PAT: A Postmortem Object Access Pattern Analysis and Visualization Tool*

Weijian Fang Cho-Li Wang Wenzhang Zhu
Francis C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
{wjfang+clwang+wzzhu+fcmlau}@csis.hku.hk

Abstract

Applying a cache coherence protocol capable of adapting to memory access patterns is a viable approach to improving the performance of software distributed shared memory. In this paper, we present an approach of postmortem memory access pattern analysis and visualization, which has been applied to our design of a global object space for a distributed Java Virtual Machine. The tool not only can enhance our understanding of the access patterns inherent in an application but can also help us to evaluate the effectiveness of an adaptive protocol used in the design of the global object space.

1. Introduction

The development of efficient cache coherence protocol for software DSM [1] for today's high-performance parallel computers is a nontrivial task as a good protocol needs to be able to take full advantage of the computer system's deep memory hierarchy and to adapt quickly to the dynamically changing memory access patterns in a networked environment.

In the past, much effort has been spent on improving the performance of DSM systems using adaptive cache coherence protocols [5, 14, 3, 2, 13, 8]. It was claimed that these advanced protocols have the ability to detect or predict the memory access patterns, and are capable to switch between different schemes for maintaining cache coherence without compiler's involvement. The adaptability of those protocols was considered from a macroscopic view, and reasoned based on the application's algorithm. For example, a common report would present the overall reduction in execution time or communication overheads after the adaptation is enabled. This somehow cannot reflect the true effectiveness of adaptive protocols. From such figures, we do not know if the runtime optimization has been applied at the appropriate moments, or if there are still some other memory access patterns that were not accounted for.

To better understand an adaptive protocol's efficiency, a memory access profiling and visualization tool can help the designer to compare memory access and communication patterns of different algorithms and gain insight into the application's behavior such as potential bottlenecks resulting from memory accesses.

In this paper, a visualization tool called PAT (Pattern Analysis Tool) is proposed that can be used to analyze memory access patterns in object-oriented software DSM systems. PAT adopts a lightweight mechanism to record various kinds of memory access events (e.g., synchronization operations) associated with runtime objects. Users can play back these events and perform postmortem object-level analysis following an application's execution.

PAT is useful in two aspects. For the protocol designers, such a tool can expose the inherent memory access patterns inside a benchmark application, and thus enable evaluation of the effectiveness of the adaptive protocol in reducing the number of network-related memory operations and the protocol's pattern detection mechanism. It can reveal how frequent a particular memory access pattern appears in an application, and how well a particular adaptation can optimize a target memory access pattern.

On the other hand, it can help the application developer in planning out the initial data layout and runtime data relocation. Since DSM systems tend to hide the communication details from application developers, performance tuning is rather difficult if not impossible. With PAT, the parallel application developer is able to discover the performance bottleneck in the application by observing the application's memory access behavior. He may then redesign the algorithm to avoid some heavy-weight memory access patterns.

We demonstrate the potential power of PAT by applying it to the development of adaptive cache coherence protocol for the *global object space* (GOS) support in a *dis-*

* This research is supported by Hong Kong RGC grant HKU-7030/01E and HKU Large Equipment Grant 01021001.

tributed Java Virtual Machine (JVM) [8] that runs on a cluster. The GOS provides the illusion of a single Java object heap spanning multiple cluster nodes to facilitate transparent object accesses issued by Java threads in different nodes. As a DSM service in an object-oriented system, our GOS is marked by its adaptability to object-level access patterns.

PAT comprises three components: the *object access trace generator* (OATG) that is plugged into the distributed JVM, the *access pattern analysis engine* (APAE), and the *pattern visualization component* (PVC).

PAT gathers object access information at runtime. Improper runtime logging could introduce intolerable overhead and interruption to the application being traced, which makes the logging unaccepted. For example, the recorded memory access behavior could be quite different from that without logging due to the interruption caused by heavyweight logging. To tackle this problem, OATG was designed to be lightweight. It leverages the Java memory model and the just-in-time compiler in distributed JVM to minimize the logging overhead. It activates the recording only on distributed shared objects. Logs are stored in a memory structure and flushed to the local disk at synchronization points or when the buffer is full. The just-in-time compiler is used to instrument only the user-interested methods; all the other methods execute at full speed.

