Issues in Building a Parallel Java™ Virtual Machine on Cenju-3/DE

Xavier Défago* Akihiko Konagaya

Computer System Research Laboratory
C&C System Research Laboratories, NEC Corporation
Miyazakidai, Miyamae-ku, Kanagawa 216, Japan
Email: defago@lse.epfl.ch, konagaya@csl.cl.nec.co.jp

Abstract

This paper describes how a Java[1] virtual machine can use the inherent concurrency of Java programs on a massively parallel processor machine (MPP) like NEC Cenju-3. An application written with the Java programming language has two different ways to introduce concurrency; *threads* and *processes*. While it is possible to introduce parallelism by using different processes, the granularity is likely to be coarse and applications would have to be written specifically for parallel machines or would lack flexibility, scalability and the level of parallelism would be very low. On the other hand, while most non-trivial Java applications are already highly multi-threaded, the problem of locality of objects shared between threads makes it difficult to take advantage of the availability of many processors. Since the availability of multi-threaded applications is going to be extremely large, there is much to gain in being able to exploit the concurrency of such applications. This is what we try to address in this paper by describing the issues raised in the implementation of a parallel Java virtual machine for Cenju-3. This paper illustrates the use of low-level communications and remote DMA accesses in implementing a distributed shared memory for the Java virtual machine.

1 Introduction

The computing world is becoming more and more focused on the distribution of services over the Internet, and it is widely believed that such a growth will go on for a few years at the very least. In this context, the Java programming language, introduced by Sun Microsystems in April 1995, tries to address the issue of programming distributed applications on the Internet but, so far, the development has mainly been oriented towards clients while issues concerning servers have been greatly ignored. With the worldwide success met by the Java technology and its current and future performance improvement, it can be expected that network servers that are making full use of this technology will begin to play a very important role in the near future.

With their performance and scalability, massively parallel machines have a good opportunity to get into new horizons and play a major role as servers on the Internet. This direction might become especially important if, as many people claim, the future will see billions of independent Internet devices and a few very powerful servers providing the computing power and most of the information. Although a little extreme, this view is quite correct and servers like AltaVista (`http://www.altavista.digital.com/`) seem to prove it.

Enabling a parallel machine to run Java bytecode aims at solving some of the weakest points of massively parallel machines. Whenever a new machine is built, many applications need to be ported to exploit the new architecture. This is very time consuming and usually results in applications being available only after a long period of time. If most of the application are available as Java bytecode, the porting effort is significantly reduced. The increasing availability of Java applications directly contribute to the machine. Further, if care is taken over time to optimize the virtual machine, every application running on top of it directly benefit from this effort.

This paper presents some of the main issues in implementing a Java Virtual Machine for Cenju-3 and presents an implementation of a distributed mem-

*Currently in Laboratoire de Systèmes d'Exploitation, Département d'Informatique, École Polytechnique Fédérale de Lausanne, Switzerland.

[1] Java is a trademark of Sun Microsystems, Inc.

ory suitable for both Cenju-3 and the Java virtual machine.

The paper is organized as follows: Section 2 gives some background information on Cenju-3 and concurrency in the Java programming language. Section 3 discusses some of the issues in building a parallel Java virtual machine. Section 4 shows the issues in sharing objects and presents some consistency protocols for distributed shared memory. Section 5 describes the distributed shared memory for parallel Java on Cenju-3.

2 Background

2.1 Cenju Network Interface (NIF)

Although Cenju-3 provides communication facilities with Mach IPC, it also has a native high-speed, low-latency communication interface called NIF [MKI94] which we directly used in building our shared objects system. The functionalities of NIF include fast, short messages and DMA transfers that can be both either unicast or multicast. A hardware multicast which cost is equivalent to the peer-to-peer communication is very interesting for building an efficient shared memory system as shown in [TKB94]. But, two simultaneous multicasts which destination sets overlap each other may result in a network deadlock. Hence its use should be restricted to one processor only. This implies using a sequencer on a PE for processing multicasts, which has the very nice side-effect of ensuring a total order of delivery.

