TECHNICAL REPORT 97-05

Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory

Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, and Kei Hiraki

June 11,1997

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku, Tokyo 113, Japan

TECHNICAL REPORT 97-05

Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory

Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, and Kei Hiraki

June 11,1997

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TECHNICAL REPORT 97-05

TITLE
Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory

AUTHORS
Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, and Kei Hiraki

KEY WORDS AND PHRASES
software distributed shared memory, release consistency, compiler, pointer-analysis

ABSTRACT
The shared memory system can reduce the cost of programming effort in the distributed memory systems. On distributed systems such as networks of computers, it is necessary to provide the shared memory model by software.
We have proposed an "Asymmetric Distributed Shared Memory: ADSM", that provides high-speed-update distributed memory access by software. The compiler for the ADSM translates the code reading to shared memory into a single load instruction and translates the code writing to shared memory into a sequence of instructions, which means instructions managing consistency are separated from the store instruction. The instructions managing consistency are explicitly inserted after the store instruction. According to this property, we can perform various optimizations: (1)We can reduce the number of instructions managing consistency. (2)We can select any protocol per page.
We have implemented prototypes of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+. We use 3 of the SPLASH-2 kernel benchmarks. We evaluate the effectiveness of the optimization on the ADSM scheme by preliminary experiments.

REPORT DATE	**WRITTEN LANGUAGE**
June 11,1997	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
13	13

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
Submitted to SC '97

DISTRIBUTION STATEMENT
This technical report is available ONLY via anonymous FTP from `ftp.is.s.u-tokyo.ac.jp` (directory `/pub/tech-reports`).

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory

Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, and Kei Hiraki

June 11, 1997

Abstract

Programming on distributed memory systems is very difficult because programmers must use message passing library directly. The shared memory system can reduce the cost of programming effort in the distributed systems. On distributed systems such as networks of computers, it is necessary to provide the shared memory model by software.

We have proposed an "Asymmetric Distributed Shared Memory: ADSM", that provides high-speed-update distributed memory access by software. As for the read, the ADSM prepares the page-size cache and exploits the virtual shared memory mechanism. As for the write, the ADSM executes a sequence of instructions for managing consistency after the corresponding store instruction.

In this paper we describe the optimization methods on the ADSM. Because the compiler inserts a sequence of instructions for managing consistency after the store instruction, the ADSM supports protocol selection per page. In order to execute write detection and write collection efficiently, we define write history which consist of the corresponding address and the corresponding size. By using this scheme, we have implemented LRC protocol, AURC protocol emulated by software and Hybrid protocol of LRC and AURC on the ADSM. In order to generate the efficient instructions for managing consistency, we implement the pointer analysis and classify the store as local, must-shared, possibly-shared. When the store is must-shared, the compiler inserts the the instructions for managing consistency.

When the store is possibly-shared, the compiler inserts the instructions which check dynamically the address and manage consistency in the case where the address is shared location. We describe the algorithm to combine sequences of instructions for managing consistency. First the compiler generate the codes using the pointer analysis. For each loopnest the algorithm progress from the innerloop to the outerloop, checking whether the instructions for managing consistency exist. If instructions for managing consistency are loop-invariant or consist of the induction variables and there are no synchronization points, they are combined and hoisted out to the outerloop.

We have implemented prototypes of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+. This time we experiment by emulating the virtual memory mechanism because the operating system of the AP1000+ does not support user-level page fault handler and the optimization methods are independent of the virtual memory mechanism.

We use 3 of the SPLASH-2 kernel benchmarks using 8 nodes. We evaluate the effectiveness of the optimization on the ADSM scheme by experiments. We confirm that our optimization algorithm reduces the execution time significantly. In LU-Contig, the optimized execution time is reduced to the 8 % of the unoptimized execution time.

1 Introduction

The shared memory system can reduce the cost of programming effort in distributed systems. The main reason is that shared data are easily accessed with a single address space, that is to say, the programmers take no account of the location of data.

On distributed systems such as networks of computers, it is necessary to provide the shared memory model by software. The main reason is that the hardware approach to implement the shared memory model requires high cost and the software approach is cost-effective. A number of software page-based shared memory models have been proposed[Li88, KCZ92, IDFL96]. But there are still problems to be solved.

