A Multiprocessor Implementation of Little Smalltalk

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902

89-60-15

A Multiprocessor Implementation of Little Smalltalk

Srinivas Raghvendra and Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, OR 97331

August 16, 1989

Abstract

In this project we have attempted to exploit the parallel programming features of a Sequent Balance multiprocessor to improve the performance of a Smalltalk interpreter. The *block* construct and the *fork* message together make Smalltalk a 'natural' parallel programming language. We describe how processes can be created from within the Smalltalk system to execute concurrently on different processors while sharing a common object memory. We have described a suite of programs that were run to test the implementation and we report the results of the benchmark tests.

1 Smalltalk and Parallel Programming

1.1 Object-oriented languages

Procedural or imperative languages such as Fortran, Pascal and C have been in use for a number of years now and have proved to be popular and adequate for most purposes. However, as computing systems have grown bigger and more complex and as the needs of the user have become more sophisticated, software development has become an error-ridden and costly process. Alternative styles of programming are being developed and tested to help overcome the problem that has come to be labeled as the "software crisis". The object-oriented method of programming is one such attempt at developing a different approach to software development. Object-orientation is the basis of languages such as Smalltalk, C++, Objective-C and Eiffel.

Object-oriented languages can be said to have evolved from the language Simula that first introduced the notion of classes. A class is an unit of classification. It is an abstraction that helps conceptualize objects with certain properties in common as being "similar" or as being of "the same type". Object-oriented languages combine the description of data and procedures within a single entity called an *object*. An object is a well-defined data structure coupled with a set of operations that describe specifically how that data can be manipulated [Micallef88]. An important concept in object-oriented languages is that of inheritance. *Inheritance* permits classes to have access to procedures and data in other classes. Inheritance is a major feature offered by object-oriented languages.

Object-oriented languages are characterized by their support of encapsulation, reusability and extensibility. *Encapsulation* is the strict enforcement of information-hiding. *Reusability* is the ability of a system to be reused, in whole or in parts, for the construction of a new system. *Extensibility* is the ease with which a software system may be changed to account for modifications of its requirements

[Micallef88]. Another characteristic of object-oriented languages is that they facilitate the creation of software components that closely parallel the application domain, an important feature for building inexpensive, understandable systems.

1.2 Smalltalk

Smalltalk is a language developed at Xerox PARC by the Xerox Learning Research Group as a part of Alan Kay's Dynabook project [Kay77]. Smalltalk is built on the model of communicating objects and was the first computer language to be based entirely on the structural concepts of messages and activities.

Smalltalk owes its elegance and power to its uniformity. Smalltalk is designed around the concept of objects and everything in the Smalltalk system is an object. This is similar to the philosophy behind LISP, in which everything is a list. In Smalltalk, the interaction between the most primitive objects is viewed in the same way as the highest-level interaction between the computer and its user.

The uniformity that results from considering everything in Smalltalk as objects provides a framework in which complex systems can be built. Several related organizational principles contribute to the successful management of complexity. These include [Ingalls81]:

Modularity No component in the system should depend on the internal details of any other component.

Classification A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the kernel classes of the system.

Polymorphism A program should specify only the behavior of objects, not their representation.

Factoring Each independent component in a system should appear in only one place.

1.3 Parallel Programming and Object-Orientation

Over the last few years, a number of multiprocessor machines have become available on the market. A partial list of vendors for such machines would include Sequent, Alliant, Encore, Floating Point Systems, Cogent, Tandem, and Amdahl. These machines are reliable and affordable and offer popular operating systems such as UNIX [1]. However, the full potential of these machines has been found difficult to exploit due to the lack of appropriate software and programming environments for multiprocessors. Attempts are being made in various directions for the realization of a truly productive environment or language for the development of software for multiprocessors. One approach that has been attracting considerable attention is the object-oriented methodology. Object oriented languages seem to lend themselves naturally to implementation on multiprocessor machines. As Yonezawa and Tokoro have pointed out in [YoneToko87]

> ...the central issues in exploiting parallelism are what and whose activities are to be carried out in parallel and how such concurrent activities should interact with one another. In designing a software system that exploits parallelism, these issues boil down to the problem of how the system should be decomposed into components that can be activated in parallel and what function should be given to each component. To maintain system transparency, the decomposition should be natural and modular. Then, in what style or form should each component be represented ? ...the notion of "objects" seems to be a highly promising form.

[1] UNIX is a trademark of AT&T Bell Labs.

It is interesting to consider why parallel programming is difficult in conventional programming languages. A colorful description of the shortcomings of the conventional computing models is quoted below from [Budd87]:

> The traditional model describing the behavior of a computer executing a program is the process-state or "pigeon-hole" model. In this view, the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various slots (memory addresses), transforming them in some manner and pushing the results back into other slots ...

In his 1978 Turing award lecture, John Backus [Backus78] offered a related set of shortcomings of conventional computing models. He calls the world of conventional programming languages a world of statements with the assignment statement at its center. He goes on to propound a new language that is strictly functional in nature and does away with the assignment statement. Whether or not one agrees with Backus about his extreme views about the assignment statement, there is no gainsaying the fact that assignment statements do make for a languages that have unneeded data dependencies and consequent serializability problems.

Yet another view of the problem is offered by Chandi and Misra [ChanMisra88]. They observe:

> The basic problem in programming is the management of complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is to be written, and the hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved, postponing considerations of architecture and language constructs.

We now need to consider why object-oriented languages are an improvement over conventional languages for programming in general and for parallel programming in particular. In our opinion, there is a two-fold advantage — the improvement in ease of programming and the improvement in ease of maintenance. Maintainabiltiy of programs is improved by properties such as polymorphism and inheritance. Programs also become easier to write and debug because of the localization of operators that can affect a data structure. Because of the encapsulation of methods for data manipulation along with the data itself, it is easier to understand written code in a program. It is true that object-oriented languages do not dispense with the assignment statement that is Backus' *bête-noire*. The retention of assignment statement can be rationalized on the grounds that it is not the assignment statements *per se* that are bad. It is the unrestricted usage of these statements that make programs hard to write and understand. Object-oriented languages impose a discipline on the usage of these statements and hence overcome a large part of the problem that exist in conventional languages.

The above hypotheses are being tested out by researchers who are trying to add concurrency to object oriented languages. With its true object-orientation and wide availability, Smalltalk has been the natural focus of a fair number of such attempts. There are also other reasons why Smalltalk has been a language of choice for these experimentations. We consider this in the next subsection.

1.4 Why Smalltalk

A major — and justified — complaint about Smalltalk is its speed, or actually the lack of it. If a particular algorithm is coded in a language such as C and in Smalltalk, the C implementation would almost certainly be faster than the Smalltalk implementation. Very approximate benchmarks have

indicated a decrease in speed by nearly a factor of 100 for implementation in Smalltalk as compared to a C implementation. This is a very real constraint to the widespread usage of Smalltalk. To a very large extent, the decrease in speed has to do with the interpreted method execution and the very late binding in Smalltalk. These are almost insurmountable problems. Attempts have been made to have compiled Smalltalk [Johnson88] but the syntax for such implementations of Smalltalk is very different from 'standard' Smalltalk and the ease of programming that one finds in standard Smalltalk disappears in such artificial implementations.

