LISP AND SYMBOLIC COMPUTATION: An International Journal, 7, 283–305 (1994)

TS/Scheme: Distributed Data Structures in Lisp

SURESH JAGANNATHAN suresh@research.nj.nec.com
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

Abstract. We describe a parallel object-oriented dialect of Scheme called TS/SCHEME that provides a simple and expressive interface for building asynchronous parallel programs. The main component in TS/SCHEME's coordination framework is an abstraction that serves the role of a distributed data structure. Distributed data structures are an extension of conventional data structures insofar as many tasks may simultaneously access and update their contents according to a well-defined serialization protocol. The semantics of these structures also specifies that consumers which attempt to access an as-of-yet undefined element are to block until a producer provides a value.

TS/SCHEME permits the construction of two basic kinds of distributed data structures, those accessed by content, and those accessed by name. These structures can be further specialized and composed to yield a number of other synchronization abstractions. Our intention is to provide an efficient medium for expressing concurrency and synchronization that is amenable to modular programming, and which can be used to succinctly and efficiently describe a variety of diverse concurrency paradigms useful for parallel symbolic computing.

Keywords: Symbolic Computation, Parallelism, Distributed Data Structures, Coordination, Object-Oriented Computing

1. Introduction

Many parallel dialects of Lisp express concurrency via a lightweight process constructor such as **future**[16], and model process communication via dataflow constraints that exist between a future instance and its consumers. These dataflow constraints may be enforced implicitly via strict operations which are passed a future as an argument (*e.g.,* as in MultiLisp[16] or Mul-T[24]), or explicitly via a primitive wait operation on futures (*e.g.,* **touch** in MultiLisp, or the **future-wait** procedure in QLisp[15]). In either case, task communication is tightly-coupled with task creation: tasks[1] can only initiate requests for the value of an object if the identity of the object is known. Because of this constraint, multiple tasks cannot collectively contribute to the construction of a shared data object without using other explicit (low-level) synchronization primitives such as locks or monitor-like abstractions (*e.g.,* **qlambda**[12] or **exlambda**[19]). Thus, while amenable to fine-grained parallel algorithms in which synchronization is predicated on the values generated by tasks upon termination, future-style process abstractions are not convenient for expressing algorithms in which tasks have no manifest dependencies with one another, but which nonetheless must communicate data.

A distributed data structure[7] is one alternative abstraction for expressing concurrency that addresses some of these concerns. A distributed data structure is a concurrent object that permits many producers and consumers to concurrently access and update its contents via a well-defined synchronization protocol.

We have built a dialect of Scheme called TS/SCHEME that incorporates first-class distributed data structures as a basic data type. TS/SCHEME uses distributed data structures specified in terms of *tuple-spaces* as its main coordination device. The semantics of tuple-spaces described here shares broad similarity with tuple-spaces found in the Linda programming model[6], but differs in a number of significant respects as we describe below. Generally speaking, the utility of the tuple-space abstraction derives from a simple interface that usually requires only small perturbations to sequential code to build parallel programs, and an appealing metaphor for parallel programming in which task communication is anonymous, and totally decoupled from task synchronization.

There are four basic operations that can be performed on tuple-space structures: *read* a tuple, *remove* a tuple, *deposit* a tuple, and *deposit* a *process-valued* tuple. (A tuple is a heterogeneous ordered collection of values.) The first two are blocking operations that are permitted to execute only if the desired element exists in a tuple-space. The last defines a task creation operation.

The semantics of distributed data structures currently found in language models such as Linda have several attributes that make them difficult to use as a foundation for large-scale modular parallel programming. First, distributed data structures are *not* first-class entities; thus, all shared data is restricted to reside in one global repository. Second, Linda provides no mechanism to build synchronized abstractions of *named* heterogeneous collections (*e.g.,* records or objects). All shared data in Linda must be represented in terms of tuples whose fields are accessible by content, not by name. This restriction is a severe one for many kinds of symbolic applications. Third, the restrictive blocking semantics in Linda makes it difficult to use tuple-spaces as the basis for a building a concurrent object-oriented framework in which shared data is organized into a user-definable inheritance hierarchy. A natural inheritance model for tuple-spaces would permit tuple references made in tuple-space T to be resolved in T's parent(s) if T contains no suitable matching element. Finally, current implementations provide no mechanisms for programmers to easily annotate their programs to make the manipulation of tuple-spaces more efficient. Annotations to limit the size of a tuple-space, for example, are not part of the Linda language model. In general, this inability significantly limits the range of optimizations typically available to optimizers or program restructurers.

The extensions we propose are motivated by issues of modularity and expressivity; the design of the runtime system is influenced by our interest in using tuple-space objects as a concurrency device for *fine-grained* parallel computing[2]. Our intention is to preserve the cognitive simplicity of the tuple-space abstraction while enhancing its expressivity and implementability in several important respects. We enumerate the salient aspects of our system below, and elaborate on these points in the sections following:

1. Tuple-spaces are first-class objects: they can be defined explicitly, bound to variables, passed as arguments to (and returned as results from) procedures.

2. Tuple-spaces are comprised of two distinct components: a *binding repository* that defines a collection of bindings, and a *tuple repository* that defines a collection of tuples. Elements in a tuple repository are accessed by content, not by name; elements in a binding repository are accessed by name only.

3. Tuple-spaces adhere to an object-oriented protocol. Even though the intended use of a tuple-space (*e.g.,* as a vector, queue, set, etc.) can be specified by the programmer, the interface to any tuple-space is specified exclusively by the read, write, remove, and spawn operations described above. Thus, all tuple-space instances define the same operations regardless of their internal representation.