APAE is used to discover knowledge concerning patterns from the raw access information collected by OATG. After an application's execution, the global (of all the processes) and complete (the entire lifetime of the application) access information can be compiled, based on which an analysis of the object access patterns is carried out precisely and thoroughly.

APAE uses a pattern-centric representation to visualize object access patterns. It can display the global and complete access information in a macroscopic view. In addition, for objects of interest to the user, it can associate access patterns with the source code lines that create the corresponding objects—referred to as allocation sites. The object access patterns can be further mapped to low-level object access events.

The rest of this paper is organized as follows. Section 2 gives the background of this research and an overview of PAT. Section 3 presents the lightweight runtime object access trace generation mechanism in PAT for collecting memory access information. Section 4 discusses our design to precisely and thoroughly analyze object access patterns given the object access trace. Section 5 presents the pattern-centric visualization part of PAT. Section 6 discusses the related work. Section 7 summarizes our work and gives a possible agenda for future work.

2. Background

In this section, we discuss an object-oriented software DSM, called the *global object space* (GOS) [8]. The GOS has been used to support transparent object access in a *distributed Java Virtual Machine* (JVM) running on a cluster. We will also give an overview of PAT.

2.1. Global Object Space

The Java programming language [4] supports concurrent programming with multiple threads, which makes it a potential language for parallel computing. A distributed JVM appears as a middleware that presents a *single system image* of the cluster to Java applications. With a distributed JVM, the Java threads created within one program can be run on different cluster nodes to achieve a higher degree of execution parallelism.

In a distributed JVM, the shared memory nature of Java threads calls for a *global object space* (GOS) that "virtualizes" a single Java object heap spanning the entire cluster to facilitate transparent object access. The GOS is indeed a DSM service in an object-oriented system. The memory consistency semantics of the GOS are defined based on the Java memory model (Chapter 8 of the JVM specification [11]), in a fashion that is similar to *lazy release consistency* [10].

In our previous work [8], we proposed a new global object space design for the distributed JVM. In the design, we use an object-based adaptive cache coherence protocol to implement the Java memory model.

2.2. Memory Access Operations Classification

Figure 1 shows all the memory access operations in the GOS. We divide Java objects in the GOS into two categories: *distributed-shared objects* and *node-local objects*. Distributed-shared objects are reachable from at least two threads in different cluster nodes in the distributed JVM, while node-local objects are reachable from only one cluster node. Distributed-shared objects can be distinguished from node-local objects (by default) at runtime [8].

Since we use a home-based multiple writer cache coherence protocol to implement Java memory model that resembles lazy release consistency, the access events of a distributed-shared object comprise those on non-home nodes and those on the home node. On non-home nodes, after acquiring a lock, the first read should fault in the object from its home. All the subsequent reads or writes can be performed on the local copy. Before acquiring or releasing a lock, the locally performed writes should be identified using *twin* and *diff* techniques and sent to the home node. We

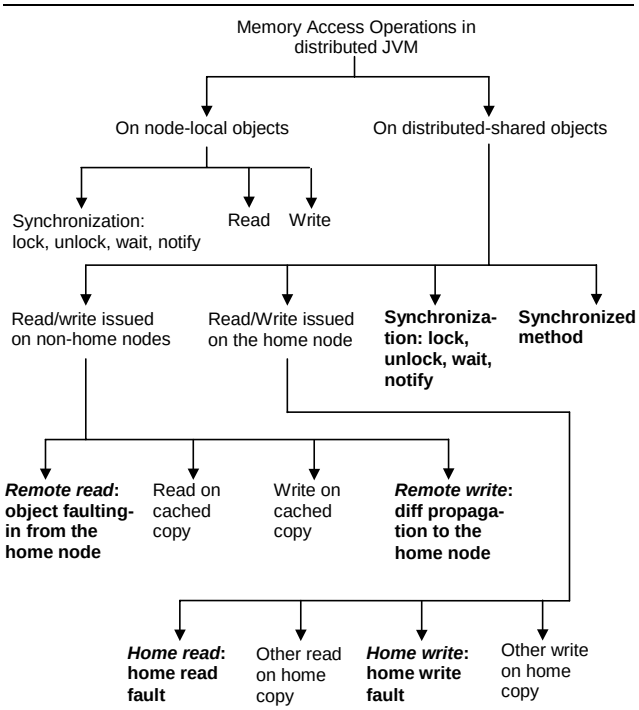


Figure 1. Memory access operations in GOS

call the object faulting-in *remote read*, and the diff propagation *remote write*.

