# OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection

Yukihiko Sohda, Hirotaka Ogawa, and Satoshi Matsuoka

Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Oo-okayama, Meguro-ku, Tokyo 152-8552
TEL: (03)5734-3876 FAX: (03)5734-3876
email: {sohda,ogawa,matsu}@is.titech.ac.jp

**Abstract.** Platform portability is one of the utmost demanded properties of a system today, due to the diversity of runtime execution environment of wide-area networks, and parallel programs are no exceptions. However, parallel execution environments are VERY diverse, could change dynamically, while performance must be portable as well. As a result, techniques for achieving platform portability are sometimes not appropriate, or could restrict the programming model, e.g., to simple message passing. Instead, we propose the use of *reflection* for achieving platform portability of parallel programs. As a prototype experiment, a software DSM system called OMPC++ was created which utilizes the compile-time metaprogramming features of OpenC++ 2.5 to generate a message-passing MPC++ code from a SPMD-style, shared-memory C++ program. The translation creates memory management objects on each node to manage the consistency protocols for objects arrays residing on different nodes. Read- and write- barriers are automatically inserted on references to shared objects. The resulting system turned out to be quite easy to construct compared to traditional DSM construction methodologies. We evaluated this system on a PC cluster linked by the Myrinet gigabit network, and resulted in reasonable performance compared to a high-performance SMP.

## 1 Introduction

Due to rapid commoditization of advanced hardware, parallel machines, which had been specialized and of limited use, are being commoditized in the form of workstation and PC clusters. On the other hand, commodity software technologies such as standard libraries, object-orientation, and components are not sufficient for guaranteeing that the same code will work across all parallel platforms not only with the same set of features but also similar performance characteristics, similar fault guarantees, etc. Such *performance portability* is fundamentally difficult because platforms differ in processors, number of nodes, communication hardware, operating systems, libraries, etc., despite commoditization.

Traditionally, portability amongst diverse parallel computers have been either achieved by standard libraries such as MPI, or parallel programming languages and compilers such as HPF and OpenMP[Ope97]. However, such efforts will could require programming under a fixed programming model. Moreover, portable implementation of such systems themselves are quite difficult and require substantial effort and cost. Instead, *Reflection* and *open compilers* could be alternative methodologies and techniques for *performance portable* high-performance programs.

Based on such a belief, we are currently embarked on the OpenJIT[MOS+98] project. OpenJIT is a (reflective) Just-In-Time open compiler written almost entirely in Java, and plugs into the standard JDK 1.1.x and 1.2 JVM. At the same time, OpenJIT is designed to be a compiler framework similar to Stanford SUIF[Uni], in that it facilitates user-customizable high-level and low-level program analysis and transformation frameworks. With OpenJIT, parallel programs of various parallel programming models in Java compiled into Java bytecode, will be downloaded and executed on diverse platforms over the network, from single-node computers to large-scale parallel clusters and MPPs, along with customization classes for respective platforms and programming models using compiler metaclasses[1].

The question is, will such a scheme be feasible, especially with strong requirements for performance of high-performance parallel programs? Moreover, how much metaprogramming effort would such an approach take? So, as a precursor work using OpenC++, we have employed reflection to implement DSM (distributed shared memory) in a portable way, to support Java's chief model of parallel programming i.e., the multithreaded execution over shared memory. More specifically, we have designed a set of compiler metaclasses and the supportive template classes and runtimes for OpenC++2.5[Chi95, Chi97] that implements necessary program transformations with its compile-time MOP for efficient and portable implementation of software-based DSM for programs written in (shared-memory) SPMD style, called *OMPC++*. A multithreaded C++ program is transformed into message-passing program in MPC++[Ish96] level0 MTTL (multithread template library), and executed on our RWC(Real-World Computing Partnership)-spec PC-cluster, whose nodes are standard PCs but interconnected with the Myrinet[Myr] gigabit network, and running the RWC's SCore parallel operating system based on Linux.

The resulting OMPC++ is quite small, requiring approximately 700 lines of metaclass, template class, and runtime programming. Also, OMPC++ proved to be competitive with traditional software-based DSM implementations as well as hardware-based SMP machines. Early benchmark results using numerical core programs written in shared-memory SPMD-style programs (a fast parallel CG-kernel, and parallel FFT from SPLASH2) shown that, our reflective DSM implementation scales well, and achieves performance competitive with that of high-performance SMPs (SparcServer 4000, which has dedicated and expensive

[1] OpenJIT is in active development, and is currently readying the first release as of Feb. 1, 1999.

hardware for maintaining hardware memory consistency). Not only this result serves as a solid groundwork for OpenJIT, but OMPC++ itself serves as a high-performance and portable DSM in its own right.

## 2 Implementation of a Portable, High-Performance Software DSM

DSM (Distributed Shared Memory) has had multitudes of studies since its original proposal by Kai Li[LH89], with proposals for various software as well as hardware assisted coherency protocols, techniques for software-based implementation, etc. However, there have not been so much work with platform (performance) portability in mind. In order for a program to be portable, one must not extensively alter the underlying assumptions on the hardware, OS, the programming language, which are largely commoditized. Moreover, as a realistic parallel system, one must achieve high performance and its portability across different platforms.

Below, we give an outline of our OMPC++ system with the above requirements in mind, as well as the building blocks we have employed.

### 2.1 Overview of the OMPC++ system

OMPC++ takes a parallel multithreaded shared-memory C++ program, transforms the program using compile-time metaprogramming, and emits an executable depending on various environments, those with fast underlying message passing in particular(Fig. 2). OMPC++ itself does not extend the C++ syntax in any way.

