Multiprocessor Smalltalk: A Case Study of a Multiprocessor-Based Programming Environment

Joseph Pallas*
David Ungar*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

We have adapted an interactive programming system (Smalltalk) to a multiprocessor (the Firefly). The task was not as difficult as might be expected, thanks to the application of three basic strategies: serialization, replication, and reorganization. Serialization of access to resources disallows concurrent access. Replication provides multiple instances of resources when they cannot or should not be serialized. Reorganization allows us to restructure part of the system when the other two strategies cannot be applied.

We serialized I/O, memory allocation, garbage collection, and scheduling, we replicated the interpreter process, software caches, and a free-list, and we reorganized portions of the scheduling system to deal with some deep-seated assumptions. Our changes yielded a fairly low static overhead. We attribute our success to the choice of a small, flexible operating system, a set of constraints which simplified the problem, and the versatility of the three strategies for dealing with concurrency. The current system has a moderate amount of overhead when parallelism is being used—25% to 65%. It is acceptable, but we believe it can be improved.

*This research was supported by National Science Foundation grant number CCR-8657631.

Proceedings of the SIGPLAN '88
Conference on Programming
Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

1 Introduction

Although it is clear that a multiprocessor will benefit a multiuser, timesharing environment, it is not so clear how to exploit a multiprocessor in the single-user environment of the personal workstation. While research into multiprocessor workstations is progressing [6, 11, 17], we do not yet have a good understanding of how to use them effectively as personal workstations. In the case of the interactive programming environment (such as LISP, Smalltalk, *etc.*), the task of adapting to and exploiting the multiprocessor will fall first to the language system implementor.

We have adapted one particular interactive programming system, the Berkeley Smalltalk (BS) implementation of the Smalltalk-80[1] language [8, 20], to an experimental multiprocessor, the DEC-SRC Firefly. The result is called Multiprocessor Smalltalk (MS). We found that the task was not as difficult as might be expected, and, in particular, that a relatively small set of strategies was sufficient to deal with the problems involved. Although some of the lessons learned may be applicable only to object-oriented systems, the basic principles should apply to any effort to adapt a single-threaded, single-address-space programming system to a shared-memory multiprocessor; tasks such as storage management, input/output, and process scheduling are common to most integrated programming environments.

Although we will present some preliminary performance figures, the focus of this paper is on the design and implementation of MS, not the performance. In particular, it should be noted that the performance of MS has not been tuned yet, nor has it been analyzed in any detail. We expect to present that work in a future paper, along with some comments on our experience in using MS.

[1] Smalltalk-80 is a trademark of ParcPlace Systems.

1.1 Context and Goals

This report describes only one part of a larger project to implement, instrument, and analyze a multiprocessor Smalltalk. The ultimate objective of the project is gather the information necessary to make design decisions about the implementation of an object-oriented system on a multiprocessor. The first part of the project will produce information about the obstacles to adapting a single-threaded uniprocessor programming system to a multiprocessor. The second stage will be devoted to gathering data appropriate for characterizing the behavior of the system and predicting the behavior of similar systems. The experiments will focus on how the choice of multiprocessor architecture affects the performance of the system, as well as the effects of different parallelization strategies (*i.e.*, choice of algorithm) at the user level.

1.2 Constraints

With these goals in mind, the implementation itself is intended to be as straightforward as possible. Since our interest is in the behavior of the programs, rather than that of the system which interprets them, we have applied two principles wherever possible:

1. Take the path of least resistance in adapting the interpreter, except where such a path conflicts with the goal of maximizing concurrency.

2. Make the minimum possible change to the user-visible programming environment.

The latter principle is reflected as the least possible change to the Smalltalk-80 *virtual image*, the existing hierarchy of object behavior.[2] In particular, we have *not* changed the existing Smalltalk abstractions for dealing with concurrency. The basic mechanisms remain the **Process** and the **Semaphore**.