Writing to shared memory is executed as a single store instruction. The consistency management instructions are executed in the write trap routine. In order to execute consistency management instructions, the write trap routine must be invoked[Li88]. In order to minimize the number of trap invocation, techniques such as diff scheme which computes the page's difference between the synchronization points have been proposed[KCZ92]. But this diff scheme can be expensive because the whole page is searched independently of the amount of the modified data. This overhead is not negligible. By using the special hardware support, the problem concerning the diff scheme can be avoided[IDFL96].

In order to minimize the number of trap invocation, we have proposed a software shared memory system, Asymmetric Distributed Shared Memory(ADSM)[MKU+96].

- Page-based system:
  The system prepares the page-size cache and provides the illusion of shared memory by exploiting the virtual memory mechanism[Li88].
- Shared read:
  The compiler translates the code reading from the shared memory into a load instruction.
- Shared write:
  The compiler translates the code writing to the shared memory into a sequence of instructions:
  1. A store instruction:
  2. Consistency management instructions:

  Consistency management instructions are explicitly inserted after the store instruction by the compiler and are not executed in the write trap routine.

We call this software DSM asymmetric because the behavior of write is different from that of read. Because consistency management instructions are not executed in the write trap routine and are inserted by the compiler, we can perform various optimizations.

1. Selecting Protocols:
   The compiler can generate consistency management instructions according to the consistency protocol we want to implement.
2. Reducing the number of consistency management instructions:
   The compiler does not always have to generate consistency management instructions after the corresponding store instruction. The compiler executes optimization by combining a series of consistency management instructions. For example, when a sequence of write is performed to the same shared location, it is not necessary to insert consistency management instructions after each corresponding store instruction. The compiler inserts consistency management instructions only after the last store instruction.

2 Related Work

In this section, we briefly review existing software DSMs.

- *Lazy Release Consistency*(LRC)[KCZ92]:
  TeadMarks system[KDCZ94] implements this protocol. It is page-based, all-software, multiple writer protocol. The coherence action(e.g., invalidation) does not propagate at the release point but the corresponding acquire point. The invalidation message consists of *write-notice* which records the updated page number.
  1. Write detection:
     The operating system detects the shared writes. The page is mapped read_only when cold miss. The first write to the page invokes the protection violation and the write trap routine creates the copy of the page(called *twin*) and maps the page read_write. After that, the writes to the page can be executed with no overheads until the program reaches to the release point.

2. Write collection:
   The system compares the current page with the twin, and the differences are recorded as a *diff* which is a run-length encoding of the difference. The diff creation is executed eagerly(at the synchronization points) or lazily(on demand).

- *Automatic update release consistency*(AURC)[IDFL96]:
  AURC is LRC supported by automatic update hardware. Automatic update is a mechanism that automatically propagates updates at fine granularity when a mapping has been established between the local page written and a remote page.

  1. Write detection:
     AURC has a *home* for each shared page. By automatic update hardware, the shared writes are propagated to the home without software overhead.
  2. Write collection:
     The home is always kept up-to-date. The consistency of other copies is managed like LRC invalidation scheme. When the page fault occurs, fetching the whole page from the home is only required.

- *Entry Consistency*(EC)[ZSB94]:
  In the Midway which implements EC, each shared data is bound to the synchronization variable. On a lock acquire point, EC only propagates modified data associated with that lock variable.

  1. Write detection:
     The compiler detects the shared writes. Every shared address has a *software dirty bit* that reflects whether or not the address has been written. After each write to shared memory, the compiler inserts instructions which set a software dirty bit with that address.
  2. Write collection:
     Searching the software dirty bit region, the processor determines the data that it modifies.

3 Features of the ADSM

- Page-based system:
  The ADSM prepares the page-size cache and provides the illusion of shared memory by exploiting the virtual memory mechanism[Li88]. The local memory holds both locally allocated data and cashes for non-local pages in the global virtual space.

  Unlike Midway based on Entry Consistency, we are based on the page-based DSM and we do not force the programmers to rewrite the program. The ADSM does not explicitly force programmers to associate the data with the synchronization variables.

- Shared read:
  The compiler translates the code reading from the shared memory into a load instruction. Only when the processor reads from the shared page that is not allocated or invalid, consistency management instructions are invoked by the read trap routine.

- Shared write:
  The compiler translates the code writing to the shared memory into a sequence of instructions:

  1. A store instruction:
  2. Consistency management instructions:

  Consistency management instructions are explicitly inserted after the store instruction by the compiler and are not executed in the write trap routine. When consistency management instructions contain the remote memory access, the access is performed by using message passing library directly.