2.2 Concurrency in Java

The Java programming language inherently supports concurrency [Lea97]. As a result, applications written in this language tend to show a high degree of concurrency. The language provides a few basic constructs designed to support concurrent programming:

- *Initiate concurrency.* In order to create concurrency, the language provides the standard class `java.lang.Thread`. Each time this class is instantiated, a new thread is generated.
- *Mutual exclusion.* In order to control the execution of code and guarantee mutual exclusion, the language provides the keyword `synchronized`.
- *Synchronization.* It is possible to synchronize threads with `wait()`, `notify()`, and `notifyAll()` defined in `java.lang.Object`.

Since applications will rely on this, it is important to know the guarantees made by the specification. A parallel virtual machine — or any virtual machine — should at least provide the following guarantees:

- *Priority.* Although Java supports 10 levels of priority for its threads, conformance is not strictly enforced by the specification. As a result, no assumption based on thread priorities can be made by Java developers.
- *Atomic update.* Java guarantees that basic operations (most bytecode instructions) are executed atomically. This includes accesses (read or write) to all built-in scalar types except `long` and `double`.
- *Cache issue.* Except for variables marked as `volatile`, there is no guarantee that assignments performed in one thread will be seen in another thread.
- *Mutual exclusion.* A lock is associated with each object. Code qualified as `synchronized` begins by acquiring the lock associated with the object, executed its code, and releases the lock before leaving. Mutual exclusion is thus guaranteed for all piece of code protected by the same lock (i.e., the same object).

3 A Parallel Java

3.1 Implicit and Explicit Parallelism

At the language level, parallelism can be tackled in two ways: *implicit* or *explicit.* The choice is usually a trade-off between ease of programming and performance.

In building a Java VM for a machine like Cenju, one of the strongest motivation is to benefit from applications developed for the PC market. This implies that normal bytecode should execute on the machine. Applications developed for the Cenju should also be executable on any other architecture. But, to maximize performance, we need to optimize aspects like the placement of thread. The idea is to embed *hints* on these matters, through attributes [LY96].

Then, a parallel compiler can generate annotated bytecode for the parallel virtual machine. These annotation give hints to the virtual machine to solve problems like the placement of threads on each PE. The same code is executable by any other Java virtual machine since the hint are not understand and hence ignored.

We advocate a flexible approach that allows both an implicit and an explicit approach to parallelism.

3.2 Location transparency

In order to exploit the concurrency introduced by the multi-threaded aspect of applications written in Java, a thread should be allowed to run on any PE, without any constraint. This requirement implies that the virtual machine has to provide location transparency for threads. So, it makes it possible for an application to have its threads running on different PEs. Sharing objects between two processors is a key step toward providing location transparency for threads.

4 Sharing Java Objects

In order to share data between processors, a distributed shared memory system is needed. We have built one making use of broadcast DMA transfers. Unlike most distributed memory systems, we don't share virtual memory pages, but rather base the granularity of our system on logical objects in order to partially avoid false-sharing [BT91, TB93]. We exploit here the fact that object boundaries are easily determined at runtime. We also use a consistency protocol that reduces the communication overhead.

As shown in Figure 1, one of the PEs is dedicated to act as a sequencer. A portion of the address space on each PE is reserved for the shared memory and holds the locally consistent image of that memory. This portion of memory acts pretty much like a local cache. The figure also illustrates how active objects (threads), and local or shared passive objects can interact.

4.1 Sharing Objects between PEs

A shared memory can implement different levels of consistency depending on certain trade-offs. The consistency model supported by a shared memory system has a direct implication on the volume of communication generated. Strong semantics will imply a large cost on communication but less management overhead at runtime. The best consistency model for a distributed system depends partly on the ratio between communication and computation costs, and partly on the behavior of the application that uses the system.

In the context of consistency models for shared memories, we need to consider two aspects: the *criteria*, and the *implementation*. A formal description of the main consistency models can be found in [RS95] so we will not go too much into details. We consider here two types of consistency criteria: *sequential*, and *causal* consistency.

- *Sequential consistency*, illustrated by Figure 2, was introduced by Leslie Lamport and states that a distributed system is sequentially consistent if there always exists a valid sequential order of events which produces the same result of execution; every processor sees the same total order of events. Most protocols implementing sequential consistency rely on an atomic broadcast primitive [HT93] for guaranteeing this total order.

