VLSI Memo No. 89-530
May 1989

Experience with CST: Programming and Implementation

Waldemar Horwat, Andrew A. Chien and William J. Dally

Abstract

CST is a programming language based on Smalltalk-80 that supports concurrency using locks, asynchronous messages, and distributed objects. In this paper, we describe CST: the language and its implementation. Example programs and initial programming experience with CST is described. An implementation of CST generates native code for the J-machine, a fine-grained concurrent computer. Some novel compiler optimizations developed in conjunction with that implementation are also described.

Microsystems Research Center
Room 39-321
Massachusetts Institute of Technology
Cambridge Massachusetts 02139
Telephone (617) 253-8138

Acknowledgements

To appear in *Proceedings, SIGPLAN Conference*, June 21 - 23, 1989, Portland, Oregon. This research was supported in part by the Defense Advanced Research Projects Agency under the Office of Naval Research contracts N00014-88-K-0738, N00014-87-K-0825, and N00014-85-K-0124, by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, by an Analog Devices Fellowship, and by an ONR Fellowship.

Author Information

Horwat: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-416, MIT, Cambridge, MA 02139. (617) 253-6048.
Chien: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-415, MIT, Cambridge, MA 02139. (617) 253-6048.
Dally: Artificial Intelligence Laboratory and the Laboratory for Computer Science, Room NE43-417, MIT, Cambridge, MA 02139. (617) 253-6043.

Experience with CST: Programming and Implementation [1]

Waldemar Horwat, Andrew A. Chien and William J. Dally
Artificial Intelligence Laboratory and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

CST is a programming language based on Smalltalk-80 that supports concurrency using locks, asynchronous messages, and distributed objects. In this paper, we describe CST: the language and its implementation. Example programs and initial programming experience with CST is described. An implementation of CST generates native code for the J-machine, a fine-grained concurrent computer. Some novel compiler optimizations developed in conjunction with that implementation are also described.

Introduction

This paper describes CST, an object-oriented concurrent programming language based on Smalltalk-80 [7] and an implementation of that language. CST adds three extensions to sequential Smalltalk. First, messages are asynchronous. Several messages can be sent concurrently without waiting for a reply. Second, several methods may access an object concurrently; locks are provided for concurrency control. Finally, CST allows the programmer to describe distributed objects: objects with a single name but distributed state. They can be used to construct abstractions for concurrency.

CST is being developed as part of the J-Machine project at MIT [4, 3]. The J-Machine is a fine-grain concurrent computer. The primary building block in the J-machine is the Message-Driven Processor (MDP). It efficiently executes tasks with a grain size of 10 instructions and supports a global virtual address space. This machine requires a programming system that allows programmers to concisely describe programs with method-level concurrency and that facilitates the development of abstractions for concurrency.

Object-oriented programming meets the first of these goals by introducing a discipline into message passing. Each expression implies a message send. Each message invokes a new process. Each receive is implicit. The global address space of object identifiers eliminates the need to refer to node numbers and process IDs. The programmer does not have to insert send and receive statements into the program, keep track of process IDs, and perform bookkeeping to determine which objects are local and which are remote.

For example, a CST program[2] that counts the number of leaves in a binary tree using double recursion is shown in Figure 1. Nowhere in the program does the programmer explicitly specify a send or receive, and no node numbers or process IDs are mentioned. Yet, as shown in Figure 1[3] the program exhibits a great deal of concurrency. Making message-passing implicit in the language simplifies programming and makes it easier to describe fine-grain concurrency.

CST facilitates the construction of concurrency abstractions by providing distributed objects: objects with a single name whose state is distributed across the nodes of a concurrent computer. The one-to-many naming of distributed objects along with their ability to process many messages simultaneously allows them to efficiently connect together

[1] The research described in this paper was supported in part by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under contracts N00014-88K-0738, N00014-87K-0825, and N00014-85-K-0124, in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, an Analog Devices Fellowship, and an ONR Fellowship.

[2] This program is in *prefix* CST, a dialect that has a syntax resembling LISP. *Infix* CST [5] has a syntax closer to that of Smalltalk-80.