On the home node, the access state of the home copy will be set to invalid on acquiring a lock and to read-only on releasing a lock. *Home read fault* and *home write fault* will be trapped. For both types of fault, the GOS does nothing more than to set the object to the proper access state. We call the home read fault *home read*, and the home write fault *home write*.

All the synchronization operations performed on a distributed-shared object, such as lock, unlock, wait, and notify, influence the object access pattern, and are thus considered access events too. The synchronized method may be migrated to the object's home node by the GOS as a whole, and is also treated as an access event.

All these memory access events can make up of various memory access patterns. In our previous work [8], we propose an access pattern space to enumerate various patterns. According to the number of writers, we identify three cases:

- *Multiple writers*: the object is written by multiple nodes.
- *Single writer*: the object is written by a single node.
- *Read only*: no node writes to the object.

According to our experience, the single-writer pattern is significant in many scientific applications. So our GOS incorporates an *object home migration* optimization that improves the performance of the single-writer pattern by setting the sole writing node to be the home node, which has been proven to be very useful.

2.3. Overview of PAT Architecture

Figure 2 shows the architecture of PAT. PAT comprises three components: the *object access trace generator* (OATG), the *access pattern analysis engine* (APAE), and the *pattern visualization component* (PVC).

We make use of some source code of the logging facility in MPE (Multi-Processing Environment of MPICH) [6] for collecting the access logs. However, our logging facility does not require MPI support during logging. It is implemented as a library and linked against the distributed JVM. At runtime, each process of the distributed JVM independently generates its own log. The log records are firstly put into the local memory and then dumped to the local disk at synchronization points or when the memory buffer is full. After the multi-threaded Java program exits, an MPI program will merge all those local logs into one log file according to the time stamps. We rely on the Network Time Protocol (NTP) [12] to synchronize the computer times on different cluster nodes. The time offset between cluster nodes can be adjusted to less than one millisecond. On merging the node local logs, the time stamps will be further tuned by calculating the current time offset. APAE will do the pattern analysis and PVC will provide a visualization of the memory access behavior and object level access patterns of the application. Some details of these three components will be presented in the ensuing sections.

3. Lightweight Runtime Access Trace Generation

PAT uses several techniques to achieve the lightweight runtime logging of memory access information.

Firstly, it relies on the Java memory model to carefully choose the memory access events to be logged. In figure 1, only those access types in bold font are logged.

In the GOS, we focus on distributed-shared objects since only they will incur communication overhead. Consequently, we are only interested in the access patterns presented by the distributed-shared objects. On non-home nodes, the object faulting-in and diff propagation can represent the reads and writes on the cached copy, respectively. Similarly, the home read fault and home write fault can represent all the reads and writes happening in the home node, respectively. All these remote and home reads/writes, together with synchronization operations on object and synchronized methods, constitute the object's access behavior.