1.3 Related Work

Other integrated programming environments for personal workstations include Cedar [16] and several Lisp systems. Cedar is supposed to be structured so as to avoid difficulties in running the system on a multiprocessor, although, as far as we know, it has not been run on one. The SPUR project aims to produce a multiprocessor workstation for running LISP [22]. The project spans all the design levels from processor to software. The Apiary [9] is a design for an object-oriented multiprocessor that does not use shared memory. It merges the **Process** and **Object** concepts into **Actors**. Actors behave like active objects, and provide implicit concurrency and serialization. The Actra project [2, 18] aims to produce an industrial multiprocessor Smalltalk system. One significant difference between MS and Actra is that MS avoided changes to the user-visible environment, while Actra layers an Actor-like structure for dealing with parallelism on top of the Smalltalk environment. Bennett's Distributed Smalltalk [3, 4] also addresses distributed systems which do not use shared memory. Work which addresses the problems of garbage collection in shared and distributed memory is cited in Section 3.1.

[2]Usage note: we use "Smalltalk" to refer to the language, and "Smalltalk-80" to refer to the language, object hierarchy, and virtual image defined in *Smalltalk-80: The Language and Its Implementation* [8]. The virtual image is a static representation or "snapshot" of the compiled code, class descriptions, *etc.*

2 Environment

The hardware base for the project is the Firefly, an experimental shared-memory multiprocessor developed at Digital Equipment Corporation's Systems Research Center. The machine consists of five microVAX processors and 16 megabytes of shared memory. Each of the processors is approximately the speed of a VAX 11/780, and each has a private cache of 16K bytes. The cache hardware guarantees the consistency of shared data (details of the architecture have been published elsewhere [17]).

The software platform for the project is the V distributed operating system [5]. The V kernel provides a message-passing inter-process communication facility, along with address spaces and lightweight processes, upon which most operating-system level services, such as program loading and a file system, are built. The current implementation of the V kernel on the Firefly allows multiple user-level processes to run concurrently on the five processors. Although there is no support at this time for multiple processes inside the kernel, this is only a minor consideration for our work, because of the relative infrequency of requests for kernel operations by the Smalltalk system.

The experimental subjects are the Berkeley Smalltalk interpreter and the ParcPlace Systems Smalltalk-80 virtual image release VI2.1. Berkeley Smalltalk is a byte-code based interpreter for the Smalltalk-80 virtual machine. The use of an interpreter may have some influence on the data generated by the experiment, because some activities (such as storage allocation) occur less often due to interpretive overhead than they would in a compiled system. Unfortunately, the only readily-available Smalltalk system that uses compiled machine code does not run on the available hardware, and it is written in assembly language, making it difficult to modify. The version of Berkeley Smalltalk with which we are working differs from that described in *Smalltalk-80: Bits of History, Words of Advice* [20] in that it employs the Generation Scavenging garbage-collection scheme [19] instead of reference counting. It also eliminates the *object table*, which otherwise would add a level of indirection to object references.

The overall structure of the system is depicted in Figure 1. BS introduces a level of interpretation into the system, as shown in Table 1. A Smalltalk **Process**[4] is a thread of execution of Smalltalk byte codes. The byte codes are produced by the Smalltalk compiler from Smalltalk source code, and they reside in the object memory. The execution of the bytecodes is performed by the Smalltalk interpreter, and the scheduling of Smalltalk **Processes** is done by the Smalltalk **ProcessorScheduler** (whose behavior is defined by the language and implemented both in Smalltalk code and by the interpreter). The Smalltalk interpreter is itself a V process, which is the execution of machine code generated by the C compiler. The machine code resides in the V process's address space,[5] and is executed directly by the processor. Scheduling of V processes is done by the V kernel.

3 Adaptation Strategies

Concurrent access to shared data must be controlled to guarantee a consistent view of the state of a system. This control, commonly known as *synchronization*, is the common characteristic of almost all of the problems involved in adapting single-threaded code to run multi-threaded on a multiprocessor. Synchronization problems may occur any time that more than one thread of control makes use of the same data. For this project, our chosen constraints allowed us to ignore the most common references to global data: the values of variables in the Smalltalk object space. With the exception of storage management functions, which are performed by the run-time system, we have relegated to user-level code responsibility for the synchronization of references to user-visible data. This allows the interpreter to run at full speed most of the time, and affords maximum flexibility for experimenting with concurrency at the user level.

[4]To avoid confusion, we have tried to use "**Process**" consistently whenever we are referring to a Smalltalk **Process** as distinct from a V process or the general concept of a process.