[3] The concurrency profiles presented in this paper are produced by an Icode level simulation of CST programs.

```
(class node (object) left right tree-node?)

(method node count-elements () ()
       (if tree-node? (+ (count-elements left)
                         (count-elements right))
           1))
```

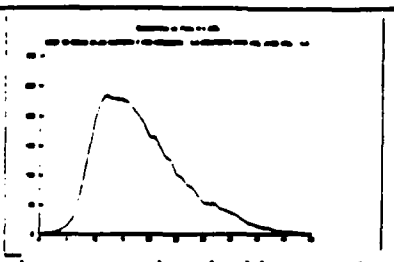

Figure 1: A CST program that calculates the number of leaves in a tree using double recursion. Its concurency profile (active tasks in each message interval) is shown to the right.

large numbers of objects. Distributing the name of a single distributed queue to sets of producer and consumer objects, for example, connects many producers to many consumers without a bottleneck.

The Optimist compiler [8] compiles Concurrent Smalltalk to the assembly language of the Message-Driven Processor (MDP) [9]. It includes many standard optimizations such as register variable assignment, dataflow analysis, copy propagation, and dead code elimination [2, 13] that are used in compilers for conventional processors. Due to the fine-grained parallel nature of the J-machine, compiling for the MDP is unlike compiling for most conventional processors in a few important aspects. For instance, loops are not important[4], while minimizing code size, tail forwarding methods, and efficiently and seamlessly handling parallelism are extremely important.

The development of Concurrent Smalltalk was motivated by dissatisfaction with process-based concurrent programming using sends and receives [11]. Many of the ideas have been borrowed from actor languages [1]. Another language named Concurrent Smalltalk has been developed at Keio University in Japan [14]. This language also allows message sending to be asynchronous, but does not include the ability to describe distributed objects.

Concurrent Smalltalk

Top-Level Expressions

A CST program consists of a number of top-level expressions. Top level forms include declarations of program and data as well as executable expressions. Linking of programs (the resolution from selectors to methods) is done dynamically.

```
<top-exp> :=    (Global <global-variable> <value>) |
                (Constant <constant-name> <value>) |
                (Class <class-name> (<superclas  :instance-vars>) |
                (Method <class-name> <method-na >
                        (<formals>) (<locals>)
                        <expressions>) |
                <expression>
```

Globals and Constants Globals and constant declarations define names in the environment. These names are visible in all programs, unless shadowed by a instance, argument, or local variable name. The global declaration simply defines the name. Its value remains unbound. The constant declaration defines the name and binds the name to the specified value.

Classes Objects are defined by specifying classes. Objects of a particular class have the same instance variables and understand the same messages. A class may inherit variables and methods from one or more superclasses. For

[4] In fact, the current version of Concurrent Smalltalk does not even have loops.

example:

```
(class node (object) left right tree-node?)
```

defines a class, **node**, that inherits the properties of class **object** and adds three instance variables. This means that methods for the class **node** can access all the instance variables of class **object** as well as those defined in their own class definition. Methods defined for class **object** are also inherited. Of course, this inheritance is transitive, so **node** actually inherits from a series of classes up through the top of the class hierarchy. Instance variables in the class definition may hide (shadow) those defined in the superclasses if they have the same name. The same kind of shadowing is allowed for selectors (method names).

Methods The behavior of a class of objects is defined in terms of the messages they understand. For each message, a method is executed. That execution may send additional messages, modify the object state, modify the object behavior, and create new objects. Methods consist of a header and a body. The header specifies class, selector, arguments, and locals. The body consists of one or more expressions. For example:

```
(method node count-elements () ()
       (if tree-node? (+ (count-elements left)
                         (count-elements right))
           1))
```

defines a method for class **node** with selector **count-elements**. The two empty lists indicate that there are no explicit arguments and no local variables. If present, the keyword **reply** sends the result of the following expression back to the sender of the **count-elements** message. In this case, there is no **reply** keyword, so the method replies with the value of the last expression. If the programmer wishes to suppress the reply, he can use the **(exit)** form which causes the method to terminate without a reply.

Messages are sent implicitly. Every expression conceptually involves sending a message to an object. Of course, commonly occurring special cases, like adding two local integers, will be optimized to eliminate the send. For example, **(count-elements left)**, sends the message **count-elements** to **left**. **(+ x y)** sends the message **+** with argument **x** to object **y**. If both **x** and **y** are local integers, this operation can be optimized as an add instruction.