The problems with Smalltalk that we mentioned above may ensure that Smalltalk never becomes a 'production' language in which software such as word-processors is written for the application software market. Even if this prediction does come true, there are many other applications in which Smalltalk can be used. One such application that is appropriate as an example is the software that is written for research in the physical sciences. Such projects usually involve large amount of data and such processing is almost always done with computers. Software for these projects is usually written in C or Fortran, not so much because they are the best languages for development, not because they are the best languages for such applications, but because they are widely available and are fast enough for such purposes. The ideal language for these projects would be Smalltalk because it allows rapid-prototyping and thus allows software to be developed quickly and modified easily as the requirements keep changing. Speed and availability are problems that hinder such use. We suggest that with the availability of public domain software such as Little Smalltalk, the availability problem will disappear. We also hope that with the wide availability of multiprocessors and the newer versions of the Smalltalk interpreter that run on multiprocessors, the execution speed of the application written in Smalltalk should become tolerable.

A discussion of parallelism and languages is incomplete if the issue of style of parallelism is not reviewed. There are two styles of parallel programming — the implicit style and the explicit style. The *implicit style* is characterized by programs that do not contain explicit constructs to aid parallelism. The programmer is free from having to think about parallelism. The compiler does an exhaustive analysis of the program and decides which parts can be executed in parallel. *Explicit parallelism* is characterized by programs that offer explicit hints to the system to execute certain parts of the program in parallel. A Fortran implementation, for instance, may have a *DO_ACROSS* construct that hints that a *DO* loop can be executed in parallel.

In trying to decide whether to use implicit or explicit parallelism, we had to consider two issues. First, the Smalltalk language has parallel programming constructs built into the language in the form of *blocks* and the *fork* message. Smalltalk programmers routinely use these features even when programming for uniprocessor machines so that the programmers need not learn a new syntax to program in parallel. Thus, the parallelism, though explicit, is naturally expressed in an elegant manner without any additional burden for the programmer. Secondly, a perusal of literature on the subject indicates that most implicit parallelism implementations have not been able to achieve the performance of explicitly coded parallel programs. We therefore chose to use the explicit method of parallelism for our implementation.

1.5 Description of the project

In this research project, we have attempted to parallelize a Smalltalk interpreter to run on a Sequent Balance multiprocessor running a version of the UNIX operating system. We have measured the performance of the original implementation and the multiprocessor implementation and have reported performance improvements that can be attributed to the parallel implementation.

2 Bringing Concurrency to Smalltalk - a literature survey

2.1 Introduction

We noted in the previous section that a fair number of attempts have already been made to provide a degree of concurrency to Smalltalk and other object-oriented languages. In this section we review some of the work done in this area. We have restricted this survey to implementations of the Smalltalk language only. We have also excluded those implementations that attempt to solely provide a *distributed* version of Smalltalk as against our idea of a *concurrent* Smalltalk. For the purposes of our discussion, we define a distributed system as having a disjoint memory space over more than one machine. We define a concurrent system as being resident on single (multiprocessor) system and sharing a common memory space. Thus we distinguish between the two based on the tightness of coupling of the individual processes constituting the whole Smalltalk system.

2.2 CST

Dally and Chien, in their paper "Object Oriented Concurrent Programming in CST" [Dally88], describe the implementation and syntax of a language that they call CST. CST is based on Smalltalk 80 but supports concurrency using locks, distributed objects and asynchronous message passing. CST differs from most other implementations in that it supports distributed objects. The idea of a distributed object is not so much to enable the object to be distributed over space but to enable the same object to be addressed by multiple objects simultaneously without creating a bottleneck. Each constituent of a distributed object can accept messages independently.

CST has a LISP-like syntax, but is otherwise closer to Smalltalk in other aspects. Concurrency is achieved in CST by allowing subexpressions to execute concurrently and independently. For example, in the following

```
(cset subint1 (integrate self midpoint epsilon selector)
(cset subint2 (integrate midpoint hpoint epsilon selector)
(+ subint1 subint2)
```

the first two expressions execute concurrently and the third waits for the first two to complete.

2.3 Actra

David Thomas, et al. in their paper "Actra - A multitasking/multiprocessing Smalltalk" [ThomDav88], describe the design and implementation of Actra, a version of Smalltalk designed to support multiprocessing. To put their work in the right perspective, we quote the following from their paper

> Smalltalk processes provide traditional coroutine based uniprocessor multitasking with semaphore synchronization. Processes are created from blocks (closures) and are not first class objects. They cannot be sent messages. Processes are implemented as co-routines.

In order to support true multiprocessing, objects of a new type called 'Actors' have been introduced. Actors are first class objects. Actors encapsulate a community of objects intended to execute concurrently with other communities. Every actor is a task and has the granularity of light weight processes. Actors execute with blocking semantics. Because Actors are first class objects, they can be specialized into categories such as clients, workers, servers, *etc.* for use in process structuring and

anthropomorphic computation. Concurrency is realized by making the communication between an actor and a non-actor object to be non-blocking. Actor-Actor communications, however, is blocking in nature.

Actra is implemented on top of the Harmony real-time kernel. Harmony is described in [Gentle84]. From the view point of the kernel, every actor is a task. The body of the task is an instance of the Smalltalk virtual machine. The state of the virtual machine is kept in local data for each task instance. The Garbage Collector is also a separate task that runs at high priority and executes whenever requested by a message from the virtual machine.

2.4 Concurrent Smalltalk

Concurrent Smalltalk has been proposed as a means of adding concurrency to Smalltalk by Yokote and Tokoro [YokoToko87]. Concurrency is proposed to be achieved by providing concurrent constructs and atomic objects. To get an idea about how Yokote and Tokoro view Smalltalk in its standard form and what they think should be changed to achieve concurrency, the following extract from their paper is significant

> Smalltalk uses the notion of 'process' to describe a problem which contains concurrency. A process is created by sending a fork message to a block context. However, this decision eventually imposed upon the programmer the cumbersome labor of modelling the problem in two different level modules : objects and processes. This impairs descriptivity and understandability.

In Concurrent Smalltalk, objects and processes are unified into a single notion of concurrent objects. There are two methods to express concurrent execution - (i) let the receiver continue execution after it returns a reply and (ii) send a message and proceed with execution without waiting for a reply message. There are also two kinds of communication mechanisms (i) synchronous method calls and (ii) asynchronous method calls.

To help implement asynchronous calls, a new object called CBox object has been introduced. A CBox object is returned when an asynchronous call is made. The reply is retrieved from the CBox using a receive call. If the reply is not yet ready when receive call is made, the caller blocks on reply. CBox objects support the notion of 'transaction keys'. Transaction keys can serve as passwords for transactions and can be used to group together sets of messages.

In Smalltalk 80, when many messages arrive at an object simultaneously, a new context is created for each request, and each request is processed in its context independently from any others. The execution is in a LIFO (Last In First Out) manner.