[5]The interpreter processes share a single address space, which is why they are called *lightweight* processes.

The interpreter itself, however, makes use of global data for a number of purposes. These purposes include input and output operations, storage management functions, a method lookup cache, and others. We adopted three basic strategies to deal with the synchronization problems of global data:

Serialization of concurrent access to shared resources.

Replication of resources to allow concurrent access.

Reorganization of the computation's structure to avoid the need for shared data. This strategy is used only as a last resort, because of the constraints we have specified (in Section 1.2).

Every problem of concurrent access we encountered yielded to one of these three approaches.

3.1 Serialization

Serialization is the most familiar tactic for dealing with concurrency problems (in various forms, such as critical regions, semaphores, and monitors). On a uniprocessor, locks on data are an acceptable way to achieve serialization because (in the absence of deadlock) the process which holds a data lock is always able to proceed. Hence, the number of ready processes is not less than the number of processors available to run them. In a multiprocessor, however, serialization with locks is a much more expensive form of synchronization, because it eliminates parallelism. When more than one processor is available, demanding exclusive access to data can cause the number of ready processes to drop below the number of processors, reducing processor utilization. This indicates that serialization should be avoided if contention for a lock is likely.

MS uses serialization when either:

- Access is brief and relatively infrequent, or
- No straightforward non-serial solution exists.

For very brief periods of exclusion, we rely on a spin-lock mechanism based on the processor's interlocked `test-and-set` instruction.[6] If the test fails, the locking code invokes the kernel's `Delay` operation with a minimal timeout, which allows V process switching to occur, if necessary, and also avoids monopolizing the memory bus.

Despite the potential drawbacks of spin-locks, we solved the majority of the potential synchronization problems in MS with serialization:

[6]The V System spin-locks were implemented by Cary Gray.

Figure 1: System structure

The basic structural elements in MS are the virtual image, the virtual machine (including the interpreter and run-time library), and the V kernel. The thick lines indicate protection boundaries which can only be crossed by invoking special primitive operations. The window system, compiler, and other elements of the system that usually are inaccessible at the user level are completely visible, and applications reside in the same object-memory space as the rest of the Smalltalk system.

Let's consider a typical sequence of events in order to illustrate the relationships among the various layers. Suppose the Smalltalk compiler is reading a file which resides on a remote file server. The compiler sends a Smalltalk **read** message to a Smalltalk file object whose behavior is defined by the Smalltalk file system. The file object, if it doesn't have the data buffered, invokes a file-read Smalltalk primitive operation.[3] This crosses the protection boundary into the BS interpreter, which calls the V run-time library `fread()` function (written in C). If the data requested are not buffered at this level, that function will invoke a V kernel operation to send a V READ message to the file server. This crosses the lower protection boundary, where it is handled by the V interprocess communication (IPC) manager. The IPC manager then directs the message to the network through the low-level device interface. The return path is similar, only reversed.

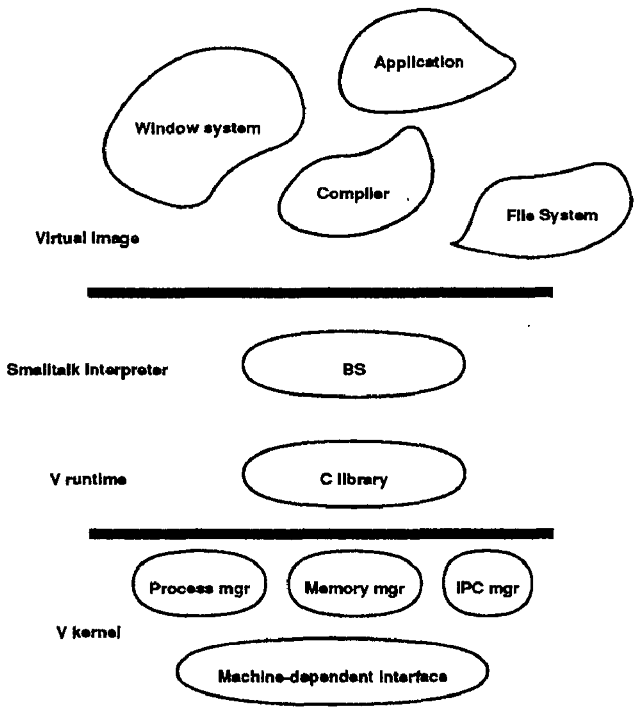

Table 1: Process and interpreter relationships

	Virtual image	Interpreter
Execution process is	Smalltalk **Process**	lightweight process
Compiled code consists of	byte code	machine code
Code is written in	Smalltalk	C
Code and data reside in	object memory	address space
Execution is by	Smalltalk interpreter	machine processor
Execution scheduler is	Smalltalk **ProcessorScheduler**	V kernel