More concretely, OMPC++ defines several template classes (distributed shared array classes), and OpenC+ metaclasses for performing the necessary program transformations for implementing software DSM (such as read/write barriers to shared regions, and initialization/finalization). The OpenC++ compiler generates a customized program transformer, which then transforms the program into message-passing MPC++ program, which is further compiled into C++ and then onto a binary, and finally linked with DSM template libraries as well as MPC++ runtime libraries, resulting in an executable on the RWC cluster.

Our system exhibits competitive and sometimes superior performance to software DSM systems implemented as class libraries. This is because OMPC++ can analyze and pinpoint at compile time, exactly where we should insert runtime meta-operations (such as read/write barriers) that would result in performance overhead. Thus, we only incur the overhead when it is necessary. On the other hand, for traditional class-based DSM systems, either the programmer must insert such meta-operations manually, or it would incur unnecessary runtime overhead resulting in loss of performance.

### 2.2 Underlying 'Building-Blocks'

We next describe the underlying 'building-blocks', namely OpenC++, MPC++, SCore, and the PC Cluster hardware(Fig. 1). We note that, aside from OpenC++, the components could be interchanged with those with similar but different functionalities and/or interfaces, and only the metaclasses have to be re-written. For example, one could easily port OMPC++ to other cluster environments, such as Beowulf[Beo].

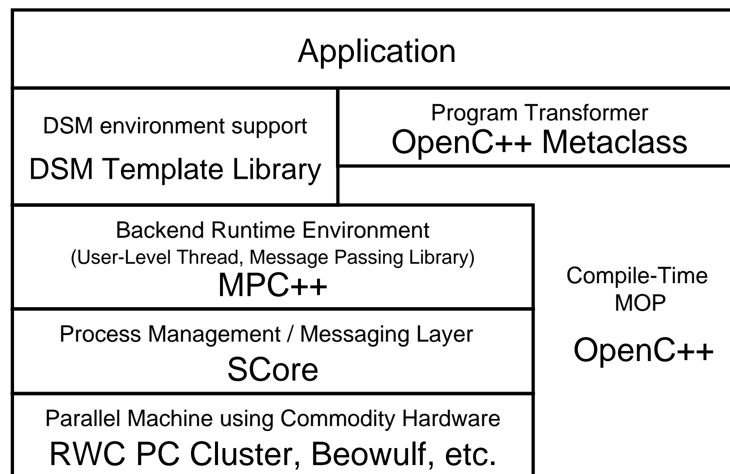


**Fig. 1.** Building-blocks hierarchy

**OpenC++** OpenC++ provides a compile-time MOP capability for C++. By defining appropriate compiler metaclasses, the OpenC++ compiler effectively generates a preprocessor that performs the intended program transformations. The preprocessor then can be applied to the base program for appropriate meta-operations. Compile-time MOP could eliminate much of the overhead of reflection, and coupled with appropriate runtime, could simulate the effects of traditional run-time MOPs. (OMPC++ currently employs OpenC++ 2.5.4.)

**MPC++** MPC++ 2.0 Level 0 MTTL is a parallel object-based programming language which supports high-level message-passing using templates and inheritance. MTTL is solely implemented using language features of C++, and no custom language extensions are done. For OMPC++, we employ most of the features of MTTL, such as local/remote thread creation, global pointers, and reduction operations[2].

[2] MPC++ Level 1[Ie96] supports compile-time MOP features similar to OpenC++. The reason for not utilizing them was not a deep technical issue, but rather for

**RWC PC Cluster and SCore** The default OMPC++ library emits code for RWC PC cluster, although with a small amount of metaclass reprogramming it could be ported to other cluster platforms. The cluster we employed is a subset of the RWC cluster II, with 8–10 Pentium Pro (200MHz) PC nodes interconnected with Myrinet gigabit LAN. The nodes in the cluster communicate with each other using PM, a fast message-passing library, and the whole cluster acts as one parallel machine using the SCore operating system based on Linux. SCore is portable in that it only extends Linux using daemons and device drivers.

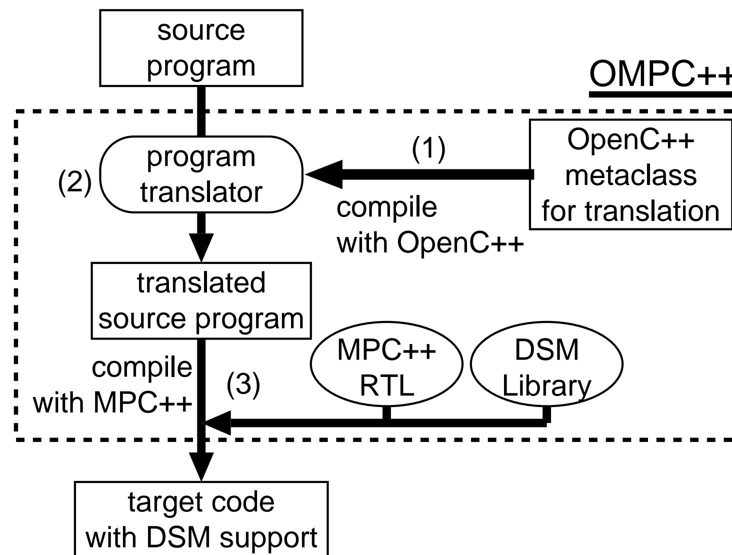


**Fig. 2.** Program Transformation and Compilation

## 3 Implementation of Software DSM in OMPC++

We outline the process of compilation and execution in OMPC++(Fig. 2).