Concurrent Smalltalk provides non-atomic objects to maintain compatibility with Smalltalk 80. These objects follow the above mentioned (LIFO) behaviour. Concurrent Smalltalk also provides atomic objects which have serially accepted/executed non-interferable property. Several messages to one object are serialized in FIFO order.

Asynchronous method calls are created by appending an ampersand '&' to a message. Thus, the following is an asynchronous method call

```
| t1 |
t1 ← db lookup: #key &
```

The CBox returned by this call is associated with t1.

2.5 Distributed Concurrent Smalltalk

Distributed Concurrent Smalltalk is an extension of Concurrent Smalltalk. Distributed Concurrent Smalltalk has been proposed by Nakajima, Yokote, Tokoro and others [Nakajima88] at the Keio University in Japan. The main refinement in Distributed Concurrent Smalltalk over Concurrent Smalltalk is the introduction of the concept of a distributed object model. Distributed Concurrent Smalltalk uses an implicit process creation facility as against an explicit facility in Smalltalk ('fork'). A process is created by an asynchronous method call. The authors claim this is better idea because they believe that an explicit call to create a process creates a dichotomy in the abstraction. Programmers have to deal with objects and processes as seperate constructs.

Distributed Concurrent Smalltalk has an unusual object structure in that objects support multiple activities with more than one thread of control. Objects execute messages as soon as they are received. Objects need a synchronizing mechanism among threads in order to preserve a consistent state.

The computational model of Distributed Concurrent Smalltalk has five entities. Object and Message have their usual definitions. Activity is what an object creates to execute the method specified by a message. An object can contain more than one activity. Thread is a sequence of activities created by a synchronous message send. Finally, there are Thread Groups which are sets of threads. Tasks such as in transaction processing are represented by Thread Groups.

2.5.1 Object Structure

An Object is made up of a State Object and a Secretary Object. Any message to an object is first sent to the Secretary Object. The secretary creates an activity to execute the method associated with the message unless there is to be a delay due to conditional or exclusive synchronization. The name space in Distributed Concurrent Smalltalk is divided. Each user can have his own name space. Name spaces are implemented as dictionaries and there is a 'root' Super Name Dictionary.

2.5.2 Synchronization Mechanism

There are two kinds of synchronizing mechanisms. 'Method Relation' serializes several activities by defining the exclusive relationship between two methods. If there is no such relationship between two objects, they can be executed concurrently. When a Secretary object gets a message, it checks to see if this message has an exclusive relation with the currently executing methods. If so, execution of the new message is delayed till the older methods finish execution within the object. The other mechanism is the 'Guarded Method'. Each method may have an Ada-like guard represented by a block. The block contains a conditional expression. When a method is to be executed, this condition is first checked. If it is false, execution is delayed until it becomes true.

2.6 Comments and Conclusion

Four different approaches for providing concurrency in Smalltalk have been reviewed. They differ significantly in the manner in which concurrency is sought to be achieved. Important among the differences is the level at which concurrency is attempted. At least one implementation has tried to provide concurrency right down at the object level while another tries to make objects capable of handling more than one message simultaneously. Most approaches seem to include some new syntactic constructs. The designers have included new classes of objects and new abstractions that seem, at times, far removed from the original Smalltalk paradigm. In particular, the introduction of the 'Actor' class of objects in Actra [ThomDav88] is a radical departure from the standard

Smalltalk hierarchy. Nakajima and others [Nakajima88] argue eloquently about the undesirability of the dichotomy in abstraction caused by having processes as non-objects. However, they go on to suggest two different kinds of message passing - synchronous and asynchronous. This reviewer believes that the dichotomy caused by having two different types of messages is more serious and unsettling than the process-object dichotomy. Dally and Chien [Dally88] have not been very clear about the syntax of CST but it does appear that CST has an equivalent of the CBox object suggested by Yokote and Tokoro in [YokoToko87].

Significantly, the approaches surveyed have suggested changes to the syntax of the standard Smalltalk language. This may be an incorrect approach to take as it could make a lot of code already written unusable. While it certainly is more fashionable to suggest new languages or new syntactic changes to older languages, this reviewer is of the opinion that it might be more productive to devote time to study the feasibility of bringing concurrency to languages without requiring syntactic changes. In particular, Smalltalk, notwithstanding its quirks and warts, has a certain conceptual unity to it that is in danger of disappearing with the changes being suggested to its syntax.

From this review we can see that the goal of bringing concurrency to Smalltalk has interested quite a few researchers. We see that the objective — that of delivering the best of the object-oriented and the parallel processing world to the user — is elegant and appealing. But it is being approached in ways that would involve important changes to the Smalltalk syntax. We would like to explore an alternative method of bringing concurrency to Smalltalk without requiring any changes to its syntax.

3 The Little Smalltalk System

3.1 About Little Smalltalk

Little Smalltalk is a Smalltalk system developed by Timothy Budd [Budd87]. Little Smalltalk is very close to Smalltalk-80 [2] in syntax but its user interface is different from Smalltalk-80 in that Little Smalltalk does not offer the elaborate graphical user interface that Smalltalk-80 does, nor does it offer the 'browser' and other aids of the Smalltalk-80 programming environment. Efforts are currently being made to offer a friendly graphical interface. The absence of a graphical user interface in the standard Little Smalltalk system in not an oversight. It is a deliberate omission made in order to support the main objectives of developing Little Smalltalk. The objectives of developing the Little Smalltalk system have been enumerated in the preface to Budd's "A Little Smalltalk" [Budd87]. They include the following:

- the system should support a language that is as close as possible to the Smalltalk-80 system.
- the system should run under UNIX using only conventional terminals.
- the system should be written in C and *be as portable as possible.*
- the system should be small and should work on 16-bit machines.

A feature of Little Smalltalk that needs to be emphasised is its portability. Written initially to work on the UNIX system, it has been subsequently reworked to make it portable across a wide range of operating systems and hardware. The Little Smalltalk system is written in C and in Smalltalk. Little Smalltalk is in the public domain and is freely available from many sources in many parts of the world.