We call this software DSM asymmetric because the behavior of write is different from that of read. Since consistency management instructions are not executed in the write trap routine and are inserted by the compiler, we can perform various optimizations described in the following sections.

4 Selecting consistency protocols

Midway only inserts the routine that sets the software dirty bit after each store instruction[ZSB94]. But in the ADSM, we can generate consistency management instructions according to consistency protocol. That is to say, we can implement any consistency protocol. Therefore the ADSM supports consistency protocol selection per page[Mat89]. Each existing system supports only one protocol[Li88, KCZ92, IDFL96, ZSB94]. Therefore the ADSM is more flexible in consistency protocol than any other system. We have implemented three protocols.

We define the *write history* of store as the tuple of the store address and the store size. In order to execute the write collection and write detection efficiently, we use the *write history* information.

- LRC protocol:
  We use an invalidation scheme using the write notice[KCZ92]. But we do not use a twin-diff scheme.

  1. Write detection:
     After the store instruction, the compiler generates consistency management instructions which record the *write history* of the store.
  2. Write collection:
     By using the *write history* information, the processor computes the local write result efficiently when it is necessary(Figure 1). The time to collect the write result is proportional to the amount of the created *write history*. The created *write history* can not be discarded without explicit synchronization, which means the large amount of memory is consumed.

- AURC protocol[IDFL96] emulated by software:
  We implement AURC protocol without special hardware support. It has been implemented as HLRC[YZL96] which uses diff scheme. We do not use the diff scheme but the *write history* scheme.

  1. Write detection:
     After the store instruction, the compiler generates consistency management instructions which record the *write history* of the store. At the release point, the local write result is propagated to the home by using the *write history* information(Figure 2). After sending the local write result, the processor discards the created *write history*. The home discards the *write history* information after modifying the page.
  2. Write collection:
     The home is always kept up-to-date. The consistency of other copies is managed like LRC invalidation scheme. When the page fault occurs, fetching the whole page from the home is only required.

- HYBRID protocol:
  In LRC protocol, more than one remote processor may have to be visited in order to obtain diffs at the page fault. In AURC protocol, the processor has only to visit the home at the page fault. But the whole page must be fetched from the home. Therefore higher bandwidth is required than in the case of sending only the diffs. We propose a hybrid protocol of LRC and AURC emulated by software to solve these problems.

  In HYBRID protocol, at the page fault, the processor has only to visit the home and obtains from the home only modified data rather than the whole page.

  1. Write detection:
     After the store instruction, the compiler generates consistency management instructions which record the *write history* of the store. At the release point, the local write result is propagated to the home by using the *write history* information. After sending the local write result, the processor discards the *write history*. The home records the *write history*(Figure 3).
  2. Write collection:
     The home is always kept up-to-date. The consistency of other copies is managed like LRC invalidation scheme. When the page fault occurs, the faulting processor sends its timestamp

to the home, and the home sends only the modified data by using the recorded *write history* information.

5 Reducing the number of consistency management instructions

Optimization methods to reduce the number of consistency management instructions are described in this section.

5.1 Reducing the number of possibly-shared write

When a write through a pointer exists, it may access the shared locations. Therefore after the store instruction through the pointer, the compiler always need to insert the instructions which check dynamically the address and manage consistency in the case where the address is shared location. This overhead is not be negligible.

In Midway, in order to reduce this overhead, the programmer can annotate the code with `shared` or `private`. In order to reduce this overhead without using annotations, we use the pointer analysis[EGH94, WL95]. When a write through a pointer exists, the compiler executes following operations by using informations obtained in the pointer analysis:

1. When the pointer always points to the shared locations, i.e., (a shared variable or a shared heap location), the write to this location is considered as *must-shared*. Therefore the consistency management instructions are inserted after the store instruction.
2. When the pointer points to the local locations, i.e., (the local variable or the heap allocated by the local malloc), the write to this location is considered as *local*. No instructions are inserted after the store instruction.
3. When the pointer points to the local locations and the shared locations, the write to this location is considered as *possibly-shared*. The compiler inserts the instructions which check dynamically the address and manage consistency in the case where the address is shared location.