Each expression consists of a selector, a receiver, and zero or more arguments. Identifiers must be one of: constant, global variable, argument, local variable, or instance variable. Subexpressions may be executed concurrently and are sequenced only by data dependence. For example, in the following expression from the program in Figure 1

```
(+ (count-elements left) (count-elements right))
```

the two **count-elements** messages will be sent concurrently and the **+** message will be sent when both replies have been received. The only way to serialize subexpression evaluation is to assign intermediate results to local variables.

A complete list of CST expressions is shown below:

```
<exps> := <exp>+
<exp> :=
    <name> |
    (<selector> <receiver-exp> <argument-exp>*) |
    (send <selector-exp> <receiver-exp> <argument-exp>*) |
    (value <exp>) |
    (set <name> <exp>) |
    (cset <name> <exp>) |
    (msg <node> <selector> <receiver> <actuals>) |
    (forward <continuation> <selector> <receiver> <args>) |
    (reply <exp>) |
```

```
(block (<formals>) (<locals>) <expe>) |
(if <exp> <exp> <exp>) |
(begin <expe>) |
(exit)
```

An Example CST Program

We now introduce a slightly more complicated version of the program shown in Figure 1. Rather than simply counting the leaves on a tree, we compute the lengths of all the lists linked to the tree and sums those lengths together.

```
(class node (object) left right)

(method node count-list-elements () ()
       (+ (count-list-elements left)
          (count-list-elements right)))

(class pair (object) car cdr)

(method pair count-list-elements () ()
       (length right 0 )))

(method pair length (n) ()
       (if (eq cdr 'nil) (+ 1 n)
           (length cdr (+ 1 n))))
```

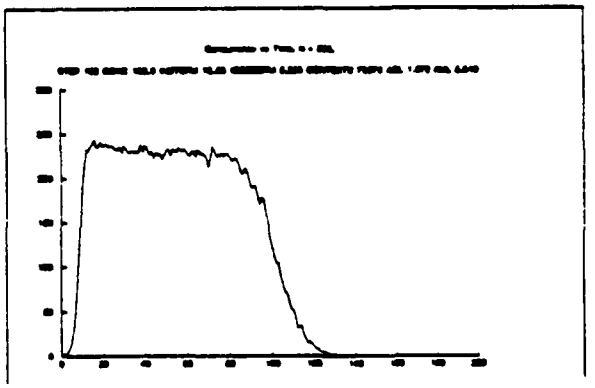

Figure 2: A CST program that computes sum of list lengths and its execution profile

The node class definition is the same as it was in Figure 1. `left` and `right` are the children of the current node in a binary tree. The right of each leaf node points to a linked list of pairs. The method `count-list-elements` recursively counts the lists lengths by doing so for the right subtree and the left subtree concurrently. At the bottom of the tree, the late binding SEND operation causes the `count-list-elements` method for pairs to be invoked. This method computes the length of each list using the tail recursive method `length`.

Distributed Objects

CST programs exhibit parallelism between objects, that is many objects may be actively processing messages simultaneously. However, ordinary objects can only receive one message at a time. CST relaxes this restriction with Distributed Objects (DOs). Distributed objects are made up of multiple representatives (constituent objects) that can each accept messages independently. The distributed object has a name (Distributed object ID or DID) and all other objects send messages to this name when they wish to use the DO.

Messages sent to the DO are received by one and only one constituent object (CO). Which constituent receives the message is left unspecified in the language. A clever implementation might send the messages to the closest constituent whereas a simpler implementation might send the messages to a random constituent. The state of a distributed object is typically distributed over the constituents. This means that responding to an external request often requires the passing of messages amongst the constituents before replying. No locking is performed on the distributed object as a whole. This means that the programmer must ensure the consistency of the distributed object.