- *Causal consistency* is a more relaxed consistency model than sequential consistency and usually requires less communication. As the name indicates, it relies on the causal dependencies that might exist between some events, defining a partial order. The criterion requires the partial order defined by causal dependencies to be seen identically on every processor, but the processors can see a different serialization — or linear extension — of this partial order.

When the semantics of both consistency models are acceptable, the pertinence of choosing one over the other depends on a trade-off between communication cost and management overhead. In distributed systems, where the cost of communication is very important compared to computation, much can be gained in using a causally consistent protocol and thus reducing the amount of communication. On the other hand, hardware implementation of distributed shared memory for parallel machines benefit from a better communication environment and can afford generating more communication if difficult and time-consuming book-keeping can be avoided. For these reasons, causal consistency is more interesting in the case of distributed systems, while sequential consistency is often preferred for parallel machines. In our implementation, we rely on the broadcast facility provided by NIF to implement a release consistent protocol which finds itself between causal and sequential consistency.

4.1.1 Release Consistency

The release consistency model(RC) [CBZ91], an algorithm for building a causally consistent memory, assumes that four different events can occur, as far as the shared memory is concerned:

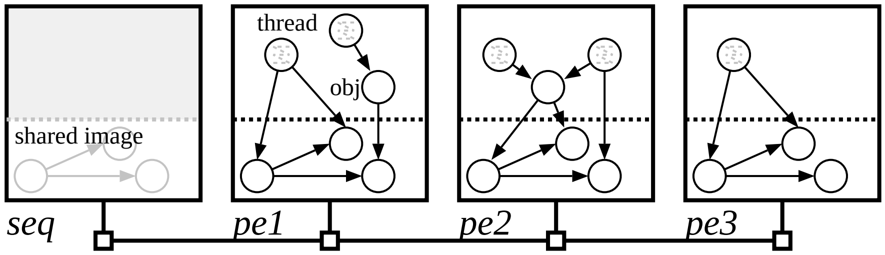

Figure 1: Architecture of the overall system.

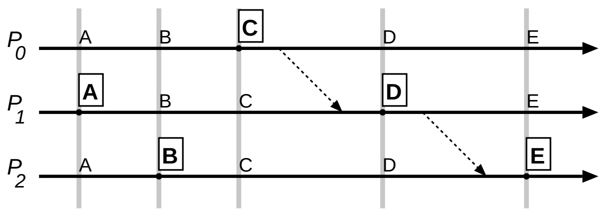

Figure 2: Sequential consistency

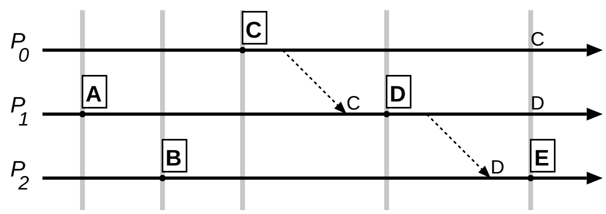

Figure 3: Causal consistency

1. *read(A)*. Read object A from the shared memory.

2. *write(A)*. Write object A in the shared memory.

3. *acquire(l)*. Wait until the lock l is available and acquire it.

4. *release(l)*. Release the lock l.

The *acquire* and *release* events roughly correspond to synchronization on a lock and delimit a critical section. They can be considered as "strong" events since they generate inter-process causal dependencies. On the other hand, read and write events do not create any dependency in this model. In short, release consistency requires that shared memory updates performed by a processor p_i become visible at a processor p_j, when the next release by p_i is seen by p_j. The Figure 4 illustrates this. In this figure, the modification performed by P_1 become visible to the other processors only after the release is performed. This is equivalent to say that groups of *acquire/release* have to be sequentially consistent with respect to each other.

It can be seen that a release on a lock and a subsequent acquire on that lock generates a causal dependency[2]. Therefore, release consistency requires that the occurrence of events from the point of view of each processor are ordered causally. In that respect, release consistency can be understood as a causal memory where locks are carrying the notion of causality and reads and writes are the events to order.