5.2 Combining a series of consistency management instructions

In many applications, a series of write to the contiguous location is often seen. In this case the compiler can perform further optimization. For example, let us consider the following code derived from the LU kernel in the SPLASH-2[WOT+95].

```
for (i=0; i<n; i = i + 1) {
 a[i] = a[i] + alpha*b[i];
}
```

When the array `a` is shared, the code is transformed to the following code without combining consistency management instructions(Let the protocol be LRC and `CMI` be the consistency management instructions).

```
for (i=0; i<n; i = i + 1) {
 a[i] = a[i] + alpha*b[i];
 CMI(LRC,&a[i],sizeof(double));
}
```

In the RC model the write to the shared location need not to be informed to other processors until the program comes to the synchronizing point. This means that it is not necessary for consistency management instructions to be inserted exactly after the corresponding store instruction.

When a series of *must-shared* store is performed to the contiguous locations and the program does not reach to the synchronizing points, a series of corresponding consistency management instructions is combined.

Each loop has a *depth*. It is 0 when the loop is outermost. It is n when the loop is innermost of the n-th loopnest. We represent consistency management instructions as a tuple $I(A, S)$. A represents the corresponding memory location and S represents the corresponding size.

From the program, we enumerate the n-th nested loopnest that contains the *must-shared* store which is always executed. For each loopnest,

1. We generate the code without combining optimization.
2. $depth := n$.
3. if $depth = 0$ then end.
4. Take a loop L whose level is $depth$.
   (a) When the sync code or the return code exists in the corresponding loop or inner loops, goto 5;
   (b) In the corresponding loop, when there is a series of consistency management instructions whose location is contiguous or the same, the series of consistency management instructions is combined. For example, when there is a series of instructions such as $I1 = I(A, S)$, $I2 = I(A', S')$(supposed $A < A'$) and $A' < A+S$, $I1$ and $I2$ are combined as $I(A, A'-A+S')$(Figure 4).
   (c) For each consistency management instructions represented as $I(A, S)$.
      i. When A is loop-invariant, the consistency management instructions are hoisted out from the L to the outerloop of L(Figure 5).
      ii. When A is an induction variable and the variable's stride(A_{stride}) is equal to or smaller than S, the consistency management instructions are hoisted from the L to the outerloop of L. Let c be the loop count and A_{low} be the lower bound of the A. The hoisted consistency management instructions are represented as $I\ (A_{low}, (c-1) * A_{stride} + S)$(Figure 5).
5. $depth := depth - 1$ and goto 3.

When this algorithm is applied to the above loop, the following code is generated.

```
for (i=0; i<n; i = i + 1) {
  a[i] = a[i] + alpha*b[i];
}
CMI(LRC,&a[0],sizeof(double)*n);
```

We give a more complicated example. The following loopnest is derived from the LU kernel in the SPLASH-2[WOT+95].

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    if (n - i <= edge) {
      ibs = edge;
      ibs = n - edge;
      skip = edge;
    } else {
      ibs = block_size;
      skip = block_size;
    }
    if (n - j <= edge) {
      jbs = edge;
      jbs = n - edge;
    } else
      jbs = block_size;
    ii = i / block_size
         + (j / block_size) * nblocks;
    jj = i % ibs + (j % jbs) * skip;
    rhs[i] = rhs[i] + a[ii][jj];
  }
}
```

Let the array `rhs` be shared and the protocol be LRC. We apply the algorithm to the loopnest.

- The code is translated into the following code.

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
    CMI(LRC,&rhs[i],sizeof(double));
  }
}
```

- $depth := 2$
- In the loop whose level is 2, there are consistency management instructions represented as $I($`&rhs[i]`,`sizeof(double)`$)$. No other consistency management instructions are found. Therefore we go to next step.
- In this case $A =$`&rhs[i]` and $S =$`sizeof(double)`. `&rhs[i]` is an induction variable and the stride(A_{stride}) is `sizeof(double)` and is equal to the S. Therefore we perform the combining and hoist out the consistency management instructions to the outerloop. In this case A_{low} is `&rhs[0]` and the loop count is `n`. The hoisted code is represented as $I($`&rhs[0]`,`n*sizeof(double)`$)$. Then the following code is generated

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
  }
  CMI(LRC,&rhs[0],n*sizeof(double));
}
```

- $depth := 1$
- In the loop whose level is 1, there are consistency management instructions represented as $I($`&rhs[0]`,`n*sizeof(double)`$)$. No other consistency management instructions are found. Therefore we go to next step.
- In this case $A =$`&rhs[0]` and $S =$`n*sizeof(double)`. `&rhs[0]` is loop-invariant and we perform the combining. We hoist out the consistency management instructions to the outerloop. Then the following code is generated.