Support for Distributed Objects

CST includes two constructs to support distributed objects. For DO creation, we add an argument for the new selector – the number of constituents desired in this DO. In order to pass messages within the object, each constituent object must be able to address each of the other constituents. This is implemented with the special selector co. Each distributed object can use this selector, the special instance variable group (a reference to the DO), and an index to address any constituent. For example, (co group 5) refers to the 5th constituent of a distributed object. Each constituent also has access to its own index and the number of constituents in the entire distributed object. Thus a description of a distributed object might look something like the example shown in Figure 3.

```
;;   Distributed Array Abstraction.
;;   The constituents are spread throughout the machine.
;;   The array state is allocated into equal sized chunks on the constituents.

(class distarray (distobj) nr-elts chunk-size elt-array)

;;
;;  Given an uninitialized DO, init makes each one an array,
;;    tells it how many elts it has, and how many elements
;;    are in the entire array.

(method distarray init (arr-size) ()
  (do-1 self (block (constit elts) ()
                    (co-init (co (group constit) (myindex constit)) elts)
                    (reply constit))
        arr-size))
;
;; helper for init
;;
(method distarray co-init (elts) ()
        (begin (set chunk-size (/ elts (+ 1 maxindex)))
               (set nr-elts elts)
               (set elt-array (new array chunk-size))
               ))
;;
;;  Tree recursive apply, with one argument
;;
(method distarray do-1 (sblock arg1) () (ldo-1 (co group 0) sblock arg1))
(method distarray ldo-1 (sblock arg1) (a b lindex rindex)
        (set lindex (lindex self))
        (set rindex (rindex self))
        (cset a (if (<= lindex maxIndex) (ldo-1 (co group lindex) sblock arg1)
                    '()))
        (cset b (if (<= rindex maxIndex) (ldo-1 (co group rindex) sblock arg1)
                    '()))
        (touch a b)
        (reply (value sblock self arg1))
        (exit))
;;
;;  Select array element at index
;;
(method distarray at (index) (selector)
        (if (or (< index (* chunk-size myindex))
                (>= index (* chunk-size (+ myindex 1))))
            (begin (set selector (truncate (/ index chunk-size)))
                   (forward requestor at (co group selector) index)
                   (exit))
            (at elt-array (mod index chunk-size))))
;;
;;  Set array element at index to value
;;
(method distarray at.put (index value) (selector)
        (if (or (< index (* chunk-size myindex))
                (>= index (* chunk-size (+ myindex 1))))
            (begin (set selector (truncate (/ index chunk-size)))
                   (forward requestor at.put (co group selector) index value)
                   (exit))
            (at.put elt-array (mod index chunk-size) value)))
;;
;; to make a distarray of 256 constituents and 1024 elements do
;;
;;  (init (new distarray 256) 1024)
;;
```

Figure 3: A Distributed Array Example

In the example of the distributed array, we would create a usable array with two steps. First we construct the distributed object using the new form. The example in Figure 3 creates a distributed object with 256 constituents. After the DO is created, we must initialize in a way that is appropriate for the distributed array. We do so by sending it an `init` message (also defined in Figure 3). This initialization sets each constituent up with an private array of the

appropriate number of elements. For example, if we wanted a distarray of 512 elements, in this case each constituent would have a private array of two elements. This initialization is done in a tree recursive fashion and therefore takes $O(lg(n))$ time.

The mapping of the distarray elements onto the private arrays is done by the `at` and `at.put` methods. Each constituent is responsible for a contiguous range of the distarray elements. Any requests received by a constituent are first checked to see if they are within the local CO's jurisdiction. If they are not, they are forwarded to the appropriate CO. If they are, the request is handled locally. This is a particularly simple example because each constituent is wholly responsible for his subrange and need not negotiate with other constituents before modifying his local state.

Distributed objects are of great utility in building large objects on a fine grain machines. In the J-machine, we restrict ordinary objects to fit within the memory of single node, thus restricting object size. With distributed objects, we only require that a constituent of the DO fit on a single node. Some useful examples for distributed objects are dictionaries, distributed arrays, sets, queues, and priority queues.

Experience with CST

We have written a large number of Concurrent Smalltalk programs and executed them on our Icode simulator. These programs include various data structures, distributed arrays, sets, rings, B-trees, grids, and matrices. They also include several application kernels: N-body interaction and charged particle transport (Particle-in-cell algorithm). To date, the programs studied range from toys to applications of over 1000 lines. It is clear from our experience that CST programs exhibit large amounts or parallelism. However, we are just beginning to exploit the potential of Distributed Objects as building blocks for concurrent programs. We will continue to study data structures, algorithms and full-blown applications in our continuing evaluation of Concurrent Smalltalk.