[2] *release* and *acquire* are similar to a *send* and a *receive* respectively.

4.1.2 Lazy versus eager consistency

We can consider two families of algorithms implementing release consistent shared memories; *eager* and *lazy* release consistency. The basic difference between these two models is the time at which updates are being carried over to other processors.

- *Eager RC.* As shown in Figure 4, the eager implementation of release consistency postpone the updates. On a release event, all previous modified data are directly brought up-to-date to every other processor. The system just have to make sure that no other processor will be able to get the lock before it effectively sees the update. It can be seen that a lot of unnecessary traffic is generated compared to a purely causal implementation. Eager release consistency requires groups of *acquire/release* to be sequentially consistent with respect to each other.

- *Lazy RC.* On the other hand, the Figure 5 shows a lazy consistent implementation of release consistency, where network traffic can be kept very low by delaying updates as much as possible [KCZ92, KDCZ94, CBZ95]. Lazy consistency relies directly on the definition of causality and strives to perform the modifications only

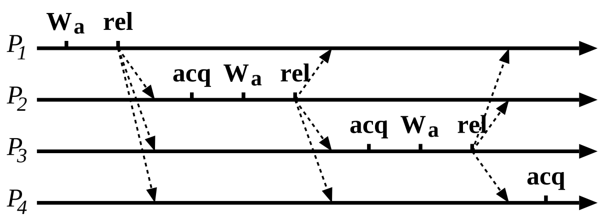

Figure 4: Message traffic with eager RC.

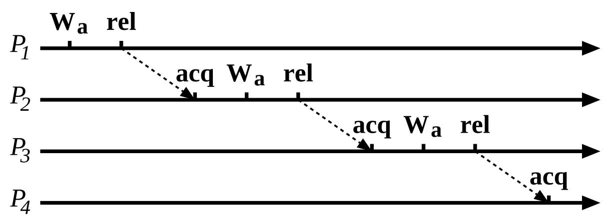

Figure 5: Message traffic with lazy RC.

at the very last moment. Although the network traffic is considerably reduced, keeping track of modified data becomes dearly significant. In other words, lazy release consistency requires groups of *acquire/release* to be causally consistent with respect to each other.

In distributed systems like clusters of workstations or even wide-area networks, the tremendous cost of communication makes negligible any extra overhead due to managing information on updates. For this reason, lazy release consistency is often preferred over the eager implementation in distributed systems. On the other hand, this is not true for parallel machines, and if a cheap broadcast primitive is available, the eager algorithm is often a better choice as it involves little management of extra resources.

5 Protocol

Since synchronization plays a central role in the release consistent protocol, it deserves a particular attention. We mentioned earlier that synchronization is achieved by using acquire and release operations on distributed locks. In order to make it clearer, we distinguish two levels for our locking algorithm:

- *Per processor arbitration.* the usage of a lock is granted to a PE and the notion of thread is totally ignored. As long as a PE owns a lock, it may deal with it as it sees fit
- *Per thread arbitration.* Arbitration occurs between threads of a single PE, competing for the lock.

Our protocol combines these two levels to allow arbitration between threads in the whole system. The protocol has to satisfy the following properties:

- *Safety.* The protocol guarantees that any execution of the protocol is correct. For any execution, there is an equivalent sequential execution.
- *Liveness.* Any execution of the protocol is guaranteed to finish eventually. Any execution that terminates on a conventional virtual machine (e.g., it does not result in a deadlock) also terminates on the parallel virtual machine.
- *Fairness.* At any time, threads have an equal chance to obtain a lock. This property is actually related to liveness, since it ensures no starvation in the system.

5.1 Per processor arbitration

The sequencer propagates the updates of objects and manages shared locks by arbitrating their use amongst PEs. As shown on Figure 6, when a PE needs a lock, it sends an ACQUIRE message to the sequencer and waits until the lock is granted with an OK message. At this point, the PE may continue, enter the critical section and modify objects. These "dirty" objects are tracked down and put into a list. At the end of the critical section, the PE does a release which implies transferring by DMA the "dirty" objects to the sequencer then sending a RELEASE message which contains the identifier of the lock as well as the coordinates of the updated objects. At this point, the PE can continue its execution while the sequencer has to multicast the updated objects before it can grant a right on that lock to the next requesting PE.