- OpenC++ generates a preprocessor from the OMPC++ metaclasses (1).
- The source program is transformed using the preprocessor (2). The transformed program is a message-passing MPC++ program.
- MPC++ adds compiler variables such as the SCore library, and hands it off to the backend C++, which generates the executable (3).

practicality at the time of OMPC++ development such as stability and compilation speed. In theory we could replicate our work using MPC++ Level 1.

A real compilation session is depicted in Fig. 3. The numbers correspond to those in Fig. 2.

```
% ompc++ -m -- SharableClass.mc                |(1)

% ompc++ -v -- -o cg cg.cc problem.o           |
[Preprocess...  /usr/score/bin/mpc++           |
  -D__opencxx -E -o cg.occ -x c++ cg.cc]       |
compiling  cg.cc                               |(2)
[Translate... cg.occ into: cg.ii]              |
Load SharableClass-init.so.. Load              |
  SharableClass.so.. Initialize.. Done.        |

[Compile...  /usr/score/bin/mpc++              |
                 -o cg cg.ii problem.o]        |
compiling  cg.ii                               |(3)
[done.]                                        |
[GC happened 10 times.]                        |
```

**Fig. 3.** A Compilation Session for OMPC++

### 3.1 Program Transformation

There are 3 steps in the program transformation for implementing the DSM functionality:

1. Initialization and finalization of memory management objects created on each node.
2. Initialization of shared memory regions (creation of management tables, allocation of local memory).
3. Insertion of locks and read/write barriers to shared variables in the program.

The examples of program transformations are depicted in Fig. 4 and Fig. 5[3]. The program transformation possible with OpenC++ is restricted to transformation of base programs, and in particular those of class methods; so, for OMPC++, we assume the user program to conform to the following structure:

```
sdsm_main() { .. };   // main computing code
sdsm_init() { .. };   // application-specific initialization
sdsm_done() { .. };   // application-specific finalization
```

[3] Note that, there are restrictions of the template features in the metaclasses for the current version of OpenC++, forcing us to do some template expansions manually.

sdsm_init() is called following the system-level DSM initialization. sdsm_main() function is the body of parallel code, and is called independently in parallel on each PE after initialization of the shared memory region. sdsm_done() is called after sdsm_main() terminates, followed by system-level DSM finalization, after which the whole program terminates. sdsm_init() and sdsm_done() is sequentially executed on a single PE (normally PE 0), while sdsm_main() is executed on all the PEs.

The following functions are added as a result of program transformation:

```
mpc_main() { .. };        // the main MPC++ function
Initialize() { .. };      // initialization of shared region
Finalize() { .. };        // finalization: freeing of shared region
```

Because the resulting code of program transformation is MPC++ code, the program starts from mpc_main(). This in turn calls Initialize(), which then calls the application-specific sdsm_init(). After the computation completes, sdsm_done() is called, and finally Finalize() is executed on all the processors, and the program terminates.

```
Shared<double> a(100);                          // ..(1)
Shared<double> b;
sdsm_main() {
  b = (double*)malloc(sizeof(double) * 10); // ..(2)
  b[0] = a[0];  ...
};
sdsm_init() { a[0] = 10.0; ... };
sdsm_done() { ... };
```

**Fig. 4.** OMPC++ Program before Transformation

**Initialization and Finalization** The initialization process initializes the memory management object and the DSM objects. These are quite straightforward, and only involve initialization of locks (the MPC++ Sync object). Finalization involves freeing of all the memory regions and the associated management objects, and are also straightforward.

**Initialization of Shared Regions** There are two types of initialization of shared regions, depending on when the memory is allocated by the user. Type 1 is when the size of the shared region (on variable definition) is fixed, and initialization is done in Initialize(). Transformation from Fig. 4-(1) to Fig. 5-(1) is an example. Type 2 is when the user does not specify the size on variable definition, but instead dynamically allocates a memory region within sdsm_init, sdsm_main(); in this case, allocation is done on the spot as we see in Fig. 4-(2) to Fig. 5-(2).

```
Shared<double> a(100);
Shared<double> b;
sdsm_main() { b.allocate(10);              // ..(2)
              { double _sym52501_8 = a[0]; // ..(5)
                b.WriteStart(0);           // ..(3)
                b(0) = _sym52501_8;
                b.WriteEnd(0); } ... };    // ..(3)
sdsm_init() { { a.WriteStart(0);           // ..(4)
                a(0) = 10.0;
                a.WriteEnd(0); } ... };    // ..(4)
sdsm_done() { ... };
Initialize() { a.allocate(100); };         // ..(1)
Finalize() { a.free();
             b.free(); };
mpc_main() {
  invoke Initialize() on all PEs.
  sdsm_init();
  invoke sdsm_main() on all PEs.
  sdsm_done();
  invoke Finalize() on all PEs.
};
```

**Fig. 5.** After Transformation

**Access to Shared (Memory) Regions** For accessing shared regions, lock/unlock are inserted on writes. Fig. 5-(3),(4)). When a shared variable occurs on the RHS expression, writes are first performed to a temporary variable to avoid duplicate lockings (Fig. 5-(5)) on further RHS evaluation. Read access does not require any locks.