```
for (j = 0; j < n; j = j + 1) {
  for (i = 0; i < n; i = i + 1) {
    ......
    rhs[i] = rhs[i] + a[ii][jj];
  }
}
CMI(LRC,&rhs[0],n*sizeof(double));
```

- $depth := 0$
  Therefore this algorithm terminates.

6 Performance Evaluation

We have implemented the prototypes of the compiler and the runtime system of the ADSM on a multicomputer Fujitsu AP1000+[HDH+94, SKI+91]. Each node consists of 50MHz Super SPARC with 20 Kbytes I-cache 16 Kbytes D-cache and 16MB memory. The nodes are interconnected by a 2-D torus network whose bandwidth is 25 Mbytes/sec per link. The operating system of the AP1000+ does not support user-level page fault handler. Therefore we can not exploit the virtual memory mechanism. But on the AP1000+ we can evaluate the effectiveness of the compiler optimization which is independent of the virtual memory mechanism. This time we experiment by emulating the virtual memory mechanism. When the processor attempts to access the shared page that is not allocated or invalid, we call the user-level page fault handler. In order to detect that the processor attempts to access the shared page that is not allocated or invalid, we insert the code checking the corresponding page's validity before each shared access.

Because it takes high cost to handle messages via interrupts on the AP1000+, request messages from remote nodes are serviced through polling mechanism. We insert polls at every loop backedge and every function call.

We evaluate the performance on three kernels(LU-Contig, Radix, FFT) of SPLASH-2[WOT+95] using 8 nodes. LU-Contig performs blocked LU factorization of a dense matrix. The problem size is a 256×256 matrix with 16×16 blocks. Table1 shows the results of LU-Contig. Radix performs an integer radix sort. The problem size is 256K keys to be sorted and a radix of 1024. Table2 shows the results of Radix. FFT performs a complex, 1D-FFT computation. The problem size is 16384 complex data points. Table3 shows the results of FFT. In all cases we perform the optimization using the pointer analysis described in 5.1 and all accesses through pointers are classified as either *local* or *must-shared*. We evaluate the optimization of combining consistency management instructions.

For each table, LRC(*) shows the results when LRC protocol is selected. SAURC(*) shows the results when AURC protocol emulated by software is selected. HYBRID(*) shows the results when HYBRID protocol we propose is selected. *(NO) shows the results when the compiler does not perform an optimization(combining consistency management instructions). *(O) shows the results when the compiler performs the optimization. The "task" shows the computing time. The "message" shows the time spent in handling remote requests by polling and also shows the ratio of the number of messages handled by polling to the total message number. The "PF" shows the page fault time and also shows the number of the page fault. The "CM" shows the consistency managing time and also shows the number of consistency management instructions. The "sync"shows the synchronization time such as lock/unlock and barrier. The "GC" shows the time spent in garbage collection and also shows the number of garbage collection.