I/O: The interpreter places input events on a queue which is shared (potentially) by several processes. There is also an output queue associated with the display controller, into which display commands are placed. In both of these cases, access to the shared resource is for very brief intervals.

Memory allocation: Memory allocation in the Generation Scavenging system is quite fast— it amounts to little more than incrementing a pointer. Allocation is also comparatively infrequent, making serialization appropriate in this case. It's also simpler than the alternative of having a separate allocation area for each of the processors. These arguments may not hold when multiple processes are busily allocating storage, however, or in a non-interpretive system, which would allocate storage more often.

Garbage collection: Several parallel garbage collection algorithms have been proposed [1, 7, 10, 12, 13, 15, 21]. Those which rely on an extra level of indirection (as with forwarding pointers or object tables) slow down common operations. Without such indirection, it appears to be impossible to have normal processing continue in parallel with garbage collection. It may, however, be possible to apply multiple processors to the garbage collection task, especially if the memory system is divided into several spaces.

BS collects garbage using *Generation Scavenging*, a stop-and-copy scheme. Since scavenging requires all of the live new objects to move, and no indirection or forwarding is used except during the scavenging activity, the interpreter must suspend all other activity for the duration of the operation. Fortunately, scavenging is fast—it takes about 3% of the available processor time on a uniprocessor [19].

The two factors influencing this figure of 3% are how often scavenging is performed and how long each scavenging operation takes. The first of these is roughly $\frac{r}{s}$, where r is the rate of allocation and s is the size of the allocation space, and can be kept constant by increasing s as r increases (provided memory is available). If scavenging occurs every t seconds in a uniprocessor system with an allocation space of size s, then a multiprocessor system with k processors should require scavenging no more often than every t seconds if the allocation space is of size $k \cdot s$.

In the current version of MS, the size of the allocation space, s, is only 80K bytes with the time between scavenges, t, being roughly 10 seconds. This leaves plenty of room for faster systems with more processors before memory becomes a concern.

Furthermore, the time for each scavenging operation is proportional to the amount of new live data (that is, data which are *not* garbage). Although we haven't yet investigated the behavior of parallel Smalltalk programs, it may be that a k-processor system, when working cooperatively on a single computation, will not generate k times as much *surviving* data; so the total scavenging overhead may be less than $3k$% when scavenging is done by a single processor. Applying multiple processors to the scavenging operation should yield a total overhead of no more than 3%; we haven't yet performed this experiment.

Since garbage collection takes a long time compared to other interpreter activities, we do not employ spin-locks in serializing scavenging. Instead, all of the processes are synchronized with a global flag and the V interprocess communication mechanism.

Entry Table Maintenance: Entry table maintenance (also called *remembering* or store checking) is an essential operation in generation-based garbage collection schemes—it is the recording of old objects which refer to younger ones, so that the younger ones can be scavenged without scanning all of the space containing old objects. BS implements the set of remembered objects with an array and a flag on each object indicating whether it has already been remembered. MS puts a lock on the array that also synchronizes tests on the "remembered" flag.

Scheduling: The Smalltalk-80 system employs a simple scheduling model. It is based on a priority queue which is examined whenever a **Semaphore** is signalled or a **Process** manipulation primitive (*e.g.*, **suspend** or **resume**) is invoked. These events are relatively infrequent, so serialization through a lock on the queue is adequate. Since the queue is visible at the user level, there is some potential for concurrency problems in its manipulation; this is discussed further in Section 3.3.