### 3.2 Program Transformation Metaclasses

In OpenC++, metaclasses are defined as subclasses of class Class, and by overriding the methods, one could change the behavior of programs. OMPC++ overrides the following three OpenC++ methods:

**TranslateInitializer** Called when the shared class object is created. We can then obtain the name, type, size, and other information of the distributed shared array object, and are used for initialization of the shared regions.

**TranslateAssign** Called when there is an assignment to the shared class object. We can then transform the initialization and the writes of the distributed shared array objects. We analyze the expression which is passed as a parameter by the OpenC++, and if there is an malloc (or other registered allocation functions), then we perform similar task as the TranslateInitializer to obtain the necessary information, and if it is within sdsm_main(), the initialization code for shared regions directly replaces the allocation. When a shared variable occurs on the RHS expression, TranslateAssign generates a

new statement which assigns it to a temporary variable. For assignments to shared variables, their operators `[]` are transformed to operators `()`, and the WriteStart() and WriteEnd() methods are inserted to sandwitch the assignment.

**FinalizeInstance** Called upon the end of transformation. Here, we insert the initialization and finalization functions discussed earlier. More concretely, we generate Initialize(), Finalize() based on the information obtained with TranslateInitializer and TranslateAssign, and also generate mpc_main(), obtaining the whole program.

We illustrate a sample metaprogram of TranslateAssign in Fig. 6. Here, we sandwitch the assignment with lock method calls. Metaprograms are quite compact, with only 250 lines of code (does not include runtime as well as template code, etc.).

```
Ptree* SharedClass::
  TranslateAssign(Environment* env,
    Ptree* obj, Ptree* op, Ptree* exp){
      Ptree* exp0 = ReadExpressionAnalysis(env, exp);
      Ptree* obj0 = WriteExpressionAnalysis(env,obj);
      return Ptree::List(obj0, op, exp0);  };
Ptree* SharedClass::
  WriteExpressionAnalysis(Environment* env, Ptree* exp){
    Ptree *obj = exp->First();
    Ptree *index = exp->Nth(2);
    if (!index->IsLeaf())
      index = ReadExpressionAnalysis(env, index);
    InsertBeforeStatement(env,
      Ptree::qMake("‘obj‘.WriteStart(‘index‘);\n"));
    AppendAfterStatement(env,
      Ptree::qMake("\n‘obj‘.WriteEnd(‘index‘);\n"));
    return Ptree::qMake("‘obj‘(‘index‘)");  };
```

**Fig. 6.** Metaprogram Example

### 3.3 Distributed Shared Memory

OMPC++ does not implement DSM transformation with OpenC++ compile-time MOP alone; rather, it also utilizes C++ templates and operator overloading. Also, in OMPC++, read/write barriers are performed in software, instead of (traditional) hardware page-based strategies such as TreadMarks[ACD$^+$96]. Although such checks are potential sources of overhead, they provide the benefit of maintaining the coherency blocks small, avoiding false sharing. Recent work in Shasta[SGT96] has demonstrated that, with low-latency networks, software-based checks do not incur major overhead, even compared to some hardware

**Table 1.** Entry of the management table

| | |
|---|---|
| `addr` | pointer into the real (local) memory (4) |
| `copyowner` | pointer to a list of PEs holding a copy (4) |
| `owner` | the block owner (2) |
| `copyowers` | # of PEs with copies (2) |
| `havedata` | A flag indicating whether data is present (1) |
| `lock` | Lock (1) |
| `dummy` | Padding (2) |

() indicates the number of bytes for the field

based DSMs. Moreover, for portability, software-based checks are substantially better than paged-based checks, as the latter would incur adapting to differing APIs and semantics of trapping page faults and upcalling into the user code for a variety of operating systems.

We note that, with languages such as Java which does not have templates, more program transformation responsibility will be delegated to metaprogramming. It would be interesting to compare the real tradeoffs of the use of templates versus metaprograms from the perspective of performance, code size, ease-of-programmability, etc.

**Distributed Shared Array Class** Distributed Shared Array Class is a template class written in MPC++. The class implements a shared array as the name implies, and objects are allocated on all the PEs. For the current version, the memory consistency protocol is write-invalidate with sequential consistency, and weaker coherence protocols and algorithms such as LRC[Kel94] are not employed. The array elements are organized in terms of consistency block size, which is the unit of memory transfer among the nodes. The size of the block can be specified with `BlockSize`, and can be of arbitrary $2^n$ size. The management tables of the block resides entirely within the class. Each entry in the table is as shown in Table 1, and is aligned to 16 bytes to minimize the cost of address calculations. When the memory allocation method `allocate`(size) is called, (size/`BlockSize`) entry management table is created, and each PE allocates (size/#PE) memory for storage (excluding the copies).

**Read Access Processing** Because we employ the array access operator `[]`, we overload the `[]` operator in the distributed shared array class. The overloaded behavior is as follows:

1. If the `havedata` flag is true, then return that address.
2. If the `havedata` flag is not true, then allocate the necessary copy block, and request for copying of the block contents to the block owner by passing the MPC++ global pointer to the owner.
3. The requested owner of the block remotely writes the contents of the block onto the copy block pointed to by the global pointer.
4. After the write is finished, the local address of the copy block is returned.

**Write Access Processing** The write access overloads the () operator, but recall that the OMPC++ metaclasses have inserted the lock methods WriteStart() and WriteEnd(). The overloaded behavior is as follows, for write invalidation. Other protocols could be easily accommodated.