On the AP1000+, since the cache writes through, all modifications that the processor makes to cached data are reflected externally[HDH+94, SKI+91]. That is to say, the cost of store is relatively high. In the "CM" time, store instructions are executed most frequently. Therefore the overhead of consistency managing time is relatively high.

For each protocol, the optimization combining consistency management instructions(*(O)) obtains good results because the overhead of consistency managing time is relatively high. When the number of consistency management instructions is reduced, "CM" time is reduced and the amount of the created *write history* is also reduced. Many operations executed in "PF" time,"sync" time and "message" time are depend on the amount of the created *write history*. Therefore when the number of consistency management instructions is reduced, "message" time, "PF" time "CM" time and "sync" time are all reduced.

In LU-Contig, the optimization combining the consistency management instructions is most effective in three kernels. Because LU-Contig is an application that the processor accesses the contiguous locations, there are many opportunities to combine the consistency management instructions. The number of consistency management instructions is reduced to 8 %. In LRC protocol and HYBRID protocol the sync time is reduced to about 10 %. In LRC protocol total time is reduced to 8 %.

In Radix, the optimization combining the consistency management instructions is not performed so much. The number of consistency management instructions is reduced to only 92 %. Therefore the difference between *(O) and *(NO) is small. Because Radix is an application where multiple processors update the same page simultaneously, total message number of LRC is quite larger than that of AURC or HYBRID.

In FFT, the number of consistency management instructions is reduced to 30% by the optimization combining the consistency management instructions. In HYBRID the total time is reduced to 46 %.

In our implementation on the AP1000+, SAURC is best consistency protocol. The results of HYBRID are almost same to those of SAURC. These 2 protocols show better results than LRC protocol. In LRC protocol, LRC's consistency managing time is very high because LRC protocol is very complicated. Moreover, many operations executed in "PF" time,"sync" time and "message" time are more sensitive to the amount of *write history* than those of other 2 protocols. Therefore the result of LRC(NO) is worst in all cases.

7 Conclusion