The algorithm is relatively simple since it consists only in a sequencer managing a token representing each lock, and granting it to the PEs which request it. As a matter of fact, this is the most straightforward solution to the problem of distributed locks, which also means that it is the easiest to implement. This solution has many draw-backs like limited scalability but, in our context of broadcast-based memory, there is no point in trying to avoid the bottleneck of a sequencer if we need one for broadcasting anyway. Therefore, this algorithm seems to be the best suited to our needs because it fits very well with the way we are handling broadcasts.

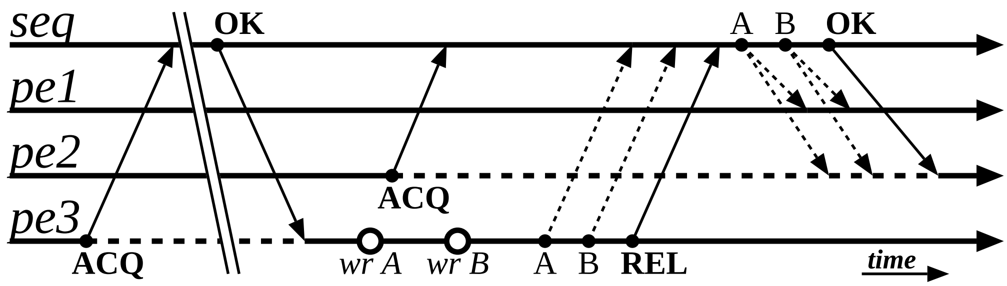

Figure 6: Global locks granting mechanism.

A lock is associated to an object which address provides a unique identifier. The sequencer keeps track of every locks and to whom it has been granted. When a PE requires a lock, the sequencer grants it right away if it is available or put the request into a waiting-queue associated with each lock. When the lock is released by the PE which owns it, the first request queuing for the lock is satisfied and a message is sent to the corresponding PE. At this point, it should be noted that nested requests for the lock by one PE are not handled specifically and might result in a deadlock if a second request is issued by the same PE. This behavior is important in order to support the client algorithm presented in the next section and illustrated by Figure 7.

5.2 Per Thread Arbitration

Some additional management was needed in order to allow many threads to run on each PE. In order to improve efficiency, some special heuristics were used. But, before going too far into details, it is better to see an example.

Three threads running on the same PE and competing for the same lock are represented in Figure 7. Thread a asks first for the lock and an ACQUIRE message is sent to the sequencer. Meanwhile, b also asks for the lock and its request is registered locally to be processed after a's. After a while, the lock is granted to the PE, which can give it to a then b. On the other hand, c asked for the lock after the PE received it and had to issue a new request to the sequencer in order to prevent starvation of other PEs. It is granted the lock only during the next cycle; once it is relinquished to the sequencer and acquired again.

As shown in Figure 8, we distinguish periods when the lock is granted to the processor. The first time a PE receives the lock delimits the beginning of a period which ends when the lock is released. The period for which a thread is eligible for a lock depends on the time when its request was issued. Once a period starts, new requests are not eligible for that period, but only for the following one.

This mechanism relies on two waiting-queues of requests, associated with each lock. Figure 9 shows that the quick queue — on the right — holds the requests made during the previous period and that are granted during the current period. The second queue keeps requests that need to wait until the next period. Whenever a new period begins, the requests enqueued in the slow queue are transferred in block to the quick queue. This ensures that a single thread can never obtain the a lock twice during the same period. The change of period occurs at line 39 in Figure 10.

This protocol has been implemented so that the number of messages is kept low while avoiding starvation problems between PEs. The two local queues of requests on a lock allow to group accesses together in order to reduce the number of requests to the sequencer. This reduces the number of messages and particularly makes a difference in the case of a high number of threads and high contention. On the other hand, if the lock is kept whenever a local request comes, the other PEs risk not having the opportunity to acquire it. Hence, in order to achieve a greater fairness, the mechanism using two request queues ensures that no thread will ever be granted the lock more than once before it is relinquished to the sequencer, thereby giving a chance to other PEs to obtain the lock. Assuming that the number of threads running on each PE is finite, it is guaranteed that the lock will eventually be released to the sequencer. Of course, this also assumes that all critical sections eventually terminate and that no deadlock occurs. But, since such a program is incorrect, this issue is irrelevant in this context.