The Optimist Compiler for CST

Goals

The main goal of the Optimist compiler is to produce Concurrent Smalltalk code that is as small as possible without sacrificing speed. In almost all cases optimizations that reduce space also reduce speed, but there are a few cases in which they conflict; in those cases the decisions were made in favor of optimizing space. Compilation speed was not a major goal of the compiler project; simplicity and flexibility were considered more important. Still, the compiler does achieve reasonable compilation speed, taking between one and fifteen seconds to compile most methods on a 2-megabyte Macintosh[5] II using Coral Software's Allegro Common Lisp.

Organization

The Optimist compiler is comprised of four phases, as shown in Figure 4. The Concurrent Smalltalk Front End can be replaced by other front ends to compile other languages for the MDP. Also, the Icode can be extracted from two places in the compilation process and either compiled onto different hardware or run on an Icode simulator.

The source code is converted by the Front End into an intermediate language called Icode. The Icode is at a somewhat higher level than the triples or quadruples codes that most compilers use, in that it specifies units such as entire procedure calls in single instructions. The Icode also allows for the possibility of having more than one source language compile into MDP assembly language code or having the same source language compile into several assembly languages. Figure 5 shows the `length` method in its Icode form.

[5] Macintosh is a trademark of Apple Computer, Inc.

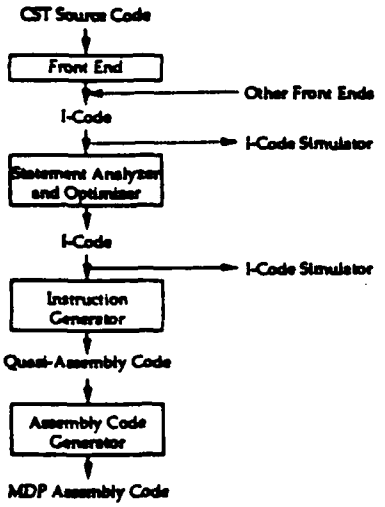

Figure 4: Compiler Organization.

```
(CSEND (TEMP 0) (METHOD EQ) (IVAR 1) (CONST NIL))
(FALSEJUMP (TEMP 0) 0)
(CSEND (TEMP 1) (METHOD +) (CONST 1) (ARG 0))
(JUMP 1)
(LABEL 0)
(CSEND (TEMP 2) (METHOD +) (CONST 1) (ARG 0))
(CSEND (TEMP 1) (METHOD LENGTH) (IVAR 1) (TEMP 2))
(LABEL 1)
(RETURN (TEMP 1))
```

Figure 5: Icode for the `length` Method: The Icode output by the Front End is a literal translation of the source code with few optimizations. At this point all method calls, including primitives, are compiled as CSENDs.

The Statement Analyzer and Optimizer processes and optimizes the Icode generated by the Front End. It performs all of the compiler's optimizations that are relevant at the Icode level of abstraction. Internally it works with Icode in the form of a directed control-flow graph. These optimizations include dead code elimination, move elimination, dataflow transformations, constant folding, tail forwarding, and merging of identical statements on both sides of paths of a conditional. The optimizations are repeatedly attempted until none of them can improve the code.

The Instruction Generator compiles each Icode statement to a number of quasi-MDP instructions and outputs the MDP code in the form of a directed control-flow graph. At the same time, the Instruction Generator assigns variables to either registers or memory locations and performs statement-specific optimizations on Icodes.

The Assembly Code Generator inserts branches into the directed graph of quasi-MDP instructions created by the Instruction Generator and performs several peep-hole optimizations. The important optimizations include shifting instructions wherever possible to align DC (Load Constant) instructions to word boundaries (all other instructions need only be aligned at half-word boundaries) and combining SEND and SENDE instructions to SEND2 and SEND2E. The Assembly Code Generator replaces short branches by long ones where necessary; such replacements are complicated by the fact that long branches alter the value of MDP's register R0. The Assembly Code Generator outputs a file of assembly language statements which can be read, assembled, and executed by our MDP simulator