3.2 Replication

When some resource is essential to the operation of the system (for example, one that is used continuously), replication of that resource is likely to be the best way to make it available to multiple processes. In the case of an interpreter, we obtain parallelism by replicating the interpreter itself (or, rather, the interpretation process). In MS, multiple light-weight processes are supplied by the V kernel to achieve this. We create processes for as many interpreters as are desired, up to the maximum number of processors available.[7]

Since the Smalltalk interpreter process is replicated to achieve parallelism, any resources which are used continuously by the interpreter must also be replicated. Most of these resources are local to the interpretation process, but one in particular is not. The interpreter's notion of the active process (that is, which Smalltalk **Process** the interpreter is executing) must be visible to the other interpreter processes so that they can cooperate in implementing the behavior of the **ProcessorScheduler**.

Although the interpretation process or *virtual machine* was replicated, the Smalltalk representation of the virtual machine, the **ProcessorScheduler**, was *not* replicated. That is, at the Smalltalk level, in-

[7]Additional processes could be created, but they could not all run in parallel.

stead of seeing several virtual machines, we see a single virtual machine which is able to run multiple processes in parallel; there remains a single priority queue of active **Processes**, rather than one for each interpreter process. One reason for doing this was that it would have a lesser impact on the existing Smalltalk code which implements the scheduling of Smalltalk **Processes**.[8] Another reason for keeping a single ready queue is to avoid the necessity of moving **Processes** from one queue to another when an interpreter process becomes available. The adaptation of the V kernel to the multiprocessor, however, does use replication to deal with the ready queue; the kernel maintains a separate queue of ready processes for each processor. At the moment, V processes are statically assigned to processors, so there may be cases in which processors are idle even though there are processes which could run. In MS, however, Smalltalk **Processes** are dynamically assigned to V processes, which is practical only because of the shared-memory architecture, and convenient only because the ready queue was not replicated.

A resource that we found necessary to replicate was the *method cache*. A Smalltalk implementation performs a "method lookup" (mapping from the name of a polymorphic procedure to the actual code or *method*) very frequently; in typical interactive use, more than 10% of the bytecodes interpreted require lookup [20]. As a result, most Smalltalk implementations rely heavily on software method-lookup caches to achieve acceptable performance, and BS is no exception. We originally applied a serialization strategy for the method cache, using a two-level locking scheme to allow multiple readers. When the system was finally up and running, however, we found that contention for the lock was causing it to run much too slowly. Replicating the cache on a per-processor basis solved the problem. The drawback, of course, is that more overhead is involved in access to the cache because it is replicated. Some fine-tuning of the implementation may reduce this.

Profiling of an earlier version of MS revealed that serialization of access to the free context list caused a bottleneck. The free context list serves as an optimization of the memory allocation process for Smalltalk stack frames, or **Contexts**. BS maintains a list of unused stack frames, because it is more efficient to reuse one than to allocate and initialize a new one. Replication of the free context list yielded a reduction in the worst-case overhead from 160% to 65%, an improvement of 60%.

[8]In retrospect, this should not have been a consideration; existing Smalltalk code that manipulates the queue makes no allowances for concurrency.

3.3 Reorganization

Sometimes there are assumptions which are embedded so deeply in parts of a system that they cannot be accomodated, but must instead be expunged. We call this activity reorganization, because it involves more than the simple locking or replicating of resources; it may even require eliminating some data or rearranging their use. Although a good designer tries to confine such assumptions, they can be expressed in many unexpected ways which are not immediately obvious—until one tries to go against them.

The Smalltalk-80 code that manipulates **Processes** displays one of these embedded assumptions—that Smalltalk **Processes** run only one at a time. Although there is but a single place where this assumption is explicit (in the definition of the **ProcessorScheduler**), it appears as an implicit assumption of the following form: *If an active* **Process** *manipulates any* **Process** *other than itself, that other* **Process** *is not active.* This assumption shows up in a number of subtle ways, and, as might be expected, it causes several problems.

The **ProcessorScheduler** has room for only one active **Process**, which is not adequate to represent the current state of the system in MS. Although replication would have been an acceptable strategy in this case, for example, by making the activeProcess slot hold an array or linked list of **Processes**, we instead opted for compatibility with other Smalltalk interpreters. Instead of replicating the activeProcess, we ignored it; only the interpreter knows directly whether a **Process** is running. By not changing the structure of the **ProcessorScheduler**, MS retains image-level compatibility with BS. The only requirement is to fill in the activeProcess slot before taking a snapshot and to empty it afterwards.