[2] Smalltalk-80 is a trademark of the Xerox Corporation

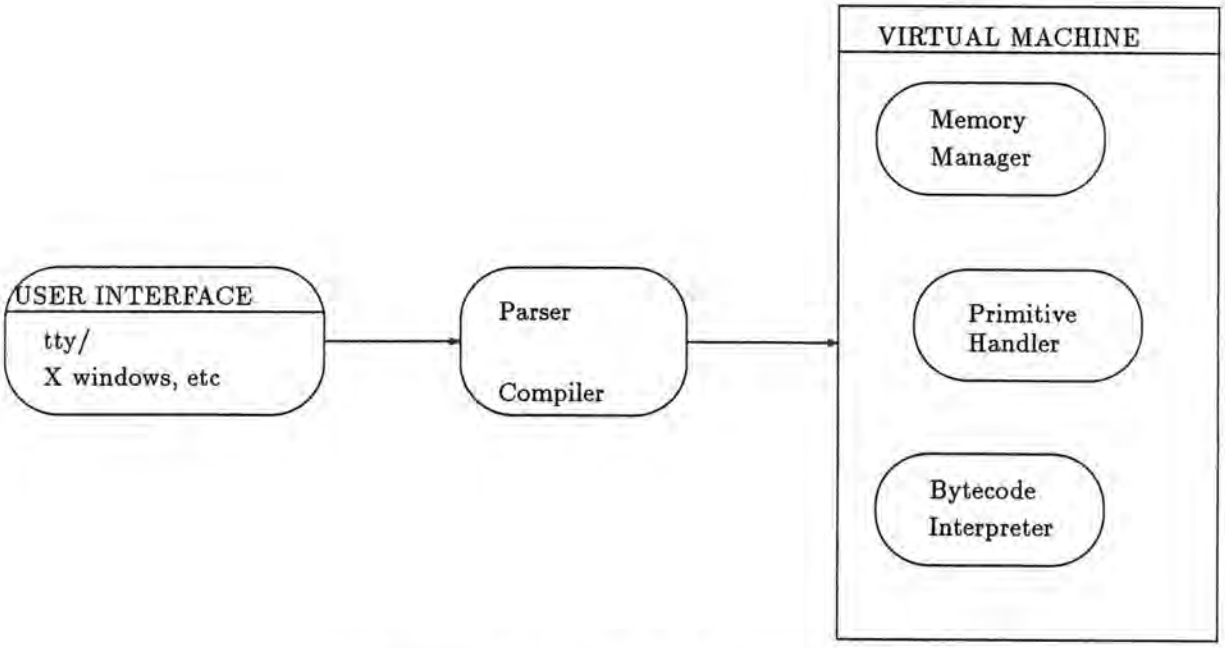

Figure 1: The Little Smalltalk system

3.2 Architecture of the Little Smalltalk system

Traditionally, interpreters are implemented by defining a *virtual machine* for a simple language and then translating the higher level language into instructions for this simple machine [Budd87]. Little Smalltalk has also been implemented in this manner. The *interpreter* is the control program that controls the virtual machine. The 'machine instructions' for the virtual machine are in the form of *bytecodes*.

The Little Smalltalk system consists of a front-end that accepts input and converts it into bytecodes for the interpreter. This front-end consists of a parser and a compiler that compiles parsed input into bytecodes. For the Little Smalltalk system, the parser is a simple *recursive descent* parser. For the actual user interaction (keyboard input/screen output) Little Smalltalk has a standard "tty" interface, but newer versions of Little Smalltalk have an optional graphical interface such as a X windows interface or a Macintosh graphical interface.

The interpreter is the part of the virtual machine that performs the actions described by the bytecodes of the methods *i.e.*, the machine code of the virtual machine. The information needed to implement the interpreter is the description of the bytecodes, the representation of the methods and the technique to find the method to run in response to a message [Krasner81].

To augment and aid the virtual machine, a *memory manager* that manages object memory and a *primitive handler* that interfaces the Little Smalltalk system with the operating system are also implemented. These are described in later sections.

3.2.1 The Bytecodes

The bytecodes define the virtual machine as a *stack-oriented* machine. Bytecodes represent various operations that the virtual machine can perform. These include the following:

- push an object on to a stack
- pop an object from a stack
- branch to a location
- send a message, and others

It must be observed that some procedural languages are also implemented on stack-based machines. The difference between a Smalltalk virtual machine and a procedural virtual machine is the way a procedure (which is the equivalent of a 'method' in Smalltalk) is found. In conventional, procedural stack-based machines, the address of the procedure to be called is determined at compile time. In the Smalltalk virtual machine, only the name of the procedure (method) is determined at compile time. At run time, the appropriate method to be executed is found through a strategy involving the receiver of the message and its class [Krasner81]. This is a form of 'late binding'.

3.2.2 Representation of Methods

The representation of methods is simple. Bytecodes that form the compiled equivalent of a method are stored in arrays of bytes. The only complication involved is in addressing variables. In source code, a method contains variable names and literals but the bytecodes are defined only in terms of field offsets. There are three types of variables — variables local to a method, called *temporaries*, variables local to a receiver of a message, known as *instance variables* and variables found in the global dictionary, known as *global variables*. The compiler translates references to these variables into field offsets of the temporary area, the receiver, or the global table. The *global dictionary* is a system maintained table that is useful in keeping track of global variables. Smalltalk has a few global variables that are used mostly by the system.

3.2.3 Finding the right Method

To find a method associated with a message, the dictionary associated with the class of the receiver is first checked. If an appropriate method is not found, the superclass of the current class is checked and so on until no more further classes remain or the required method is found. An optimization that is done in Little Smalltalk is the cacheing of recently used methods. The capacity of the cache is about 200 (this is a configurable parameter). Thus, the 200 most recently used methods are found directly in the cache without the usual hierarchical search. Since methods tend to be reused often, the cache leads to better system performance.

3.2.4 The Memory Manager

The *memory manager* helps allocate memory for various objects. It also reclaims memory from objects that are not being used. Memory reclaimation is by the 'reference count' style of garbage collection. This and other operations performed by the memory manager and the detailed structure of object memory in Little Smalltalk is explained in detail in the next section.

3.2.5 The Primitive Handler

The *primitive handler* is the part of the virtual machine that interfaces with the native operating system and hardware. Primitives are 'methods' in native machine code that are needed to terminate the recursion of message sending that the Smalltalk paradigm conceptually implements. Primitives are also needed to optimize certain often used methods. In the Little Smalltalk system primitives are written in C and compiled to machine code. Examples of operations done using primitives include basic arithmetic and bit operations, string operations, file operations, process management and optional interfacing such as graphic user interfaces.

3.3 Summary

This section gives an overview of the architecture and implementation of the standard Little Smalltalk system as available for various hardware and operating systems. In the sections of this report that follow, this implementation is referred to as the 'uniprocessor implementation' to distinguish it from the multiprocessor implementation that has been the result of this project. The next section describes the multiprocessor implementation in detail.

4 The Implementation

4.1 The Sequent Balance architecture and computation model

For this project, the Smalltalk interpreter was implemented on a Sequent Balance multiprocessor system. In order to understand the design of the new interpreter, it is necessary to understand certain details about the hardware on which it is implemented and the programming paradigm that the hardware encourages.

4.1.1 The Sequent Hardware

The Sequent Balance is a shared memory multiprocessor system that can accommodate between 2 and 30 processors. The processors are all connected to a common memory by a shared 32-bit bus. Each of the processors is a National Semiconductor NS32032 32-bit CPU. The availability of multiple processors connected together by a common bus to a large shared memory promotes tightly coupled parallel processing. Hardware support is provided for mutual exclusion in the form of Atomic Lock Memory (ALM) at the user level and in the form of 'gates' controlled by the System Link Interrupt Controller (SLIC) at the system level. An interesting feature of the hardware is the provision for 8K byte RAM on each processor board. This RAM is known as the 'local RAM' and contains copies of the kernel interrupt vectors, module table, first-level page table and time-critical kernel code. Each processor board also contains a 8K byte, 2-way set-associative cache for each CPU [Sequent85].