1. WriteStart() sets the `lock` flag of the corresponding management table entry. If it is already locked, then wait.
2. If the PE is the block owner, and there are copies elsewhere, then issue invalidation messages to the PEs with copies.
3. If the PE does not have the block data, then request for a copy in the same manner as the read.
4. If the PE is not the block owner, then transfer the ownership. The previous owner notifies to the other PEs that the ownership change has occurred.
5. The operator () checks the `havedata` flag, and returns its address.
6. WriteEnd() resets the `lock` flag.

**Optimizations** Minimizing software overhead is essential for the success of software DSM algorithms. Upon analysis, one finds that the overhead turns out to not be caused by invalidation, but primarily due to the reads/writes which require address translation and barrier checks. In OMPC++, we optimize this in the following way. For accessing the management table entries, we have made the entry size to be $2^n$ so that shifts could be used instead of multiplications and divisions. The overloaded [] and () operators as well as WriteStart() and WriteEnd() are inlined. Moreover, the block address computed in the last access is cached; as a result, if one accesses the same block repeatedly (which is the assumed action of "good" DSM programs in any case), read/write access speed are substantially improved.

One could further define sophisticated metaclasses to perform more through analysis and program transformation to physically eliminate coherency messages, as is seen in [LJ98]. As to whether such analysis are easily encodable in terms of compiler metaclasses of OpenC++ is an interesting issue, and we are looking into the matter.

## 4 Performance Evaluation

The current version of OMPC++ is approximately 700 lines of code, which consists of 250 lines of metaclass code, and 450 lines of template runtime code. This shows that, by the use of reflection, we were able to implement a portable DSM system with substantially less effort compared to traditional systems.

In order to investigate whether OMPC++ is sufficiently fast for real parallel programs, we are currently undertaking extensive performance, analysis, pitting OMPC++ with other software-based DSM algorithms, as well as large-scale SMP systems such as the 64-node Enterprise 10000. Here we report on some of our preliminary results on the PC cluster, including the basic, low-level benchmarks, as well as some parallel numerical application kernels, namely the CG kernel, and the SPLASH2[WOT+95] FFT kernel and LU.

### 4.1 Evaluation Environment

Evaluation environment is a small subset of the PC Cluster II developed at RWCP (Real-World Computing Partnership) in Tsukuba, Japan. The subset embodies 8–10 200MHz Pentium Pro PC nodes with 64MB memory (of which 8 is used for benchmarking), and interconnected by the Myrinet Gigabit network (LinkSpeed 160MB/s). The OS for this system is the Linux 2.0.36-based version of SCore, and uses MPC++ Version 2.0 Level 0 MTTL as mentioned. The underlying compiler is pgcc 1.1.1, and the optimization flags are "-O6 -mpentiumpro -fomit-frame-pointer -funroll-loops -fstrength-reduce -ffast-math -fexpensive-optimizations". For comparative environment, we use the Sparc Enterprise Server 4000, with 10 250MHz UltraSparc II/1Mb, and 1Gb of memory. The Sun CC 4.2 optimization options are "-fast -xcg92 -xO5".

### 4.2 Basic Performance

As a underlying performance basis, we measured the read/write access times. For access patterns to shared arrays, we measured continues access, strided access, and write access with invalidation.

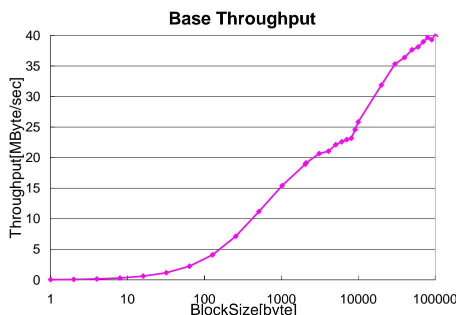


**Fig. 7.** Throughput of basic remote memory operations

**Continuous Read/Write Accesses** We measured the total read/write access times of size 1024×1024 double shared array. All the blocks are setup so that they reside on other PEs; in other words, none of the blocks are initially owned or cached. For reads, this obviously means that accesses must first start by fetching the copies. For writes, since none of the blocks are cached on any of the PEs, write access does not involve invalidation, and thus most of the incurred overhead is remote writes.

We show the averaged times of a single access for different block sizes in Fig. 8. For both reads and writes, average access time decreases with increased

block size. This is naturally due to amortization of message handling and lock processing overhead. When block size increases further, we see a falloff due such amortization has mostly making the overhead negligible, and the speed is primarily determined by the network bandwidth. For `BlockSize` 64kbytes, it is 0.34$\mu$sec/1dword, which corresponds to 23.5Mbytes/sec. By comparison, raw MPC++ RemoteMemoryRead is approximately 40Mbytes/sec (Fig. 7). The difference here is the cost of address translation and barrier on the read operation.

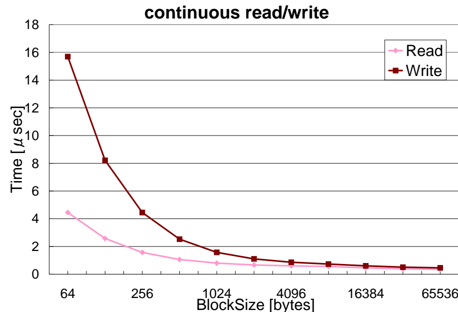


**Fig. 8.** Cost of Continuous Reads and Writes Accesses

**Strided Read/Write Accesses** Strided Access is a worse-case scenario where the arrays are serially accessed at a stride, and the stride is equal to `BlockSize`. As a result, no cached access can occur, but rather the entire block must be transferred per each memory access. The size and the initialization of memory is identical to the continuous access. However, the number of elements accessed would differ according to the stride. The results are shown in Fig. 9. As we can see, the access times increase along with the increase in `BlockSize`. This is due to increased memory transfer per each access. Also, the difference between read and write diminishes; this is because the transfer times become dominant, despite using a very fast network, as opposed to software overhead such as address translation and locking.