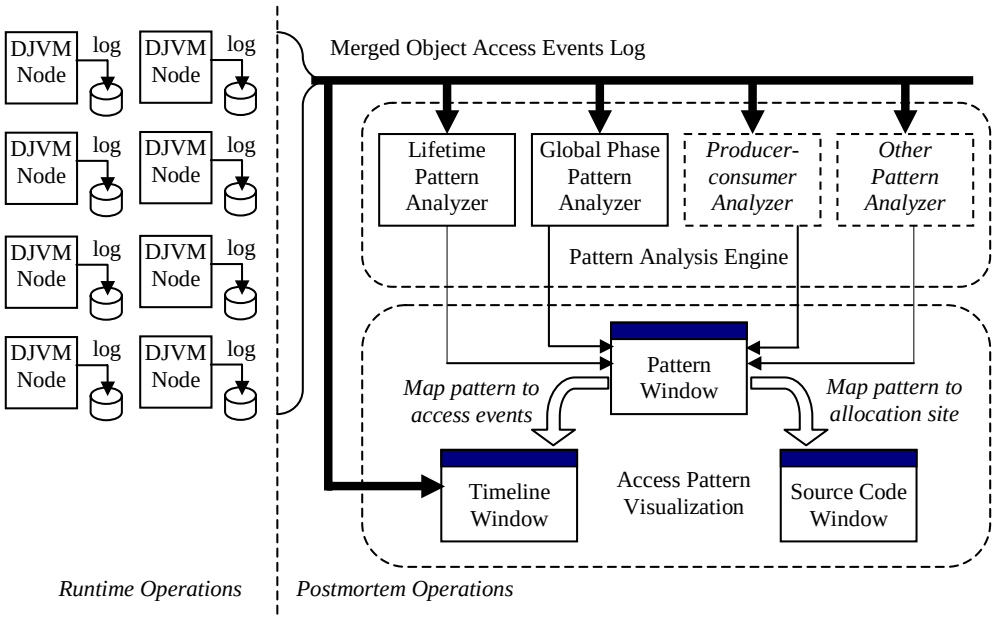


Figure 2. PAT Architecture

All these interested access operations will go through the cache coherence protocol. So we instrument the protocol to intercept and log them.

PAT leverages the just-in-time compiler in a distributed JVM to dynamically instrument translated Java method code to log interesting events. Usually we are interested not only in those access events themselves, but also the relationship between them and other program states. For example, we may want to know how the object access behavior looks like inside a Java method. Or we may want to log a particular method that implements barrier synchronization among all threads to observe the object access events against the barrier synchronization.

PAT allows the user to provide a list of interested Java method signatures[1] to the distributed JVM. During just-in-time compilation, the signature of the to-be-translated method will be compared against the user provided list. If there is a match, the just-in-time compiler will insert the log code at both the start and the end of the method. In doing so, the user is able to choose his interested method events to log. All the other methods are left untouched and operate at full speed. If the just-in-time compiler is not used, we have to do the instrumentation for each method in advance since each method could potentially be a user-interested event. The overall slowdown could be significant.

Table 1 reports the logging overhead in two multi-threaded Java applications, ASP and SOR. ASP computes the shortest paths between any pair of nodes in a graph of 512 nodes using a parallel version of Floyd's algorithm. SOR performs red-black successive over-relaxation on a 2-D matrix (2048 x 2048) for 30 iterations. Both of them run on 8 cluster nodes. In the table, the second column shows the application's execution time when the logging is disabled, and the third column shows the time when the logging is enabled. From the table, we can see that although the generated log size is quite large, the slowdown caused by the logging is still insignificant due to our lightweight logging mechanism.

	Time w/o log	Time w/ log	Slowdown	Log file size
ASP	6.754	6.865	1.64%	33,030,144
SOR	20.421	20.704	1.39%	16,908,288

Table 1. Log overhead (The time is in seconds and the log file size is in bytes.)

4. Access Pattern Analysis

We observe the diversity of object access patterns and the complexity to detect them. Therefore, we propose an extensible design for the pattern analysis engine.

There can be many independent modules sequentially reading in the same log in the analysis engine. Each module is responsible for detecting one or several related ac-

1 The format of Java method signature is defined in the JVM specification.

cess patterns. The access pattern analysis results from all the modules are fed into the pattern visualization component, which will be discussed in the next section. It is extensible in the sense that we can plug in new modules to detect any precisely defined access patterns. Currently there are two analysis modules already in place: the lifetime pattern analyzer and the global phase pattern analyzer.

The lifetime pattern analyzer detects object access pattern that is fixed in the whole lifetime for each distributed shared object. It will check whether an object presents read-only, single-writer, or multiple-writers pattern in its whole lifetime.