4.1.2 The Operating System

The Dynix [3] operating system runs on the Sequent Balance. Dynix is derived from UNIX 4.2 BSD. Dynix supports a 'dual universe' in the sense that it offers facilities of both the BSD and AT&T System V systems. The Dynix command interface is almost exactly like that of a standard UNIX system. However, to enable easy and efficient usage of the multiple processors, Dynix offers some

[3] Dynix is a trademark of Sequent Computer Systems, Inc.

simple extensions to the standard UNIX process model, a set of language extensions and run-time library support [BecOli89].

The processors on the Sequent have a common queue from which processes are chosen for execution. Each CPU decides for itself when it should stop executing its current process and which process it should switch to next. Sometimes however, when one CPU sees a high-priority process become available, that CPU 'nudges' another CPU that is running a lower priority process. This 'nudge' consists of an interrupt sent via the SLIC bus. Every tenth of a second, in response to a timing interrupt, a CPU takes the task of finding one running process that can be preempted. This 'scheduling CPU' checks to see whether any high-priority process is waiting for a CPU. If the priority of the waiting process is greater than or equal to the priority of the running process, the scheduling CPU nudges the CPU that is running the lowest-priority process [Sequent85].

In the multiprocessor Dynix system, several processes (each on a different CPU) may be executing in the kernel simultaneously, all accessing the same kernel data structures. To ensure that one CPU does not try to modify a data structure while another is using it, the Dynix operating system implements mutual exclusion mechanisms that enable a CPU to gain exclusive access to a data structure. These mechanisms are gates, locks and semaphores. The Dynix kernel uses the SLIC gates as its fundamental mutual exclusion mechanism in the same way that user programs use Atomic Lock Memory (ALM).

4.2 The new Smalltalk system

Drawing upon the hardware and software support described in the previous subsections, the new interpreter enables parallel processing. Multiple processes can be created from within the Smalltalk system and all these processes run in parallel on separate processors (within the limits of practicality imposed by the native operating system and hardware). To enable this parallel processing, two major modifications had to made to the original system. We made one set of modifications to the process scheduler and the other set of modifications to the memory manager. These two sets of changes are described in the next two subsections of this section.

4.3 Changes to the Process Scheduler

The Smalltalk system supports multiprocessing by encouraging the "process" abstraction. A process is a thread of execution that, conceptually at least, executes in parallel with other Smalltalk processes. The process scheduler in the original interpreter schedules processes in a round-robin fashion, preempting each process as it exhausts a pre-determined time slot. The original Smalltalk system is thus able to simulate multiprocessing.

In our present implementation, we have more than one processor at our disposal and we would like to have more than one process running concurrently. In the original system, there is only one process that represents the Smalltalk system at the operating system level. Processes created within the Smalltalk system by using the 'fork' message do not create new processes at the operating system level. The operating system was thus not involved in the scheduling of the Smalltalk 'processes'. In other words, the Smalltalk system is just one physical process at the operating system level and processes created within the Smalltalk system using the 'fork' message results in logical processes that were managed and scheduled internally by the Smalltalk system. In our multiprocessor implementation, we had to promote each Smalltalk process to the level of an operating system process to be able to achieve multiprocessing. True multiprocessing can only be achieved if more than one processor is concurrently executing code from one Smalltalk system. This is possible only if we

invoke operating system primitives to fork off processes at the operating system level. In Dynix, this is a necessity because the unit of load distribution is the process.

Once a Smalltalk process is mapped on to an operating system level process, certain operations that used to be performed by the Smalltalk system by itself are no longer needed as these are now done by the operating system. These operations include scheduling, suspension, and termination of processes. Scheduling and suspension are done by the operating system as for any other normal UNIX process. Termination is enabled by a new Smalltalk primitive. Letting the operating system take care of scheduling processes has also eliminated the need for a number of methods in the classes **Process** and **Scheduler**. These include methods such as 'execute' and 'resume' in class **Process** and 'addProcess:', 'removeProcess:' and 'currentProcess' in class **Scheduler**. In addition to decreasing code size, these changes have also helped in delegating responsibility from the Smalltalk system to to the operating system. Operating systems are designed to do process scheduling in the most optimal manner and by delegating responsibility, we avoid duplication of functionality in the operating system and in the Smalltalk system.

To summarize, we have modified the process scheduler by dispensing with it altogether. Smalltalk 'processes' are now full-fledged UNIX processes and are created with a **fork()** call. The operating system takes care of scheduling and preempting each process. The Smalltalk system has no explicit control over a process after the process is created. This latter feature must be addressed further in later implementations. It is necessary to be able to prioritize processes and to be able to explicitly suspend or resume processes from within the Smalltalk system.

4.4 Changes to the Memory Manager

A necessary requirement for the multiprocessor version of the Smalltalk system is that the various processes be able to share the object memory. Thus, regardless of the number of processes forked off by the Smalltalk system, each of them must be have access to every object in the system. This also implies the existence of a single name space over all the processes. An implication of having a single shared object memory is the potential for data corruption due to conflicting concurrent updates.

Before we discuss modifications made to the system to enable the multiprocessor implementation, it is necessary to describe the original system. The basic unit of memory usage in the Smalltalk system is the object. Everything in the system is an object. Because of its omnipresence, the structure of the object is very important. We next describe the structure of an object.

4.4.1 Structure of an Object

An object is a portion of memory that has four distinct fields. The first of these is a pointer to the object that represents the class to which this object belongs. The second field contains the reference count. The reference count is used for garbage collection and is discussed later. The third field gives the size of the object in the sense that it gives the number of instance variables that an object has. The fourth field is a list of pointers to objects that are instance variables of this object. The C code that represents structure of an object is given in Figure 4.1.

The design of the object structure is very important from the efficiency point of view. The design should provide for the least possible overhead in memory usage. This goal is met admirably in the Smalltalk system and the design presented above is nearly optimal.

An important structure in the management of objects in memory is the **objectTable**. The **objectTable** is the structure that holds all the objects in system. There is a (configurable) upper limit on the number of objects that the **objectTable** can hold. When we speak of a 'pointer' to an object, we actually mean the position of the object in the **objectTable**. In our implementation, the

```
struct objectStruct {
        object class ;
        short referenceCount ;
        short size ;
        object *memory ;
        } ;
```

Figure 2: C code representing structure of an object

objectTable is an array of structures so that the 'pointer' to an object is actually the index into this array.

4.4.2 Modifications to the Memory Manager

Two major modifications were required for implementing the memory manager for multiprocessors. The first modification was that memory for every object be taken from a shared memory area to ensure that all processors have access to every object. The second modification was to implement locking mechanisms to prevent data corruption.

Ensuring that all objects are in shared memory is quite simple. The Dynix parallel programming library has extensions to the standard UNIX memory management facility. **Shmalloc()**, for example, is an extension of the **malloc()** call and allocates memory from the shared memory pool. Apart from support for dynamic memory allocation, static shared memory can also be allocated by prepending the keyword 'shared' to a variable declaration. Using these facilities, ensuring a shared object space between processes is straightforward.