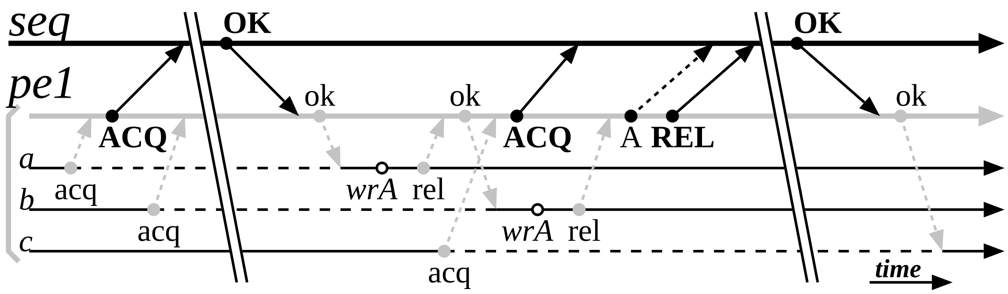

Figure 7: Local management of shared locks.

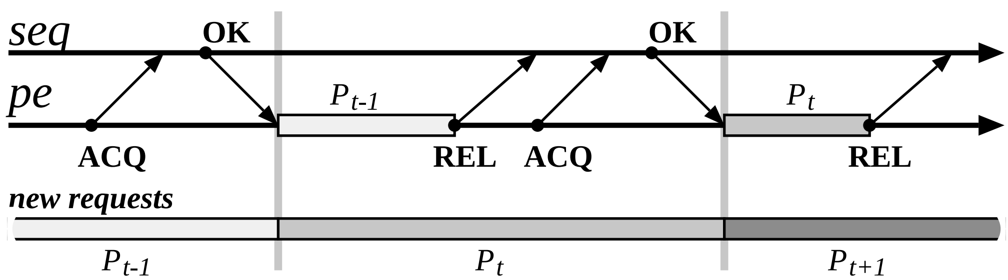

Figure 8: Period in which a lock is granted.

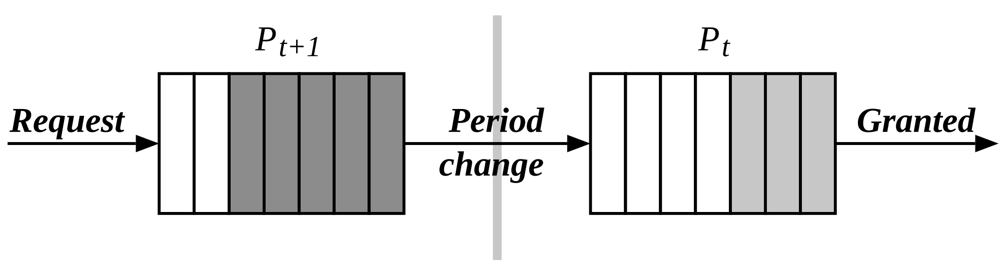

Figure 9: Request queue associated which each lock.

```
variables
   slowQ_lock ← {};                                                    {Slow request queue}
   quickQ_lock ← {};                                                   {Quick request queue}
   owner_lock ← ⊥;                                                     {Thread that owns the lock}
   dirty ← {};                                                         {list of "dirty" objects}
   request_lock ← {};                          {List of pending requests. used by the Sequencer}

procedure acquire (lock)
   if slowQ_lock is empty then
      send (ACQUIRE, lock) to Sequencer;
   append (self) to slowQ_lock;
   wait until owner_lock = self;

procedure release (lock)
   if quickQ_lock is not empty then
      owner_lock ← removeFirst(quickQ_lock);
      signal owner_lock;
   else
      foreach obj ∈ dirty do
         send (BROADCAST,obj) to Sequencer;
      send (RELEASE, lock) to Sequencer;

cobegin
|| task                                                                {Task executed by the sequencer}
   upon reception of (ACQUIRE, lock) from pe_i :
      if owner_lock = ⊥ then
         owner_lock ← pe_i;
         send (OK, lock) to owner_lock;
      else
         append (pe_i) to request_lock;
   upon reception of (RELEASE, lock) from pe_i :
      pre : owner_lock = pe_i;
         if request_lock is not empty then
            owner_lock ← removeFirst(request_lock);
            send (OK, lock) to owner_lock;
         else
            owner_lock ← ⊥;
   upon reception of (BROADCAST, obj) from pe_i :
      DMA_broadcast (obj) to all except pe_i;

|| task                                                                {Task executed by the PEs}
   upon reception of (OK, lock) from Sequencer :
      swap (slowQ_lock, quickQ_lock);
      owner_lock ← removeFirst(quickQ_lock);
      signal owner_lock;
coend
```