The global phase pattern analyzer works for those applications adopting an *phase parallel* paradigm (Section 12.1.1 of [9]). In this paradigm, every thread does some computation before arriving at a barrier. After all the threads arrive at the barrier, they can continue to the next computation phase. Two consecutive barriers define a *global synchronization phase* agreed by all threads. This is a very common paradigm in parallel programming. Both ASP and SOR belong to this paradigm. The global phase pattern analyzer will check whether an object presents read-only, single-writer, or multiple-writers pattern in each global synchronization phase. The barrier, as a synchronized Java method, will be logged as a special event at runtime. If the application does not present phase parallel paradigm, i.e. no barrier events are found in the log, the global phase pattern analyzer simply ignores the log. The advantage of the global phase pattern analyzer over the lifetime pattern analyzer is that pattern changes among different phases can be detected. Detecting read-only, single-writer, and multiple-writers patterns in the log is simply done by counting the number of writers among all the accesses on the targeting object during the lifetime or each phase.

Detecting the *producer-consumer* pattern is also straightforward. By definition, the producer-consumer pattern, which is also known as *single-assignment*, obeys the precedence constraint that the write must happen before the read. In the producer-consumer pattern, after the object is created, it is written and read only once, and then turned into garbage. In order to detect the producer-consumer pattern, we need to check whether the objects are only remotely read once by each thread. Note that the write on the objects could happen before the objects become distributed-shared, and thus has not been logged. In the pattern detection, the difference between producer-consumer and read-only is that the producer-consumer pattern contains only one read while the read-only pattern contains multiple reads.

If the applications do not follow the phase parallel paradigm, i.e., they use lock and unlock instead of barrier synchronization to establish only partial ordering between the accesses from different threads, *accumulator* and *assignment* access patterns are more likely to happen. In an accumulator access pattern, the object is updated by multiple threads concurrently, and therefore all the updating should happen in a critical section. In an assignment pattern, the object accesses obey the precedence constraint. The assignment pattern is used to safely transfer a value from one thread to another thread. The producer-consumer pattern is a special case of the assignment pattern. In the current status of PAT, an accumulator and an assignment pattern analyzer have not been implemented yet. A further consideration is to incorporate data race detection into this analyzer. If an object is accessed by multiple threads without proper synchronization, and at least one thread is a writer, it is a data race situation that probably indicates a program bug.

5. Access Pattern Visualization

PAT uses a pattern-centric presentation to provide visualization of access patterns. There are three components in the presentation, a time lines window displaying the low-level access events, an access pattern analysis result window revealing the object access patterns, and a Java source code window displaying the application's source code. The time lines window also reflects the overall access operations incurred in the execution.

The time lines window, as shown in figure 3, provides a complete execution picture for SOR on 8 cluster nodes. The x-axis represents the time. In the y-axis direction, there are 8 time lines in the figure, representing 8 threads, one thread on each node in this experiment. The rectangles on the time lines show some states, e.g., barrier synchronization in this case. The arrows show the object access events. Those in gray are writes and those in white are reads. Furthermore, the arrows started in one thread's time line and ended in another thread's time line represent the remote reads (object faulting-ins) or the remote writes (diff propagations). They are issued by the threads represented by the arrows' starting time lines. The corresponding home nodes are the nodes represented by the arrows' ending time lines. The arrows overlapping with the time lines are the home reads or home writes. We can click any arrow to see the detail information about that object access, e.g. the class name, size, and ID of that object. The time lines can be zoomed out to get an overall picture of the accesses behavior, or zoomed in to examine some particular object accesses. We implement the time lines window by modifying Jumpshot in MPE [6].

Moreover, clicking the "Pattern Analysis" button in the time lines window will trigger the pop-up of the object access pattern analysis result window, as shown in figure 4. As SOR is a barrier synchronized application, the global phase pattern analyzer can provide the pattern analysis result for each object. The objects are firstly sorted by their allocation