We describe the optimization methods on the ADSM such as protocol selection and reduction of the consistency managing codes.

We define the *write history* of store as the tuple of the store address and the store size. In order to execute write detection and write collection efficiently, we use *write history* scheme. We have implemented three consistency protocols:(1)LRC protocol, (2)AURC protocol and (3)a hybrid protocol of LRC and AURC. The compiler can generate the consistency management instructions according to the consistency protocol.

In order to reduce the number of *possibly-shared* writes, we do not force programmers to annotate the code but use the pointer analysis. We have proposed a new algorithm to combine a series of consistency management instructions in a loopnest and explain the algorithm by an example.

We have implemented prototypes of the compiler and the runtime system on a multicomputer Fujitsu AP1000+ and evaluate the performance on three kernels of SPLASH-2 using 8 nodes. We confirm that our optimization algorithm reduces the execution time significantly. In LU-Contig, when LRC protocol is selected, the optimized execution time is reduced to the 8% of the unoptimized execution time.

As a future work, the consistency management instructions can stand further improvement. The combining algorithm also can stand further improvement. If we implement the interprocedural array dataflow analysis, then the compiler can generate better codes.

References

[EGH94] M. Emami, R. Ghiya, and L. J. Hendern. Context-Sensitive Interprocedural Points-To Analysis in the Presence of Function Pointers. In *Proc. of '94 Conf. on PLDI*, pages 242–256, June 1994.

[HDH+94] Kenichi Hayashi, Tunehisa Doi, Takeshi Horie, Yoichi Koyanagi, Osamu Shiraki, Nobutaka Imamura, Toshiyuki Shimizu, Hiroaki Ishihata, and Tatsuya Shindo. Ap1000+: Architectural support for parallelizing compilers and parallel programs. In *Third Parallel Computing Workshop*, pages P1–F1–P1–F9, nov 1994.

[IDFL96] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd Inter. Symp. on HPCA*, February 1996.

[KCZ92] 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.

[KDCZ94] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[Li88] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 ICPP*, pages 94–101, August 1988.

[Mat89] Takashi Matsumoto. Fine Grain Support Mechnisms. In *IPSJ Computer Architecture SIG Notes*, volume 89-ARC-77, pages 91–98, July 1989. (in Japanese).

[MKU+96] Takashi Matsumoto, Taketo Komaarashi, Shigeru Uzuhara, Shozo Takeoka, and Kei Hiraki. A General-Purpose Massively-Parallel Operating System: SSS-CORE – Implementation Methods for Network of Workstations –. In *IPSJ Operating System SIG Notes*, volume 96-OS-73, pages 115–120, August 1996. (in Japanese).

[SKI+91] Osamu Shiraki, Yoichi Koyanagi, Nobutaka Imamura, Kenichi Hayashi, Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Architecture of highly parallel computer ap1000+. In *Third Parallel Computing Workshop*, pages P1–G–1–P1–G–8, nov 1991.