MDPSim [10]. Figure 6 contains the assembly code output for the sample method **length**.

```
MODULE PAIR__LENGTH
        DC      MSG:LoadCode+18
        DC      {Class_PAIR},{Method_LENGTH}}
        MOVE    [2,A3],R0               ;  0
        XLATE   R0,A2,XLATE_OBJ         ;  0.5
        MOVE    1,R3                    ;  1
        ADD     R3,[3,A3],R2            ;  1.5
        MOVE    [3,A2],R1               ;  2
        BNIL    R1,^L001                ;  2.5
        MOVE    [4,A3],R1               ;  3
        BNIL    R1,^L002                ;  3.5
        DC      MSG:ReplyConst+4        ;  4
        WTAG    R1,1,R3                 ;  5
        LSH     R3,-16,R3               ;  5.5
        SEND2   R3,R0                   ;  6
        SEND    R1                      ;  6.5
        SEND2E  [5,A3],R2               ;  7
        BR      ^L002                   ;  7.5
L001:   MOVE    [3,A2],R0               ;  8
        CALL    Send_Node_Nr            ;  8.5
        DC      MSG:SendConst+7         ;  9
        SEND2   R1,R0                   ; 10
        DC      {Method_LENGTH}         ; 11
        SEND    R0                      ; 12
        SEND2   [3,A2],R2               ; 12.5
        SEND    [4,A3]                  ; 13
        SENDE   [5,A3]                  ; 13.5
L002:   SUSPEND                         ; 14
        END
```

Figure 6: Final Output of the Compiler: This is the MDP assembly code into which the **length** method compiles. If the optimizations were turned off, the code size would have been 32 words, more than twice the size of the optimized code.

Optimizations

Tail Forwarder The tail forwarder performs the message-passing equivalent of tail recursion. It is often the case that the value returned by a Concurrent Smalltalk method is the value returned by the last statement of that method, and that statement is often a method call. An example of this phenomenon is a recursive definition of the **length** function in Figure 2.

If **cdr** is not equal to **nil**, the **length** method makes a recursive call and when that call returns, it immediately returns that value as the result. There is, however, no fundamental reason why **length** should wait for the result of the recursive call to **length** only to return it to the caller; on the contrary, it would be better if the recursive **length** call returned its result to the initial caller. **length** optimized this way runs in constant space instead space proportional to the list length. The Tail Forwarder performs this optimization by looking for a CSEND statement whose value is returned by a REPLY statement immediately afterwards. Such a CSEND statement is modified to inform the callee to return its result to this method's caller instead of this method.

Fork and Join Mergers These two optimizations, if they can be applied, often produce significant savings in the output code size. They try to consolidate similar statements on both sides of forks (conditionals) and joins (places

where two paths of control flow merge) in the control-flow graph.

The Join Merger looks for similar statements immediately preceding each join in the control-flow graph. Here two statements are considered to be similar if they are identical or if they are both CSENDs with identical targets and the same number of arguments; the arguments themselves need not be the same. The Join Merger moves both statements after the join; if the statements were not identical, MOVEs are generated to copy any differing arguments into temporaries before the join; the combined statement after the join will use the temporaries instead of the original arguments. These MOVEs are usually later removed by the Move Eliminator. Although more than two paths of control flow can join at the same place, the Join Merger only considers them pairwise; if more than two paths can be merged, initially two will be merged, with the other ones considered in a later pass. The Fork Merger operates analogously except that it also has to be sure not to affect the value of the condition determining which branch the program will take.

The Join Merger occasionally merges two completely different method calls which happen to have the same number of arguments, but which may even call different methods (the method selector is treated as an argument like any other), a rather unexpected optimization indeed. In each branch just before the join, the resulting object code copies the differing method arguments into the MDP's registers and stores the appropriate method selector in a register. After the join is common code that sent the message given the method selector and arguments in the registers. Since the code to send a message is long compared to the code to load values into registers, the optimization has a net savings of five words (ten instructions) of code without significantly affecting the running time.

Move Eliminator For each MOVE statement from a local variable to another local variable, the Move Eliminator attempts to merge the source and destination variables into one variable and then remove the MOVE statement. Such a merge can be done successfully if the two variables are never simultaneously live at any point in the code.

The Move Eliminator complements the copy-propagation algorithm in the Optimist. Although both try to optimize MOVE statements, each is able to handle cases that the other cannot. The copy propagation can handle constants, while Figure 7 shows an example of MOVE statements that can be eliminated by the Move Eliminator but not by copy propagation.