Ignoring the activeProcess variable requires changes in the implementation of the **activeProcess** message. At this point the troublesome assumption appears again: in the Smalltalk-80 system, there is no distinction between the questions "Which **Process** is this execution path?" and "Is **Process** x active?" In the standard Smalltalk-80 image, both of these questions are folded into the question "Which **Process** is active?", which is not a well-formed question in the multiprocessor context. Consequently, we replaced sends to the **ProcessorScheduler** of the message **activeProcess** with sends of the message **thisProcess** or sends of the message **canRun:** *someProcess*. Thanks to the way that primitives work in the Smalltalk-80 system, if one of the new primitives fails because it is not implemented (*i.e.*, the virtual image is not being running under MS), control falls through to the code for the old **activeProcess** message, preserving compatibility with BS.

The choice of name for the new **canRun:** message reflects an important difference in its meaning from that of a hypothetical **isActive:** message. The reason for this is to make allowances for concurrency which were not present in the original code. In a system with multiple concurrent processes, it is not wise to distinguish between a process which is currently running and one which is ready to run, because a change in the status of some other process could cause a change in that of the process being examined. In recognition of this, the MS system does not remove a **Process** from the ready queue when it is made active, so the ready queue contains all **Processes** which are ready to run including those running. Existing Smalltalk systems have avoided this problem because only the **Process** which is actually running can query its status, so the answer to a **Process**'s question "Is **Process** x active?" depends only on whether the asking **Process** is itself x. We must note, however, that even the **canRun:** message is dangerous, because the status of the **Process** might change concurrently with the execution of some code that tries to manipulate it. Thus, in MS, the user-level is more likely to see a consistent view of the ready queue, but still is not guaranteed one. This reflects an unfortunate appearance within the Smalltalk-80 system of a lack of concern about concurrency which could affect even a uniprocessor system.

One reason that the **ProcessorScheduler** is the source of so many problems is that the Smalltalk-80 virtual image and the interpreter are closely intertwined at this point. The basic **Process** primitives manipulate the **ProcessorScheduler**, the interpreter must manipulate it asynchronously (in response to input events, for example), and it is completely exposed at the user level. While this makes the Smalltalk-80 system one of the few systems in which one can directly examine the ready queue, it also makes it very likely that one will see an inconsistent view of that queue. Unfortunately, some Smalltalk-80 code actually manipulates the ready queue directly. A revised implementation of the **ProcessorScheduler** should take concurrency into account and eliminate user-level manipulation of the ready queue. In this case, some of Smalltalk's flexibility must be sacrificed for safety.

4 Preliminary Results

We have used a subset of the standard Smalltalk-80 benchmarks [14], the "macro" benchmarks, in evaluating MS. These benchmarks measure the performance of a number of typical user activities, such as compiling code or searching for definitions or uses of a particular message selector.

Two versions of the interpreter are represented in the measurements. The version called "baseline BS" represents the state of the interpreter after it was modified to run on the Firefly under the V kernel but before any multiprocessor support was added. This serves as a reference point against which to measure the overhead introduced by the multiprocessor support. The version called "MS" represents the interpreter with all of the multiprocessor support included.

The MS system was run in three states: with one Smalltalk idle **Process** (in uniprocessor mode), with four Smalltalk idle **Processes**, and with four busy Smalltalk **Processes**. The idle **Process** used is the trivial expression `[true] whileTrue`. It is important to note that this expression is translated by the compiler into bytecode which neither looks up messages nor allocates memory. Hence, the idle **Process** represents the minimum possible interference with other **Processes**. By contrast, the busy **Process** is intended to represent the maximum possible interference. Based on the "sweep hand" background **Process** included in the Smalltalk-80 system, it includes message sends and object allocations, and also contends for the display.

The preliminary results are graphically illustrated in Figure 2, with the actual measurements in Table 2.[9] In Figure 2, the times for each benchmark test have been normalized to the time for that test on the baseline system, to ease comparison.