[WL95] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Prgrams. In *Proc. of '95 Conf. on PLDI*, pages 1–12, June 1995.

[WOT+95] 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.

[YZL96] Liviu Iftode Yuanyuan Zhou and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proc. of the 2nd Symp. on OSDI*, 1996.

[ZSB94] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proc. of the 1st Symp. on OSDI*, pages 87–100, November 1994.

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	35.887590	1.852368	0.061613	0.519496	22.074052	11.348296	0.031765
			195/724	204	699040		3
LRC(O)	3.166359	0.624565	0.045341	0.305522	0.550551	1.610131	0.030249
			196/596	216	49640		2
SAURC(NO)	13.556636	1.816375	1.136838	0.830893	4.916014	4.856516	0.000000
			262/437	210	699040		0
SAURC(O)	3.124313	0.618595	0.165564	0.287180	0.562971	1.490004	0.000000
			217/392	210	49640		0
HYBRID(NO)	25.881067	1.811679	4.479394	2.939476	5.974661	10.648371	0.027487
			211/617	103	699040		9
HYBRID(O)	4.234046	0.621573	0.404990	0.568746	0.709688	1.929049	0.000000
			208/353	209	49640		0

Table 1: LU-Contig(256)

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	15.113220	0.654110	0.035647	0.455719	11.627622	2.216233	0.123888
			347/2158	284	212093		2
LRC(O)	9.217054	0.611876	0.023610	0.405133	6.242657	1.833465	0.100312
			285/2573	291	196635		1
SAURC(NO)	3.129691	0.581339	0.263697	0.313482	1.364119	0.607055	0.000000
			309/582	470	212093		0
SAURC(O)	2.980493	0.553462	0.298612	0.355711	1.348237	0.424471	0.000000
			294/588	472	196635		0
HYBRID(NO)	6.828741	0.660363	1.433136	0.467239	1.668042	2.548010	0.051951
			280/979	441	212093		2
HYBRID(O)	5.886107	0.599612	1.508738	0.444150	1.575028	1.705773	0.052806
			274/978	438	196635		2

Table 2: Radix(262,144)

unit : sec.	total	task	message	PF	CM	sync	GC
LRC(NO)	10.329471	0.493014	0.100594	0.791183	8.648155	0.287827	0.008698
			323/425	307	174080		1
LRC(O)	8.470524	0.279807	0.012507	0.077706	4.879793	3.208724	0.011987
			109/425	118	55248		4
SAURC(NO)	2.577256	0.469006	0.217816	0.121456	1.152603	0.616376	0.000000
			296/446	330	174080		0
SAURC(O)	1.210733	0.238359	0.091413	0.116116	0.370441	0.394403	0.000000
			291/443	331	55248		0
HYBRID(NO)	4.651262	0.473442	1.529980	0.413699	1.505096	0.724004	0.005041
			229/418	275	174080		1
HYBRID(O)	2.147053	0.240031	0.398685	0.464955	0.475990	0.567393	0.000000
			235/390	330	55248		0

Table 3: FFT(16,384)

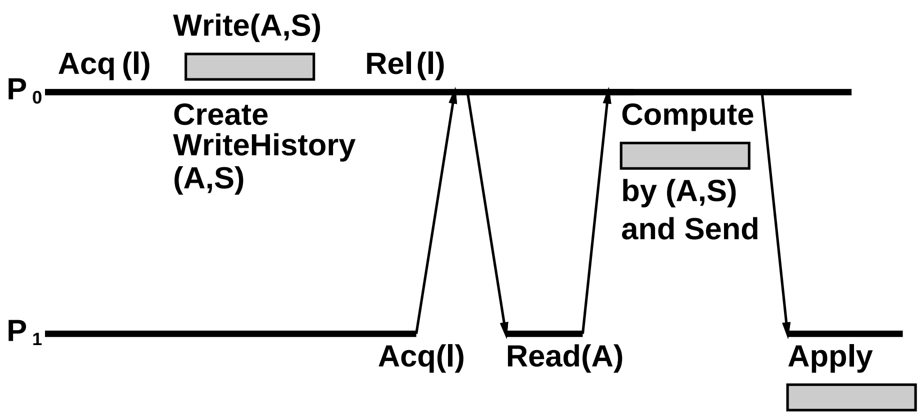

Figure 1: An example of LRC

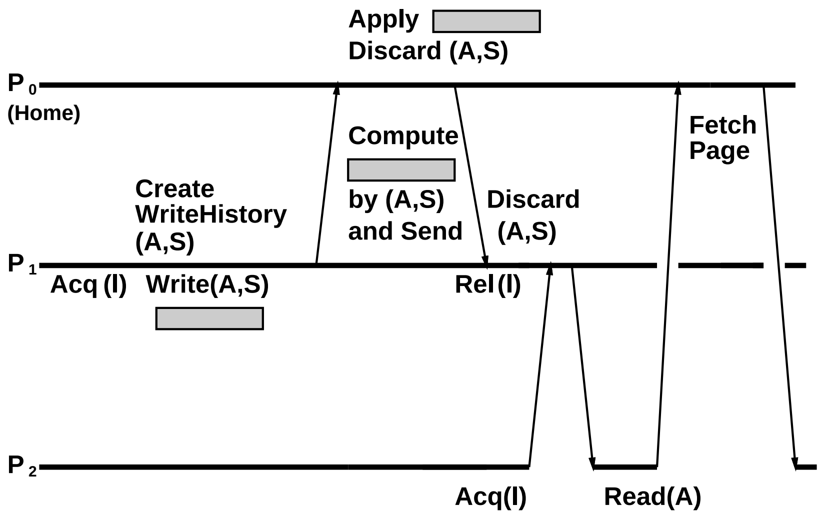

Figure 2: An example of AURC emulated by software

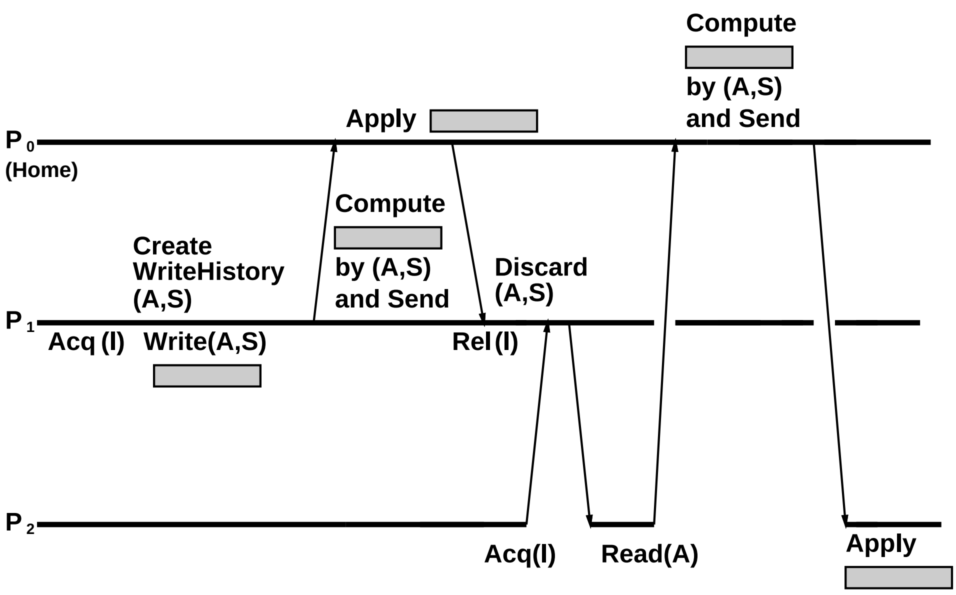

Figure 3: An example of Hybrid protocol

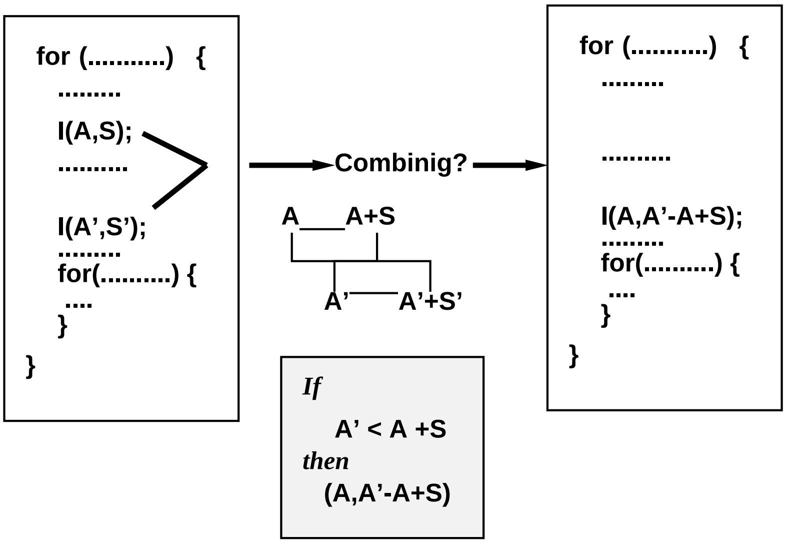

Figure 4: Combing the codes managing consistency

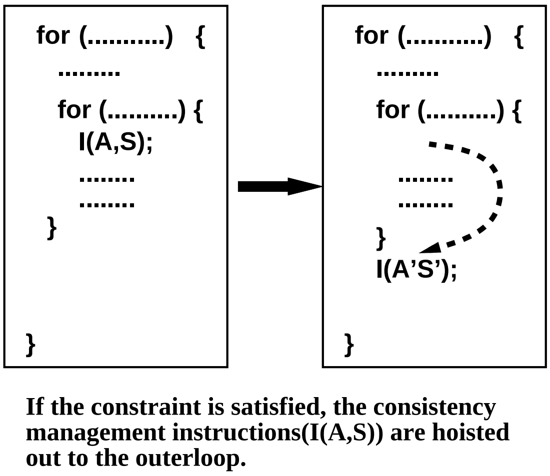

Figure 5: Hoisting the codes managing consistency