**Accesses with Invalidation** Because we currently employ the standard write-invalidate algorithm, the read cost will remain constant, while the write cost could increase as the number of shared copies of the block increases, as invalidate message has to be broadcast to all the PEs holding the copy. In order to measure the overhead, we varied the number of PEs with copies to 0,1,2,6, and the `BlockSize` to 1kbytes and 16kbytes, whose result is shown in Table 2. Here, we observe that for such small number of sharings, the differences are negligible. This is because message passing of MPC++ on SCore is physically

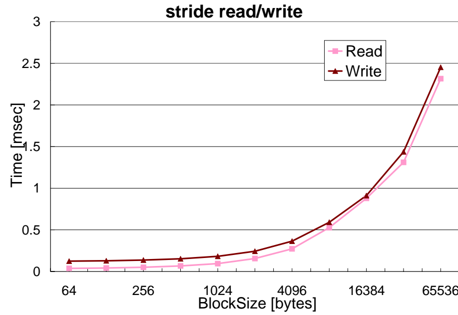


**Fig. 9.** Cost of Strided Read and Write Accesses

asynchronous, and as a result, multiple invalidations are being effectively issued in parallel. Also, access times is independent of block size, as invalidation does not transfer any memory, so its cost is independent of the `BlockSize`.

One might argue that for a larger number of sharing, parallelism in the network interface would saturate, resulting in increased cost. While this is true, research has shown that average number of sharing for invalidation-based protocol is typically below 3. This is fundamental to the nature of the write-invalidate algorithm, as shared blocks are thrown away per each write.

**Table 2.** Cost of Write Accesses with Invalidation (`BlockSize`=1kbytes,16kbytes)

| Sharings (#PEs) | Time($\mu$sec) `BlockSize` 1kbytes | `BlockSize` 16kbytes |
|---|---|---|
| 0 | 4.80 | 5.26 |
| 1 | 31.27 | 33.05 |
| 2 | 46.06 | 48.27 |
| 6 | 76.68 | 78.24 |

### 4.3 CG (Conjugate Gradient Solver Kernel)

As a typical parallel numerical application, we employ the CG (Conjugate Gradient) Solver Kernel, without preconditioning. As a comparative basis, we execute the equivalent sequential program on one of the nodes of the PC cluster, and also on the Sun Ultra60(UltraSPARC II 300MHz, 256Mbytes, Sun CC 4.2). Here are the parameters for the measurements:

– Problem sizes: 16129 (197 iterations), 261121 (806 iterations)

– BlockSizes: 256, 1k, 4k, 16kbytes
– Number of PEs: 1, 2, 4, 8

The results are shown in Fig. 10 and Fig. 11.

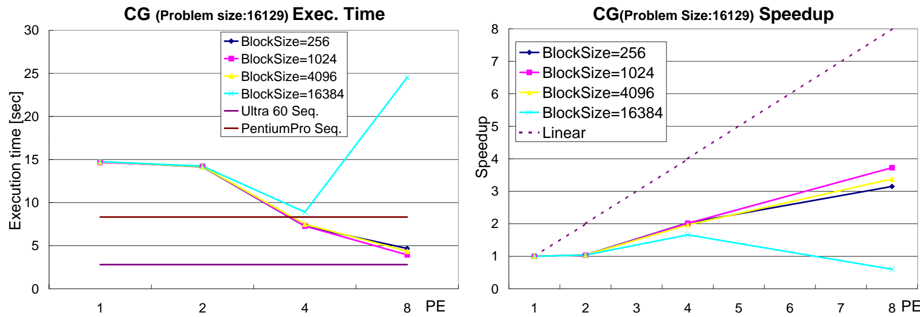


**Fig. 10.** CG Kernel Result, Problem Size=16129

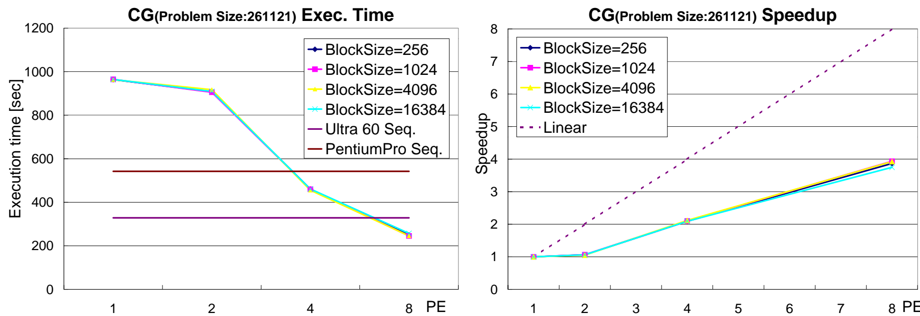


**Fig. 11.** CG Kernel Result, Problem Size=261121

According to the results of Fig. 10, we obtain only about 5% speedup from 1PE to 2PEs, whereas we observe approximately linear speedups from 2PEs to 8PEs. This is primarily due to the software overhead of DSM, in particular write-invalidation does not occur for 1PE, whereas they do occur for 2PEs and above. Even with 8PEs, we do not match the speed of sequential UltraSparc II 300MHz, which has much faster floating performance than a Pentium Pro 200MHz. We do attain about factor of 2 speedup over a sequential execution on a cluster PC node. For `BlockSize` of 16KB, we see that performance degrades significantly. This is due to the overhead of excessive data transfer.

