we know that the content is valid without using the page table checking. The other is batching to combine multiple checking for the same entry of the directory. These optimizations are intraprocedural. Since they do not perform loop level optimization, their system requires both high network bandwidth and low latency.

6. Summary

We have shown that compiler support enables efficient software DSM which can utilize communication bandwidth as much as possible. We designed an interface between a shared memory program and a runtime library, and established a coalescing and redundancy elimination problem of write commitments. Our framework enabled applying interprocedural optimizations to a shared memory program. We have described the interprocedural optimization scheme and an efficient implementation of the runtime system. We have shown that the appropriate write protocol selection is one important application specific information for the efficient software DSM.

The redundancy elimination scheme in this paper decreases the number of write commitment as much as possible and makes the size of the write commitment as large as possible. Therefore, it issues write commitments as late as possible. This policy is suitable for the runtime system on AP1000+, since AP1000+ has a fast communication network. However, this is not always optimal, especially on machines with slower communication facilities. Our future work is to reflect this tradeoff of the platform into dataflow equations.

Acknowledgments

We would like to thank the referees for their valuable comments and aadvice. This work is partly supported by Advanced Information Technology Program (AITP) of Information-technology Promotion Agency (IPA) Japan.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 1985.

[2] G. Agrawal, J. Saltz, and R. Das. Interprocedural Partial Redundancy Elimination and its Application to Distributed

Memory Compilation. In *Proc. of '95 Conf. on PLDI*, pages 258–269, June 1995.

[3] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 1993 CompCon Conf.*, pages 528–537, Feb. 1993.

[4] J. Boyle et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[5] M. Burke. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. *ACM Trans. on Programming Languages and Systems*, 12(3):341–395, July 1990.

[6] C. Chang, A. Sussman, and J. Saltz. Object-Oriented Runtime Support for Complex Distributed Data Structures. Technical Report CS-TR-3428, University of Maryland, Mar. 1995.

[7] F. C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proc. of the SIGPLAN '88 Conf. on PLDI*, pages 85–94, June 1988.

[8] J. Cocke. Global Common Subexpression Elimination. *Proc. of a Symp. on Compiler Optimization, SIGPLAN Notices*, 5(7):20–24, July 1970.

[9] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compiler*. New York University Press, 2nd edition, Apr. 1970.

[10] D. E. Culler et al. Parallel Programming in Split-C. In *Proc. of Supercomputing '93*, pages 262–273, Nov. 1993.

[11] G. B. Dantzig and B. C. Evans. Fourier-Motzkin Elimination and Its Dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[12] M. Dubois. *Scalable Shared Memory Multiprocessors*, chapter 11, Delayed Consistency, pages 207–218. Kluwer Academic Publishers, MA, 1992.

[13] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proc. of the ASPLOS-VII*, Oct. 1996.

[14] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proc. of '94 Conf. on PLDI*, pages 242–256, June 1994.

[15] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the Performance of Clusterd Distributed Virtutal Shared Memory. In *Proc. of ASPLOS-VII*, pages 210–220, Oct. 1996.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th ISCA*, pages 15–26, May 1990.

[17] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam. Interprocedural Analysis for Parallelization. In *Proc. of the 8th Int. Workshop on LCPC*. Springer-Verlag, Aug. 1995.

[18] K. Hayashi et al. AP1000+: Architectural Support for Parallelizing Compilers and Parallel Programs. In *Proc. of the 3rd Parallel Computing Workshop*, pages P1–F–1 – P1–F–9, Nov. 1994.

[19] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd HPCA*, Feb. 1996.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th ISCA*, pages 13–21, May 1992.

[21] P. Keleher and C. Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. In *Proc. of the 11th International Parallel Processing Symp.*, Mar. 1996.

[22] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):440–451, Oct. 1991.

[23] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 ICPP*, pages 94–101, Aug. 1988.

[24] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *Proc. of the Symp. on PPoPP*, pages 48–56, 1997.

[25] T. Matsumoto and K. Hiraki. Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 30–39, Los Alamitos, CA, 1998. IEEE Computer Society.

[26] T. Matsumoto, T. Komaarashi, S. Uzuhara, S. Takeoka, and K. Hiraki. A General-Purpose Massively-Parallel Operating System: SSS-CORE – Implementation Methods for Network of Workstations –. In *IPSJ SIG Notes*, volume 96-OS-73, pages 115–120, Aug. 1996. (in Japanese).

[27] E. Morel and C. Renvoise. Global Optimization by Suppression of Partial Redundancies. *CACM*, 22(2):96–103, Feb. 1979.

[28] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory. In *Proc. of the 1997 ISPAN*, pages 198–201, Dec. 1997.

[29] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of ASPLOS-VII*, pages 174–185, Oct. 1996.

[30] D. J. Scales and M. S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. of the 1st OSDI*, Nov. 1994.

[31] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. of '95 Conf. on PLDI*, pages 1–12, June 1995.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, pages 24–36, June 1995.