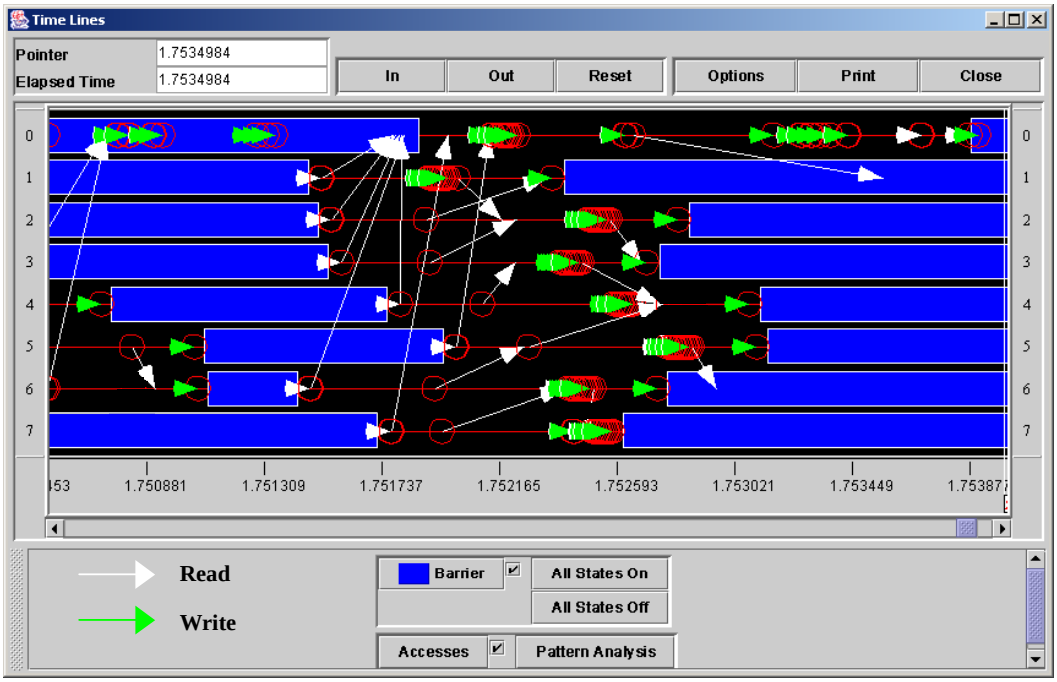


Figure 3. The time lines window

sites where they are created in the source code. Each allocation site may create many objects at runtime. For each object, its access pattern at each phase is displayed. As observed from the analysis result, most objects in SOR present the single-writer access pattern. For example, in figure 4, the object being observed presents the read-only and the single-writer pattern in alternate phases.

The window of pattern analysis result is in the center of the visualization. Inside this window, we can choose any object to highlight its accesses in the time lines window. Thus we provide a convenient association between high level access pattern knowledge and the low-level access event details. Since the objects are sorted by their allocation sites in the pattern analysis result window, we can map any object to its actual allocation site in the application's source code by clicking it, as shown in figure 4. Note that the highlighted line in the source code window is the actual position for the highlighted allocation site in the pattern analysis result window. Thus we provide a convenient association between the object access pattern and the object's corresponding allocation site in the source code.

In such a design, our visualization tool not only help us, the GOS designer, to visually evaluate the effectiveness of the adaptive protocol being applied, but also the multi-threaded Java application programmer to better understand the access behavior inherent in the program.

As a further demonstration, figure 5 shows the effect of object home migration on SOR. Figure 5 (a) is the time