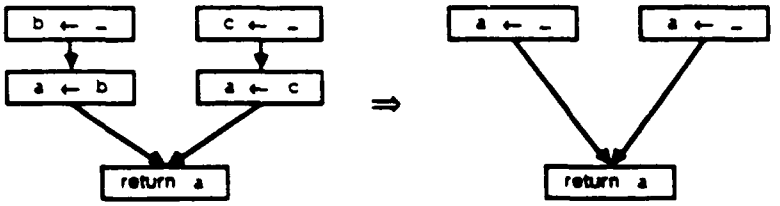

Figure 7: Move Eliminator Example: The Move Eliminator is able to remove the two MOVE statements (a←b) and (a←c) in the above code (the arrows indicate possible flow of control paths). The copy propagation algorithm would not detect the opportunity to remove these two MOVE statements because the value of a at the return statement is neither a copy of b nor a copy of c. The above code does occur in many methods.

Variable Allocator A greedy algorithm is used to assign eligible variables to registers. The shortest-lived variables with the most references are considered first. A graph coloring algorithm is used to assign the variables that did not fit in the registers to context slots; thus, fewer context slots are used, saving valuable memory space.

Summary

In this paper, we have presented a new language, Concurrent Smalltalk, that is designed for concurrency. Specific support for concurrency includes locks, distributed objects, and asynchonous message passing.

Distributed Objects represent a significant innovation in programming parallel machines. We refer to the constituents of a distributed object with a single name, but the implementation of the object is with many constituents. This different perspective allows easy use of distributed objects by outside programs while allowing the exploitation of internal concurrency.

We have described an implementation of a CST system. This programming environment includes a compiler, simulator, and statistics collection package. This set of tools allows us to experiment with new constructs and implementation techniques for the language. Although many of the optimizations used by the Optimist compiler are generally known, they have usually been applied to compilers for conventional processors. The issues involved in compiling for the MDP are quite different from compiling for conventional processors. After examining the compiler's output, it becomes apparent that the optimizations are essential to the successful use of Concurrent Smalltalk on the MDP. The compiler's optimizations reduce the amount of code output by anywhere between 20% and 60% (or even more in some cases) compared to output with all nonessential optimizations disabled. Such a reduction is very important on a processor with only 4096 words of primary memory.

There are many open issues relating to CST and similar programming systems. Key efficiency issues remain unresolved: how fine grain will the programs written in CST be and what is the run time overhead of CST programs? There are also concerns about the expressive power of languages like CST – how easy is it to write programs in CST and how useful are distributed objects?

References

[1] Agha, Gul A., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[2] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools* Addison-Wesley, Reading, MA, 1986.

[3] Dally, W. J., "Fine-Grain Message-Passing Concurrent Computers," *these proceedings.*

[4] Dally, William J. et.al., "Architecture of a Message-Driven Processor," *Proceedings of the 14th ACM/IEEE Symposium on Computer Architecture*, June 1987, pp. 189-196.

[5] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, Kluwer, Boston, MA, 1987.

[6] Halstead, Robert H., "Parallel Symbolic Computation," *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 35-43.

[7] Goldberg, Adele and Robson, David, *Smalltalk-80, The Language and its Implementation*, Addison Wesley, Reading MA, 1984.

[8] Horwat, Waldemar, *A Concurrent Smalltalk Compiler for the Message-Driven Processor*, MIT AI Technical Report 1080, October 1988.

[9] Horwat, Waldemar and Totty, Brian. *Message-Driven Processor Architecture, Version 10* MIT Concurrent VLSI Architecture Memo, March 1988.

[10] Horwat, Waldemar and Totty, Brian. *Message-Driven Processor Simulator, Version 5.0* MIT Concurrent VLSI Architecture Memo, December 1987.

[11] Su, Wen-King, Faucette, Reese, and Seitz, Charles L., *C Programmer's Guide to the Cosmic Cube*, Technical Report 5203:TR:85, Dept. of Computer Science, California Institute of Technology, September 1985.

[12] Totty, Brian, "An Operating System Kernel for the Jellybean Machine," *MIT Concurrent VLSI Architecture Memo*, 1987.

[13] Wulf, William M., Johnsson, Richard K., Weinstock, Charles B., Hobbs, Steven O., and Geschke, Charles, M. *The Design of an Optimizing Compiler*. American Elsevier, New York, 1975.

[14] Yokote, Yasuhiko and Tokoro, Mario, "Concurrent Programming in ConcurrentSmalltalk," *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro eds., MIT Press, Cambridge, MA, 1987, pp. 129-158.