The Dynix parallel programming library provides a locking mechanism that can be used to prevent data corruption due to concurrent access. Locks can be initialized with a **s_init_lock()** call, locked with a **s_lock()** call, and unlocked with a **s_unlock()** call. Using these locks to prevent concurrent access is a simple task, but ensuring that such use does not lead to deadlocks is more difficult to ensure. A later section discusses this problem at length.

In our implementation, locks are used to serialize access to internal data structures such as the **objectTable** and the **objectFreeList**. It was tempting to allocate one lock per object and implement locking at the object level. We next discuss why we did not choose to implement locking in that manner.

It is not possible to perform an atomic test and set operation on ordinary memory [4]. It is necessary to use a central lock (in a special area in memory) to perform this operation. We are of the opinion that it takes about the same time to update the object value as it would to set and release a per-object lock. Hence, instead of using a per-object lock, we use a central lock and ensure that the update operation is done using the minimum number of instructions.

4.4.3 Changes to the synchronization mechanism

Semaphores provide the basic synchronization support in the Little Smalltalk system. In the original implementation, semaphores were implemented in memory using no special hardware mechanisms. Each semaphore would contain an integer variable that signified the state of the semaphore. Each

[4] Test and set operations can be done only in restricted areas of memory known as Atomic Lock Memory (ALM).

semaphore would also maintain a list of processes waiting on a the semaphore and would schedule processes for execution from this waiting list in a FIFO fashion.

This simple implementation was sufficient and correct for a uniprocessor implementation, but on a multiprocessor this implementation would fail for the lack of atomicity in the semaphore operations. In the multiprocessor implementation therefore, semaphores are implemented by using the semaphore facility provided by the operating system. Primitives have been implemented that translate Smalltalk semaphore operations into system calls such as **semget()**. Enqueuing and dequeuing of processes waiting on semaphores is now handled by the operating system. As a result of the changes in the implementation, the synchronization mechanism is now robust and the methods for implementing the semaphore operations have been simplified.

5 Performance evaluation and analysis

One of the main objectives of this research project has been to study the performance benefits that can result from a multiprocessor implementation of the Little Smalltalk interpreter. It has been our thesis that such an implementation can result in speed-up of the interpreter. To test this hypothesis and to quantify the actual improvement in performance, tests were conducted on the interpreter. This section describes the tests, the results obtained and their analysis.

5.1 Methodology

Two types of programs were used to test the interpreter. The first type attempted to demonstrate the correctness of the synchronization and multiprocessing primitives provided by the interpreter. To this end, the programs heavily exercised the semaphores provided by the interpreter. The fork primitive and the (implicit) locking mechanism were also exercised. Note that the 'correctness' of the primitives was sought to be established by running a large number of test cases. No attempt was made to use formal means to establish correctness. The well-known Readers-Writers problem was implemented to test the primitives. Though this suite of tests was primarily meant to check correctness, execution time records were maintained and are presented in a later subsection.

The second set of tests was designed to execute for longer periods and to do substantial amounts of CPU bound computation. In designing and using these tests, the idea is to set up a large computational job initially that would typically take a few minutes to execute using a single processor. The next step is to design an algorithm so that this task can be broken up to enable parallel execution. The final step is to code this algorithm in Smalltalk and to then use the multiprocessing features to speed up the computation. We used the problem of finding the number of primes between two given integers as a good compute-intensive job for this type of test. Execution time was measured and recorded for each trial with this suite of tests. The timings are presented in a later subsection.

5.2 The Readers Writers problem

The Readers Writers problem is a classic in operating system theory. It is an exercise in devising and implementing a satisfactory solution to the problem of enforcing certain constraints on a set of reader- and writer-processes to ensure data integrity. There are variations of this basic idea such as those that allow more than one reader to read data at the same time and those that give preferences to readers over writers and so on. Our implementation is based on the material in [MaeOldOld 87] and [Deitel 84]. The program, as tested, has m readers and n writers working together to process k pieces of data. To obtain a suite of test cases, we have used successively differing values of m and n while holding k constant. This suite of tests was first tested on the multiprocessor version of the

	sequential implementation (min:sec)	concurrent implementation (min:sec)
case 1	1:67	3:05
case 2	1:38	1:33
case 3	2:24	0:33
case 4	2:29	0:32
case 5	2:54	0:32
case 6	4:62	1:28

Table 1: Timings for the Readers Writers program

Explanation:

- case 1: Reader process and Writer process are not distinct i.e., there is just one process handling both functions. All 40 data items are handled by the same process.
- case 2: Number of Reader processes: 1. Number of Writer processes: 1. Each process handles 40 data item each.
- case 3: Number of Reader processes: 4. Number of Writer processes: 4. Each process handles 10 data items each.
- case 4: Number of Reader processes: 5. Number of Writer processes: 5. Each process handles 8 data items each.
- case 5: Number of Reader processes: 10. Number of Writer processes: 10. Each process handles 4 data items each.
- case 6: Number of Reader processes: 40. Number of Writer processes: 40. Each process handles 1 data item each.

interpreter and then on the uniprocessor version. The semaphore facility provided by Smalltalk is heavily used in these tests because semaphores are used to control access to data. The fork primitive is also heavily exercised because both readers and writers are processes forked by a single parent process. A summarized tabulation of the results is presented in Table 1. [5]

5.3 The Prime Number problem

As described earlier, this problem consists of finding the number of primes between two given numbers. In a classical sequential implementation, we would start from the lower number and check each number in turn, incrementing a counter if we found the number to be prime. In our parallel implementation, we divide the interval into 'bites'. Each process bites off a portion of the interval and works on it. It keeps a local tally of the number of primes which are summed up in the end.

[5] In this table and in other tables occurring in this report, timings are only approximate and correct to the second. Timings were measured by invoking a Smalltalk primitive which in turn invoked a system call to return the current time. Also, system load may have varied slightly between one set of tests and another, but the variation is unimportant because it is the relative performance that interests us, not the exact execution time.

number of processes	sequential implementation (min:sec)	concurrent implementation (min:sec)
1	2:46	4:28
2	2:45	2:17
5	2:47	1:01
8	2:49	0:59

Table 2: Timings for the prime numbers program

After each process 'chews through' the sub-interval that it has 'bitten off' it tries to obtain more work to do. The implementation needs semaphores to regulate access to the global count of primes and to the memory locations that hold the interval bounds. Note that it is better to have a dynamic distribution of work load using 'bites' than a static distribution because the dynamic distribution enables processes working on smaller numbers (and hence requiring less computation time) to begin work on another sub-interval after finishing with one sub-interval. This method is based on the material in [BecOli 87].

A summarized tabulation of the execution time is given in Table 2. In each case, it is required to find the number of primes between 1 and 1000. The number of processes acting concurrently was varied and execution time measured.