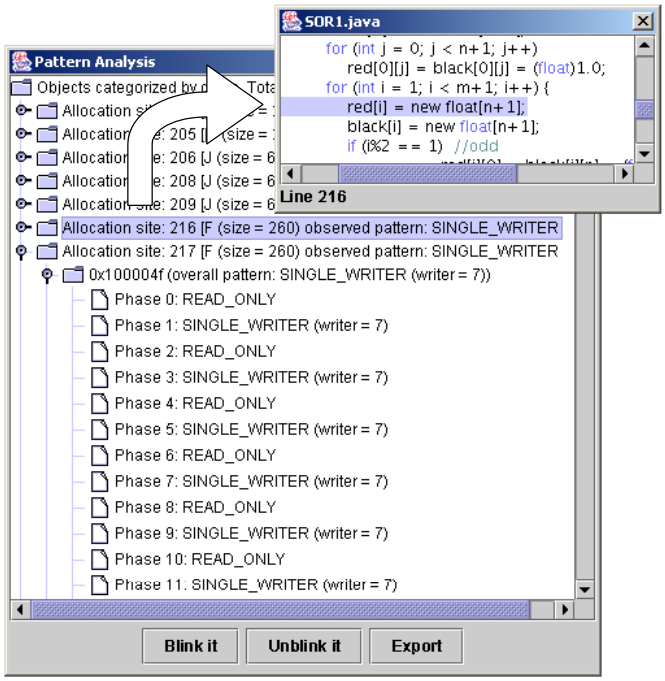
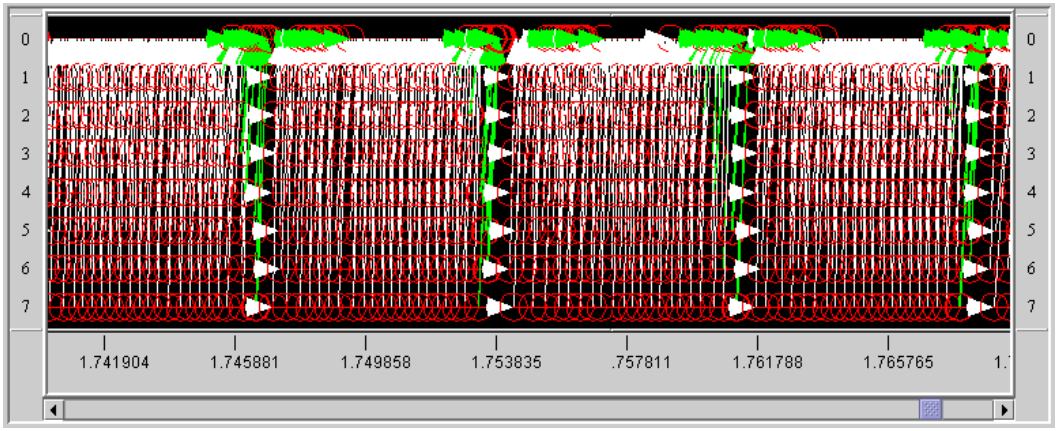
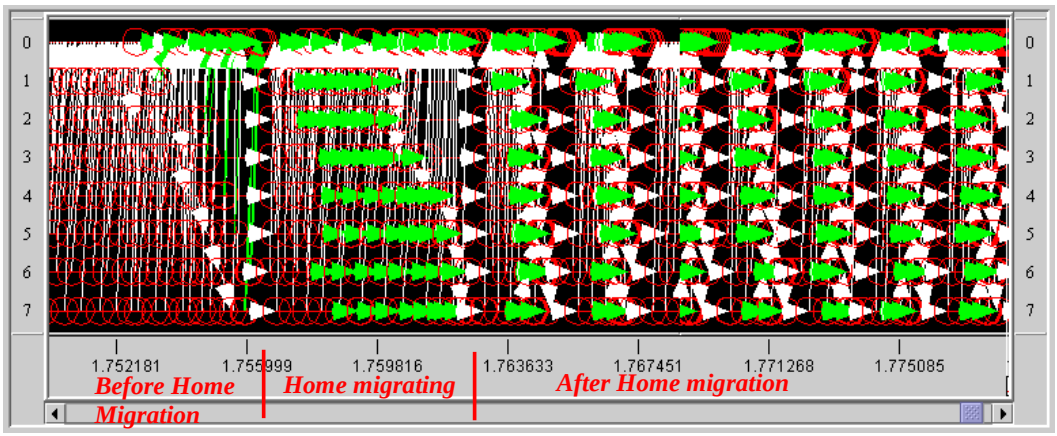


Figure 4. The window of object access pattern analysis result (the bigger one) and the window of the application's source code (the smaller one)



(a) Without home migration



(b) With home migration

Figure 5. The effect of object home migration on SOR

line window without home migration. There are four global phases, each taking approximately the same amount of time. Figure 5 (b) is the time line window with home migration enabled. Three global phases are marked in the figure: "Before Home Migration", "Home Migrating", and "After Home Migration". As we already know, most objects in SOR presents the single-writer pattern and their original homes are not the sole writing nodes. Since our GOS adopts a home-based cache coherence protocol, before home migration takes effect, we observe that a lot of remote reads and writes are sent to their home node, node 0. Our GOS is able to automatically detect such single-writer patterns and perform object home migration by setting the object's home to be the sole writing node at runtime. During the home migrating phase, we observe that although the reads (white arrows) are still sent to the original home node, the writes (gray arrows) are performed locally. It means the home has already been migrated to the local node at that moment. We can also observe that the phase after home migration takes much less time than the phase before home migration since most remote reads and writes are eliminated by object home migration. As can be observed, the effect of home migration is to change remote read/write to home read/write.