With larger problem size as in Fig. 11, the overall characteristics remain the same. We still see speedups of range 3.8 to 4.0 with 8 processors. Here, `BlockSize` is almost irrelevant for overall performance; this is because, as the amount of data required for computation is substantially larger than `BlockSize`, and thus we do not have wasteful internote data transfers. On the other hand, compared to overall computation, the software overhead of message handling is negligibly small.

We also executed the same program on the Sun Enterprise 4000 (UltraSPARC II 250MHz, 10PEs). There the speedup was approximately 6 with 8 processors. The overhead is primarily due to barrier operations.

### 4.4 SPLASH2 FFT and LU

Finally, we tested the SPLASH2 [WOT+95] FFT and LU kernel (Fig. 12) with 1kbytes BlockSize. In FFT, the problem size is 64Kpoints. In LU, the matrix size is 512.

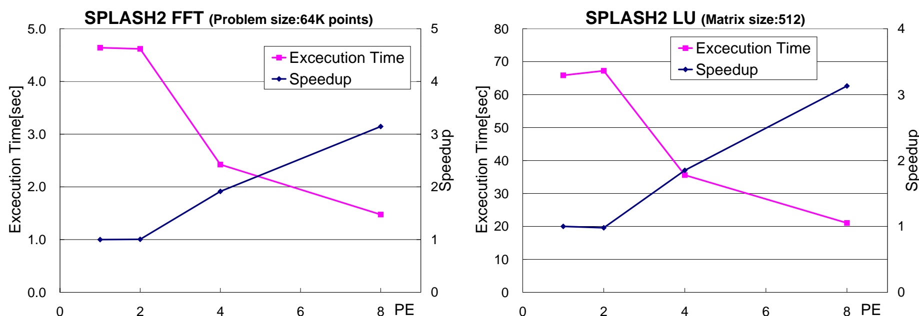


**Fig. 12.** SPLASH2 FFT and LU

As was with the CG kernel, speedup curves are similar. The required change to the original SPLASH2 was approximately 70 lines out of the 1008 in FFT, 80 lines out of the 1030 in LU, mostly the required top-level program structure (with sdsm_main, etc.) and memory allocations.

Shasta employs various low-level optimization strategies, that have not been implemented on OMPC++, such as bunching multiple read-/write-accesses. As an experiment, to qualify the effect of bunching technique, we manually applied it to the `daxpy` function in the LU kernel on OMPC++, attaining about factor of 2 speedup in terms of the total execution time(Fig. 13).

## 5 Related Work

There are several work on implementing DSM features using program transformations [SGT96]. All of them have had to develop their own preprocessor,

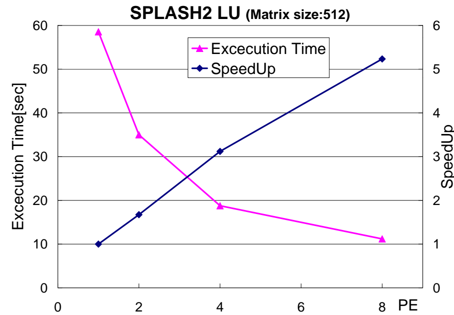


**Fig. 13.** SPLASH2 LU with bunched read-/write-accesses

and is thus expensive, and furthermore not portable nor adaptable to complex programming languages such as C++. By utilizing the reflective features of OpenC++, we have demonstrated that, developing a program transformer for an efficient DSM system was an easy effort with a small number of metaclass customizations, much easier to maintain and tune the code. In fact, substantial tuning and experimentation was possible by small changes to the metacode.

Shasta [SGT96] is a DSM system that performs program transformation at program load time, and is thus more language independent than other program transformation systems. Shasta is specifically tailored to work on DEC Alpha PEs interconnected by the MemoryChannel low-latency gigabit network. Shasta is extremely efficient, in that it optimizes the read/write barrier overhead for common cases down to two instructions. This is currently possible because Shasta performs program transformation at the binary level. On the other hand, Shasta is much harder to port to other platforms, as it is hardware and processor-binary dependent. There are numerous low-level optimization techniques that Shasta uses, that have not been implemented on OMPC++. We are planning to exploit such techniques with more metaprogramming, with the aid of better optimizing compilers such as Kai C++.

Midway[BZS93] and CRL[JKW95] are object-based DSM systems. Midway employs entry consistency as the coherency protocol, and read/write barriers are inserted by a special compiler. Thus, one must craft such a compiler, and as such portability and extensibility suffers. For example, it is difficult to add a new coherency protocol or porting to different hardware platform without detailed knowledge of the compiler. On the other hand, in CRL, the barriers must be inserted by the user. This is quite error-prone, and such bugs would be very hard to detect, as they will be dynamic, transient memory bugs. By using reflection, OMPC++ was created by only small set of customization in the underlying workings of the C++, and is easily understandable, thus being easily maintainable, customizable, and portable.

As a common shared memory programming API, OpenMP[Ope97] has been proposed as a standard. In OpenMP, the user adds appropriate annotations

to the source code to indicate potential SPMD parallelism and distribution to sequential code, and the compiler emits the appropriate parallel code. OpenMP suffers from the same problem as Midway, as customized preprocessor/compiler must be built. It would be interesting to investigate whether OpenMP (or at least, its subset) could be implemented using reflective features to indicate DSM optimizations.