The current results are quite promising. The static cost of the serialization, replication, and reorganization of BS into MS is low: the architectural changes cost less than 15% in the worst case, and we believe that this figure can be reduced with fine tuning.

The cost of competitive multiprocessing in MS is moderate. With only trivial competition, an additional 30% of overhead appears in the worst case. This is more than we would like, but it may be possible to reduce it. Non-trivial competition drives the total overhead up to 65% in the worst case, about 40% on average. Our challenge is to identify the sources of this overhead and reduce or eliminate them. Although our analysis is not complete, we suspect that a significant amount of the overhead is due to contention in storage allocation, in which case replication of the new-object space should have significant

[9] Due to non-determinism in the system, the benchmark times are somewhat variable—variations in the same test of as much as 3% were noted, so differences of 3% or less should be discounted.

Figure 2: Preliminary overhead measurements—normalized

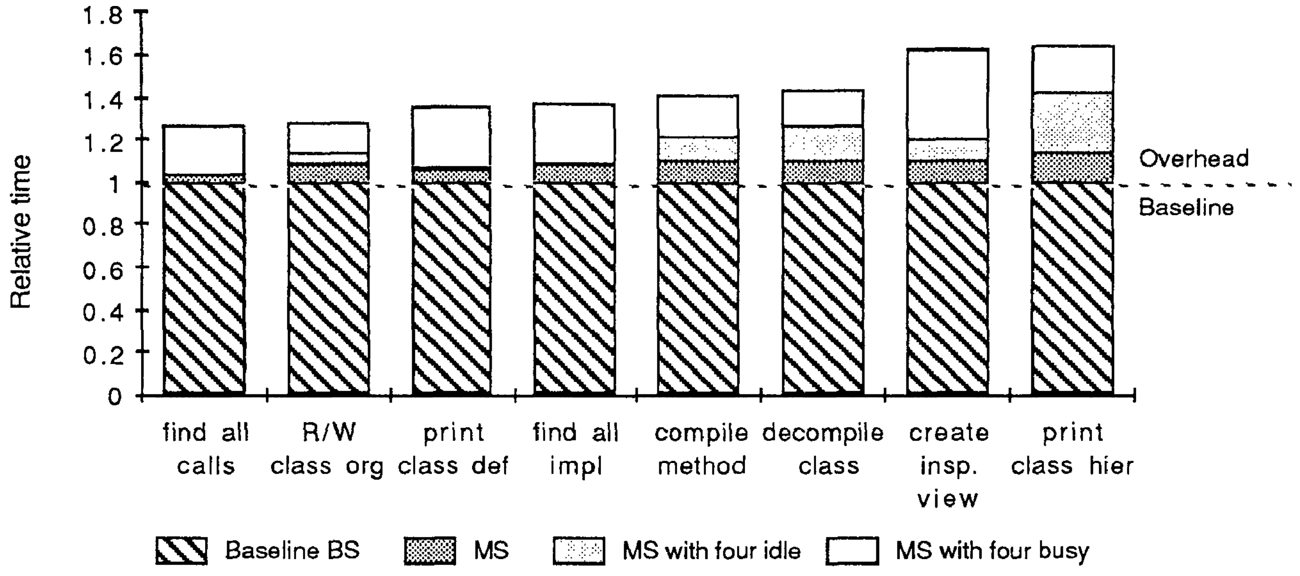

Table 2: Preliminary performance results

State	read and write class organization	print class definition	print class hierarchy	find all calls	find all implementors	create inspector view	compile dummy method	decompile class
Baseline BS on multiprocessor	14.3	8.1	10.0	26	8.2	6.1	22	12.7
MS on multiprocessor	15.6	8.6	11.4	27	8.9	6.7	25	14.1
MS with four idle Processes	16.3	8.8	14.3	27	9.0	7.4	27	16.1
MS with four busy Processes	18.4	11.1	16.4	33	11.2	10.0	31	18.2

All times in seconds, from standard benchmarks
Differences of less than 3% are not significant

benefits.