5.4 Analysis

From tables 1 and 2 it is very obvious that the multiprocessor implementation is faster than the uniprocessor implementation in most cases. It is interesting to consider cases in which this is not true. The multiprocessor implementation is slower than the uniprocessor implementation in the case where there is only one executing process. This can be explained by observing that the multiprocessor implementation incurs a substantial overhead in setting up shared memory, a feature that never gets used because there is only one process. If there were more than process, the initial (high) cost can be amortized over more than one process so the overhead per process decreases. The high cost and slower performance can also be attributed to the cost of locking each data structure that gets used. Again, overhead incurred in locking is unnecessary when only one process is being used. Thus, the multiprocessor implementation would almost certainly be slower on a uniprocessor machine. On the other hand, the results in tables 1 and 2 demonstrate very clearly that very significant performance gains can be achieved when more than one processor is available for use. This observation is intuitively correct and needs no explanation.

It can be noticed from table 2 that the execution time does not vary linearly with the number of processes. In particular, it can be seen that the curve flattens and the execution time does not decrease after a certain point. This behaviour is not a limitation of the the algorithm or the implementation. The behavior can be explained by considering that at a given instant, only a certain (small) number of processors are available for use. Increasing the number of forked processes to a number greater than the number of available processors cannot give any additional performance benefit. Such a situation may even lead to a deterioration in performance as the operating system tries to juggle memory space and processor time between a larger number of processes competing for resources. A similar observation applies to the timings presented for the sequential (uniprocessor) version of the interpreter. It may seem counter-intuitive that the execution time varies for different

number of processes in a uniprocessor implementation. In the uniprocessor implementation, multiprocessing is simulated. The Smalltalk system does the necessary house keeping in executing each process in a round-robin fashion. This should cause a small change in the overhead attributable to the system and probably explains the small (less than 2%) variation in execution time.

In conclusion, it can be stated that the results from the tests conducted support the thesis that implementation on a multiprocessor would improve the performance of the Smalltalk interpreter. It is seen that in all but one case (the case in which there is only one active process) the overhead introduced by the new synchronization, shared memory and locking mechanisms does not overwhelm the performance benefits of the parallelism enabled by these mechanisms. Comments and suggestions for further improvements in performance are deferred for the next section.

6 Comments and Conclusion

In this section, we make a few comments about the difficulties encountered during the implementation and enumerate the factors that contributed to the difficulty. We make suggestions about the kind of tools that would make an implementation such as this easier. We then go on to make suggestions about improving the performance of the implementation and suggestions about making Smalltalk easier to use as a concurrent programming language.

6.1 Problems along the way

As has been mentioned before, a major part of the interpreter is implemented in 'C'. 'C' is certainly a very good language for writing compilers and interpreters. The problem with this implementation was that it was on a multiprocessor and there are no 'C' constructs that aid concurrent programming. The Ada 'task' construct is one such multiprogramming aid that would have helped if it had an equivalent in C [MaeOldOld 87]. In the absence of such constructs, concurrent programming can only be achieved by using the system call **fork()**. This in itself suffices for the implementation of multiprogramming but is hardly an economical resource to use. It is also an inelegant way of implementing concurrent programming. Aesthetics apart, **fork()** is far from being an ideal tool even from the practical implementational viewpoint. Most debuggers on UNIX systems cannot effectively trace child processes forked from a parent. Thus, the absence of an appropriate debugger was another handicap in implementing the interpreter on the multiprocessor.

Another handicap in the debugging process is the absence of a debugger for the Little Smalltalk system. A newer version of the interpreter displays a backtrace of the messages invoked whenever there is an error. This is a big improvement over the earlier version that was used as the base system for the multiprocessor implementation. The older version did not, on error, print out the backtrace of messages invoked and this made debugging difficult when the error was in the Smalltalk code. The backtrace support in the newer version notwithstanding, debugger support for Little Smalltalk is necessary if it is to be used in really serious programming.

As can be expected in a multiprocessor environment, it was necessary to use locks extensively to maintain data integrity against concurrent conflicting updates. The parallel programming library on the Sequent offers two types of locks. Each lock can be used to control access to a particular location in memory. While using these locks in easy enough as a programming exercise, it is a non-trivial problem to ensure that deadlocks do not result from their use. It is not very difficult to ensure deadlock-free code if there are a small number of locks. But using smaller number of locks leads to a larger granularity of locking which in turn prevents maximal concurrency. Using a very small granularity for locking ensures maximum concurrency but leads to problems with deadlock avoidance.

It is our experience that using conventional methods of deadlock detection such as resource graphs and wait-for graphs [MaeOldOld 87] is impractical when a large number of resources are involved. While we are unable to think of a system where a trade-off between granularity and concurrency would not exist, we nevertheless hope to see in the near future, research dedicated to the problems of detecting deadlocks in code written in conventional (procedural) languages. It does seem to us that an extended form of data flow analysis would be able to detect such deadlocks.

6.2 Suggestions for improvements

There is certainly more than one area in which this project can be extended to obtain a better Smalltalk interpreter and a better parallel programming environment. The suggestions for improvement are presented in two groups — those that relate to the improvement of the preformance of the Smalltalk interpreter and those that relate to the development of a better parallel programming environment.

6.2.1 Performance improvements

The data presented in an earlier chapter showed that the multiprocessor implementation is significantly faster that the previous uniprocessor implementation. But in keeping in mind an anonymous but very appropriate aphorism [6] we are interested in extracting that extra ounce of performance from the system. A study of the architecture of the system and the data from a performance profiler indicate that there are two areas of improvement possible. They are discussed next.

The performance improvement obtained in the multiprocessor implementation is almost completely because of the fact that multiple processes run concurrently on multiple processors and compute what would otherwise be computed by just one processor in a much longer time by a conventional uniprocessor implementation. But the creation of the multiple processes involves operations that impact performance. The **fork()** system call of the UNIX operating system is an expensive operation to invoke [Sequent87]. Because of the large amount of memory that needs to be duplicated for the new child process and because of the other house keeping operations that the **fork()** call has to perform, it takes a long time to execute, time that can be measured in the order of milliseconds. If **fork()** can be made faster or if an alternative can be found to for it, performance would significantly improve. **Fork()** is not a problem if the number of processes forked is small and if each of the forked processes executes for long periods of time (of the order of seconds). In situations where a very large number of processes are forked off to do small pieces of computation, the impact on performance will be felt.

The solutions for this problem interest us only if they do not require modifications to the operating system itself as such, for such modifications would make the interpreter very non-portable. We are unable to think of any way of getting around this problem on a standard UNIX system. On the Dynix system however, the parallel programming library provides facilities to 'reuse' processes. Thus instead of terminating a child process after it is done, it is 'parked' and reused when a new process is needed [Sequent87]. Such reuse is supported to reduce the cost of a fork and is particularly appropriate for programs that are written with *function partitioning* [7] in mind.

Another non-standard way of reducing the cost of forks is to use the Light Weight Process (LWP) facility available on some operating systems. LWPs do not have the overheads that a normal process

[6] The need for computing power always overwhelms the available computing resources.