6. Related Work

StormWatch [7] is a profiling tool that visualizes the execution of DSM systems and links it to the program's source code. StormWatch provides three linked graphic views: trace, communication and source. The trace and the communication view combined together correspond to the time lines window in our visualization part, which reflects the low level access events in the execution. The major difference between our tool and StormWatch is that StormWatch only focuses on the low level access events, which may not provide straightforward and intuitive infor-

mation to the users. However, our pattern analysis and visualization system provides a pattern-centric view. The access pattern knowledge, as high level information, will definitely be more helpful to the users.

Xu et al. described a profiling approach for DSM systems in [15]. It can detect and visualize some cache block level access patterns. However, as an online tool, it suffers from the memory and time constraints in a thorough runtime analysis. For example, it can only show lifetime access pattern that a certain cache block presents in the whole execution time. The pattern change cannot be expressed because the memory consumption is expensive if each pattern change per cache block is recorded. This is neither flexible nor precise. On the contrary, our approach is postmortem, and thus does not has such a drawback. We can invest as much effort as affordable to precisely and thoroughly analyze the access patterns after we have the runtime trace.

7. Conclusion and Future Work

This paper describes a visualization tool for analyzing memory access patterns in object-oriented software DSM systems. It features a lightweight runtime solution for object access trace generation, an extensible pattern analysis engine, and an access pattern centric presentation for the visualization. This profiling and visualization tool can help the adaptive protocol designer and parallel program designer to observe runtime memory access and communication patterns arising from their algorithms, and therefore gain insights into the application's behavior.

From our preliminary experience, the design of our tool seems to satisfy our original goal. Firstly, it helps explain the benefits of adaptive cache coherence protocols in our GOS for distributed JVM visuallly and straightforwardly. Secondly, it helps us understand the application's memory access behavior by mapping the object access patterns to the corresponding allocation sites in the source code.

This paper presents a preliminary result of our work. In the future, we will extend the pattern analysis engine to detect more patterns, e.g., producer-consumer, accumulator, and assignment patterns. A more intuitive pattern visualization method is in the making to replace the current view in the window of object access pattern analysis result. We are also investigating the most convenient way to port our object access pattern analysis and visualization tool to other DSM systems, in particular, object-based DSM systems.

References

[1] Distributed Shared Memory Homepage. http://www.ics.uci.edu/~javid/dsm.html.

[2] C. Amza, A. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. In *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, volume 87, pages 467–475, March 1999.

[3] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), Feberury 1998.

[4] G. Bracha, J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[6] A. Chan, W. Gropp, and E. Lusk. User's Guide for MPE: Extensions for MPI Programs.

[7] T. M. Chilimbi, T. Ball, S. G. Eick, and J. R. . Larus. StormWatch: A Tool for Visualizing Memory System Protocols. In *Supercomputing '95*, December 1995.

[8] W. Fang, C.-L. Wang, and F. C. Lau. On the Design of Global Object Space for Efficient Multi-threading Java Computing on Clusters. *to appear in Special Issue on Parallel and Distributed Scientific and Engineering Computing in the Parallel Computing Journal*.

[9] K. Hwang and Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[12] D. L. Mills. RFC 1305 - Network Time Protocol (Version 3) Specification, Implementation, March 1992.

[13] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *the 4th IEEE International Symposium on High-Performance Computer Architecture*, Feb 1998.

[14] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High Level Machine-Independent Language for Parallel Programming. *Computer*, 26(6):28–38, 1993.

[15] Z. Xu, J. R. Larus, and B. P. Miller. Shared Memory Performance Profiling. In *Principles Practice of Parallel Programming*, pages 240–251, 1997.