In perspective, with respect to the maintainability, customizability, and portability of DSM systems,

- Traditional page-based DSM systems must have OS-dependent code to trap memory reference via a page fault. Also, overhead is known to be significant.
- Link-time DSM systems such as Shasta is efficient, but is platform and processor-binary dependent.
- Class-based systems requires the user to insert appropriate read/write barriers, and is thus error prone and/or only a certain set of classes can be shared.
- Macros or ad-hoc preprocessor are hard to build and maintain, let alone customize or be portable, and complete support of complex programming languages such as C++ is difficult.

## 6 Conclusion and Future Work

We have described OMPC++, a portable and efficient DSM system implemented using compile-time reflection. OMPC++ consists of a small set of OpenC++ 2.5 metaclasses and template classes and is thus easy to customize and port to different platforms. OMPC++ is a first step to realization of reflective framework for portable and efficient support of various parallel programming models across a wide variety of machines. Such characteristics are necessary as increasingly in the days of network-wide computing, where a piece of code could be executed anywhere in a diverse network.

OMPC++ was shown to be efficient for a set of basic as well as parallel numerical kernel benchmarks on the RWC PC cluster platform.

As a future work, we are pursuing the following issues:

**More comprehensive benchmarking** The number of benchmarked programs, processors, and comparative platforms, are still small. We are currently conducting a set of comprehensive benchmarks by adding more SPLASH2 benchmarks (Water, Barnes), on larger platforms (64-processor RWCP cluster and 64-processor Sparc Enterprise Server 10000), pitting against other DSM platforms (developed at RWCP). We are also attempting alternative algorithms, both for coherency and barriers. We will report the findings in another paper after more analysis is done.

**More efficient implementation** Although OMPC++ went under some tuning process, the read/write barriers and locks are still expensive. Although in our preliminary benchmarks we are finding that OMPC++ is competitive with other DSM implementations, we need to further enhance efficiency. For example, Shasta only requires two instructions for barriers, much

smaller than ours, and employs various low-level optimization strategies such as bunching multiple read-/write-accesses. Also, raw MPC++ RemoteMemoryRead throughput is 40MByte/sec, whereas OMPC++ throughput is 23.5MByte/sec. This is due to address translation and table accesses. We are considering altering the runtime data structures to further eliminate the overhead.

**Porting to other platforms** OMPC++ currently is implemented on top of MPC++ and RWC cluster. In order to port to different platforms, one must alter the dependency on MPC++, if it is not available. In principle this is simple to do, as one must only provide (1) threads, (2) remote memory read/write, and (3) global barriers. This can be easily implemented using MPI or other message passing platforms such as Illinois Fast Messages. Alternatively one could make MPC++ to be more portable, which is our current project in collaboration with RWC. In all cases, we must analyze and exhibit the portability and efficiency of OMPC++ to validate that implementing DSM systems with reflection is the right idea.

**Other coherency protocols** On related terms, we should support other coherency protocols, such as write-update, as well as weak coherency models such as LRC and ALRC. The latter will fundamentally require changes to the source code, and it would be interesting to investigate how much of this simplified using open compilers.

**Portable DSM on Java using OpenJIT** Finally, we are planning to implement a Java version of DSM using OpenJIT, our reflective Java Just-In-Time compiler. As of Feb. 1, 1999, We have almost completed the implementation of OpenJIT, and will start our implementation as soon as OpenJIT is complete.

## Acknowledgment

We would like to thank Shigeru Chiba for his support in the use of OpenC++, and the members of RWCP for their support in the usage of MPC++ plus the RWC cluster. Also, we would like to thank the Dept. of Information Science, the Univ. of Tokyo for allowing experiments on their Sparc Enterprise Server 10000. We finally thank the members of the Tsukuba TEA group for fruitful discussions on performance analysis of SPMD and DSM systems.

## References

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Warkstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[Beo] Beowulf Project. Beowulf Homepage. http://beowulf.gsfc.nasa.gov/.

[BZS93] Brain N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdown. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE CompCon Conference*, 1993.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, pages 285–299, 1995.

[Chi97] Shigeru Chiba. *OpenC++ 2.5 Reference Mannual*, 1997.

[Ie96] Yutaka Ishikawa and et.al. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Proceedings of Reflection'96*, Apr. 1996.

[Ish96] Yutaka Ishikawa. Multiple Threads Template Library – MPC++ Version2.0 Level 0 Document –. TR 96012, RWCP, 1996.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth Symosium on Operating Systems Principles*, Dec. 1995.

[Kel94] Pete Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Dept. of Computer Science, Rice University, Dec. 1994.

[LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov 1989.

[LJ98] Jae Bum Lee and Chu Shik Jhon. Reducing Coherence Overhead of Barrier Synchronization in Software DSMs. In *SC'98*, Nov. 1998.

[MOS+98] Satoshi Matsuoka, Hirotaka Ogawa, Kouya Shimura, Yasunori Kimura, and Koichiro Hotta. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, Oct. 1998.

[Myr] Myricom, Inc. Myricom Homepage. http://www.myri.com/.

[Ope97] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming.*, Oct. 1997.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of ASPLOS VII*, Oct. 1996.

[Uni] Stanford Univ. SUIF Homepage. http://www-suif.stanford.edu/.

[WOT+95] 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 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.