Figure 10: Managing acquire and release requests with many threads.

Note that, in order to make it simpler, the algorithm presented in Figure 10 do not take the problem of re-entrance into account. This is not difficult to add this property and out prototype has implemented it.

6 Conclusion

When we started to work on building a parallel version of the Java virtual machine, the language was still considered as little more than a toy by most people, and very few expected how popular it would become over only a few months. Now, all the major computer vendors are supporting, in a way or another, this new technology. Most efforts seem to be put on the client side while the issues concerning Internet servers are starting only recently to take attention.

In this paper, we propose to take massively parallel machines out of their niche market and use them as extremely powerful Internet servers. This will be beneficial to organizations needing very efficient servers like powerful search engines or services that need to handle concurrent requests from many users, like transaction systems. Furthermore, the developments that are being done to provide a suitable environment for business applications written in Java is likely to make massively parallel machines supporting this technology, very appealing to banks and other financial institutions.

The platform independence of Java programs makes it possible for new machines to benefit from a wide software base with little development; they would benefit from a very comprehensive software environment right at the start. This might prove a very big advantage for platforms with a small market like massively parallel machine as they would become attractive to organizations that need high-end computing in the context of the ever growing Internet.

This paper explores the issues in building a parallel Java virtual machine for massively parallel processors like NEC Cenju. It focuses on the solution to a key issue: sharing data between threads located on different processors. We propose an algorithm based on release consistency that makes full use of the native communication mechanism of Cenju. Although our prototype is not portable, the concept can be applied to any massively parallel machine or network of workstations with a cheap multicast primitive and remote DMA transfer.

We believe that such a work opens new horizons for research in diverse fields like compilation for concurrent object-oriented languages, parallel operating systems based on high-level languages, μkernels based on programming languages, and many other areas. This gives an opportunity for these many research areas to target new objectives, in the practical world.

7 Acknowledgments

We would like to thank Hiroyuki Araki, Koichi Konishi, Tomoyoshi Sugawara, and Kosuke Tatsukawa for their support. We would like to thank Christopher Howson for his valuable comments along this project and his help in understanding the Cenju Network interface.

References

[BT91] H. E. Bal and A. S. Tanenbaum. Distributed Programming with Shared Data. *Computer Languages*, 16(2):129–146, 1991.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[CBZ95] 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, August 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley Publishers Ltd., August 1996.

[HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press, 1993.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 18th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, 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.

[Lea97] D. Lea. *Concurrent Programming in Java*. The Java Series. Addison-Wesley Publishers Ltd., January 1997.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley Publishers Ltd., September 1996.

[MKI94] T. Maruyama, Y. Kanoh, and Y. Inamura. Cenju-3 interconnection network interface. Technical report, Computer System Lab., NEC Corporation, February 1994. translated in English by C. Howson.

[RS95] M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. In *Foundations of Software Technology and Theoretical Computer Science, 15th Conf.*, pages 180–194. Springer Verlag, LNCS 1026, December 1995.

[TB93] A. S. Tanenbaum and H. E. Bal. Programming a Distributed System Using Shared Objects. In *Proc. of the Second IEEE Int'l Symp. on High Performance Distributed Computing*, pages 5–12, July 1993.

[TKB94] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Using Broadcasting to Implement Distributed Shared Memory Efficiently. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 387–408. IEEE Computer Society Press, 1994.