[7] *function partitioning* is also called heterogeneous multitasking and is appropriate where multiple processes perform different operations on the same (shared) data set. In contrast, *data partitioning*, also called homogeneous multitasking, is used where the same operations are to be carried out concurrently on large data sets. [Sequent87]

would have and are very inexpensive to create and use. In the Little Smalltalk interpreter, almost all of the memory is shared between all the processes and such a situation is nearly ideal for the usage of LWPs. But, as indicated, LWPs are not available on standard UNIX systems. There has been a trend however of the newer versions of UNIX supporting LWPs. Sun OS 4.0 (from Sun Microsystems) already supports LWPs and some multiprocessor machine vendors also offer it. One can be fairly certain that Dynix, the operating system under which the interpreter is currently implemented, will also support LWPs sometime in the near future.

The semaphore mechanism and the locking mechanism offered by the Dynix parallel programming library have been used in the implementation of the interpreter. It has been our experience that the locking operations and the semaphore operation as provided by the system are slow. The semaphore library for example, was tested against a 'home grown' library of equivalent functionality. The system provided library was found to be much slower than the 'home grown' library. We expect a similar behavior for the locking mechanism. Thus, replacing the system libraries with hand-coded efficient libraries could make a significant improvement in performance.

6.2.2 Suggested changes to the language

This project was undertaken with the intention of parallelizing the Little Smalltalk interpreter. We made a conscious decision not to change the syntax of the language and we scrupulously refrained from succumbing to the temptation of making changes to the syntax or adding extensions to the language. We consider this to be a distinguishing feature of this effort over a host of other efforts at adding concurrency to Smalltalk. These other efforts are documented in chapter 2 of this report. Having succeeded in our intention of implementing such an interpreter, we now feel free to examine Smalltalk from a new perspective, that of critiquing its shortcomings.

We certainly believe that Smalltalk is an interesting language and its ideological purity is an example to follow for language designers. But examined with regards to its suitability as a language for multiprocessor programming, we did discover a few shortcomings. One of the most fundamental design decisions of Smalltalk, that of not making processes into ordinary (first class) objects, hinders expression of the problem in a natural way. Processes cannot be specialized into subclasses and they can be sent only a very small number of specific messages [ThomDav88]. In a multiprocessor implementation, the necessity of being able to freely communicate between processes makes it imperative that we allow some kind of a message passing between processes. The only way we can allow message passing between processes is by having them as objects, with the same properties as every other object in the system.

It is necessary to explain further why we think it necessary to have a mechanism for passing messages between processes. Consider the example of a program designed to find the factorial of a number. A parallel implementation of this would calculate (for instance) the product of the first half numbers in one process and the product of the last half numbers in a second process and then multiply the two products together to get the desired factorial value. To multiply the two intermediate results together, the third process has to know when the first two processes are done with computing their respective products. In a standard Smalltalk system, such synchronization can be achieved only by using a semaphore for each process so that each process can set a semaphore when it is done. The third process should do a wait on these semaphores and proceed after they are set. It would be conceptually simpler (and ideologically cleaner) to have the first two processes send a message (containing the products they computed) to the third process. We do not expect such a scheme to be more efficient, but we do believe that it would simplify concurrent programming in Smalltalk.

6.3 Conclusion

The interpreter for Little Smalltalk has been successfully implemented on the multiprocessor Sequent Balance. The benchmark programs run on the interpreter indicate significant speed-up for almost all cases. Our thesis that the interpreter could be implemented on a multiprocessor and that it would lead to performance benefits is thus proved correct. We also hope to have established the possibility of using Smalltalk in particular and object oriented languages in general as concurrent languages. We have concluded with a list of suggestions for further research in this area.

7 Acknowledgements

Dr. Walter Rudd and Dr. Toshimi Minoura made many useful comments on this report. We thank them for their suggestions.

References

[Backus78] Backus, John "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", in *Communications of the ACM*,**21**(8):613–641, August 1978

[BecOli87] Beck, Bob and Olien, Dave "A Parallel Programming Process Model", in *USENIX Association, Conference Proceedings*, Winter 1987, 83–102

[BecOli89] Beck, Bob and Olien, Dave "A Parallel Programming Process Model", in *IEEE Software*, 63–72, May 1989

[Budd87] Budd, Timothy "A Little Smalltalk", Addison–Wesley Publishing Company, 1987

[ChanMisra88] Chandy, K. Mani and Misra, Jayadev "Parallel Program Design – A Foundation", Addison–Wesley Publishing Company, 1988

[Dally88] Dally, William J. and Chien, Andrew A. "Object-Oriented Concurrent Programming" in *Sigplan Notices*,**24**(4):28–31, April 1989

[Deitel84] Deitel, Harvey M. "An Introduction to Operating Systems", Addison–Wesley Publishing Company, 1984

[Gentle84] Gentleman W.M., "Message Passing Between Sequential Processes: The Administrator and the Reply Concept" in *Software Practice and Experience*,**11**(5), May 1981

[Ingalls81] Ingalls, Daniel H. H "Design Principles behind Smalltalk", in *Byte*, 286–298, August 1981

[Johnson88] Johnson, Ralph E., Graver, Justin O. and Zurawski, Lawrence W. "TS: An Optimizing Compiler for Smalltalk", in *Conference Proceedings of OOPSLA '88, Special issue of Sigplan Notices*, **23**(11):18–26, November 1988

[Kay77] Kay, Alan C. "Microelectronics and the Personal Computer", in *Scientific American*, **237**(3):230–244, September 1977

[Krasner81] Krasner, Glenn "The Smalltalk-80 Virtual Machine", in *Byte*, 300–320, August 1981

[MaeOldOld87] Maekawa, Mamoru; Oldehoeft Arthur E., and Oldehoeft Rodney R. "Operating Systems — Advanced Concepts", The Benjamin/Cummings Publishing Company, Inc. 1987

[Micallef88] Micallef, Josephine "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", in *Journal of Object-Oriented Programming*, 12–36, April/May 1988

[Nakajima88] Nakajima Tatsuo *et al.*, "Distributed Concurrent Smalltalk: A Language and System for the Interpersonal Environment" in *Sigplan Notices*,**24**(4):43–49, April 1989

[Sequent85] "Balance 8000 Technical Summary", Technical Reference Manual, Sequent Computer Systems, Inc., 1985

[Sequent87] Osterhaug, Anita "Guide to Parallel Programming", Sequent Computer Systems, Inc., 1987

[ThomDav88] Thomas, David A. *et al.*, "Actra - A Multitasking/Multiprocessing Smalltalk", in *Sigplan Notices*,**24**(4):87–90, April 1989

[Xerox81] The Xerox Learning Group "The Smalltalk-80 System", in *Byte*, 36–48, August 1981

[YokoToko87] Yokote, Yasuhiko and Tokoro, Mario "Concurrent Programming in Concurrent-Smalltalk", in *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro, eds 129–158, The MIT Press, 1987

[YoneToko87] Yonezawa, Akinori and Tokoro, Mario "Object-Oriented Concurrent Programming: An Introduction", in *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro, eds 1–7, The MIT Press, 1987