5 Conclusions

We have adapted an object-oriented programming environment that was originally designed to run on a uniprocessor to a shared-memory multiprocessor. We found that the task was less difficult than one might anticipate, for three reasons:

1. A small, flexible operating system, the V System, which provided support for multiple processes within a single address space and allowed inexpensive synchronization and communication.
2. A set of constraints which simplified the problem by minimizing changes to the programming language.
3. A small set of strategies—serialization, replication, and reorganization—which were surprisingly general in their application, as summarized in Table 3.

Table 3: Applications of the three strategies

Serialization	Replication	Reorganization
allocation	interpretation	active process
garbage collection	method caches	
entry tables	free contexts	
scheduling		

The performance of the resulting system is adequate, and we believe that it can be improved without introducing any new strategies, although we have seen that the choice of which strategy to apply in a particular case can be very important. We also believe that these strategies can be successfully applied to other attempts to adapt uniprocessor programming systems to multiprocessors, given appropriate support from the operating system.

Note that a different set of constraints could well have a significant impact on the behavior of such a system. Our future plans include gathering some data on the effects of such constraints in an attempt to characterize their influence.

6 Future Work

Our most immediate goals are to analyze the overhead in the current version of MS and to eliminate as much of it as possible, since the overhead in the present version, while acceptable, is still high. Subsequently, we expect to report on our experiences in using parallelism in MS, perhaps including some comparisons of various concurrent programming approaches. Finally, we plan to add sufficient instrumentation to MS to gather data about how different concurrent programming paradigms affect memory reference patterns and contention for resources, and how architectural constraints such as the choice of shared or non-shared memory influence the system.

7 Acknowledgements

This work would not have been possible without equipment and software supplied by Digital Equipment Corporation, Sun Microsystems, Inc. and ParcPlace Systems. The work was also supported by National Science Foundation grant number CCR-8657631, and by grants from Apple Computer, IBM, NCR, and Texas Instruments. Special thanks are due to David Cheriton and the Distributed Systems Group for their work on the V System and especially to Kieran Harty for making the V kernel run on the Firefly.

References

[1] Khayri Abdel-Hamid Mohamed Ali. *Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 1984. Published as TRITA-CS-8406.

[2] Brian M. Barry, John R. Altoft, D. A. Thomas, and Mike Wilson. Using objects to design and build RADAR ESM systems. In *OOPSLA '87 Conference Proceedings*, pages 192–201, Association for Computing Machinery, October 1987.

[3] John Bennett. *Distributed Smalltalk: Inheritance and Reactiveness in Distributed Systems.* PhD thesis, University of Washington, 1987. In preparation.

[4] John K. Bennett. The design and implementation of Distributed Smalltalk. In *OOPSLA '87 Conference Proceedings*, pages 318–330, Association for Computing Machinery, October 1987.

[5] David R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2), April 1984.

[6] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 366–374, June 1986.

[7] E. W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[9] Carl Hewitt. The Apiary network architecture for knowledgeable systems. In *Conference Record of the 1980 LISP Conference*, pages 107–117, August 1980.

[10] Carl Hewitt and Henry Lieberman. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[11] Hill, et al. *SPUR: A VLSI Multiprocessor Workstation.* Technical Report UCB/CSD 86/273, University of California, Berkeley, 1985.

[12] H. T. Kung and S. W. Song. *An Efficient Parallel Garbage Collection System and its Correctness Proof.* Technical Report, Carnegie-Mellon University, September 1977.

[13] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In Philip H. Enslow Jr., editor, *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, Computer Society Press of the IEEE, August 1976.

[14] Kim McCall. The Smalltalk-80 benchmarks. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 9, pages 153–173, Addison-Wesley, 1983.

[15] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[16] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[17] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 164–172, Computer Society Press of the IEEE, October 1987.

[18] David A. Thomas, Wilf R. LaLonde, and John R. Pugh. *Actra—A Multitasking/Multiprocessing Smalltalk.* Technical Report SCS-TR-92, Carleton University, May 1986.

[19] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, PA, April 1984.

[20] David M. Ungar and David A. Patterson. Berkeley Smalltalk: Who knows where the time goes? In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 11, pages 189–206, Addison-Wesley, 1983.

[21] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, University of Washington, January 1987. Published as Technical Report 87-01-03.

[22] Zorn, et al. *SPUR Lisp: Design and Implementation.* Technical Report UCB/CSD 87/373, University of California, Berkeley, 1987.