   In addition, tuple-spaces can be organized into an inheritance hierarchy. If a tuple-space T is specified to be the parent of another T', a read or remove operation sent to T' that is not satisfiable by T' is resolved by T and its parents. The ability to organize tuple-spaces into hierarchies makes TS/SCHEME a flexible concurrent object-oriented language.

4. There is no distinction between tasks or data in bindings or tuples. Thus, programs can deposit tasks as well as data; processes can match on either. The runtime system uses the uniformity of process and data to implement an efficient dynamic scheduling and throttling policy for fine-grained parallel programs.

The next section provides a description of the language, and gives an informal semantics of tuple-spaces and the operations permissible on them. Section 2.6 gives an example of how tuple-spaces may be used to build concurrent object-based systems. Section 3 gives a brief overview of STING[22], [23], a high-level operating system kernel for symbolic computing on which TS/SCHEME is implemented. Section 4 provides benchmark results.

2. The Language

Concurrency and coordination is introduced in TS/SCHEME via the `make-ts` primitive procedure. When applied, this procedure returns a reference to a new tuple-space object.

Tuple-spaces are first-class elements in TS/SCHEME. Giving first-class status to tuple-space objects supports modular parallel programming[13], [21]. By permitting tuple-spaces to be denoted, programmers may partition the communication medium as they see fit. To encapsulate a set of related shared data objects, we deposit them within a single tuple-space; this tuple-space can be made accessible only to those tasks that require access to these objects. Other tasks need not be aware of these objects, and *ad hoc* naming conventions need not be applied to ensure that data objects are not mistakenly retrieved by tasks which do not need them.

2.1. Tuple Repositories

Tuple repositories are manipulated via operations that deposit, read, remove tuples as described below. Tuples are defined using the tuple constructor, [. . .].

`(rd TS [` *tuple-template* `]` *body*`)` reads a tuple t from `TS`'s tuple repository that matches *tuple-template* and evaluates *body* using the bindings established as a consequence of the match. Assuming `TS` is defined by the expression, `(make-ts)`, this operation blocks until a matching tuple is encountered in `TS`.

For example, the expression:

```
(rd TS [ ?x 2 ?y] (+ x y))
```

reads a tuple from `TS` whose second field is `2`, binds the first field to `x` and the third to `y` and evaluates their sum. `X` and `y` are referred to as *formals*. Because Scheme supports latent polymorphic types, the system enforces no static type constraints on the matching procedure. Thus, a tuple with non-numeric first and third fields is an acceptable target for a match operation on the above template. Such a match results in a runtime error.

Besides TS/SCHEME objects, constants, and formals, a tuple template may also specify a "don't care" value that matches against any object. Thus, we can write:

```
(rd TS [ _ _ ?x 2 ] x)
```

to read a four tuple in `TS` whose fourth field is `2`; the contents of the first two fields in the matched tuple are ignored.

Tasks reading a tuple from `TS` have exclusive access to `TS` only while bindings for a template's formals are being established; all locks on `TS` are released prior to the execution of *body*.

`(get TS [` *tuple-template* `]` *body*`)` has a semantics similar to `rd` except that it atomically removes its matched tuple from `TS`'s tuple repository after establishing all relevant bindings.

`(put TS [` *tuple* `])` deposits *tuple* into `TS`'s tuple repository; the operation completes only after the tuple is deposited. Blocked `rd` or `get` operations that match on *tuple* are resumed as a consequence of this operation. Formals are not permitted in a tuple deposited by `put`. Tasks have exclusive access to `TS` until the operation returns.

Although not specified in the semantics, the implementation of TS/SCHEME described here gives higher priority to resuming blocked readers than blocked removers. Thus, when a tuple is deposited on which a number of readers and removers are blocked, all blocked readers, and at most one blocked remover are resumed. This implementation maximizes the number of threads resumed by a tuple deposition operation.

`(spawn TS [` *tuple* `])` creates a lightweight process for each field in *tuple* and returns immediately thereafter. Thus, the expression:

```
(spawn TS [ (f x) (g y) ])
```

deposits a two-tuple into `TS`'s tuple repository. These fields are lightweight tasks computing `(f x)` and `(g y)`. Once both tasks complete, the tuple quiesces to a passive (data) tuple containing the values yielded by these tasks. In the above example, both `f` and `g` are assumed to be lexical closures and their applications evaluate in the environment in which they are defined, and *not* in the call-time dynamic environment in which they are instantiated.

Rd and get operations can match on process-valued tuples. We defer discussion of this feature until Section 3.

Tuple-spaces are represented as Scheme objects, obey the same storage semantics as any other object, and may be garbage collected if there exist no outstanding references to them.

Since it is expected that communication among tasks in TS/SCHEME will be performed exclusively via the tuple operations given above, the meaning of a program in which tuple referenced objects are mutated is unspecified. For example, the following program fragment need not raise an error, but its meaning is ill-defined:

```
(let ((foo (make-vector 1))
      (TS (make-ts)))
  (vset foo 0 'hello)
  (spawn (make-ts) [ (put TS [ foo ])
                     (rd TS [ ?x ] (vref x 0)) ])
  (vset foo 0 'goodbye))
```

The result of the rd operation will be either the symbol hello or the symbol goodbye. The semantics does not constrain an implementation to make a copy of foo when it is deposited into TS; programmers should not rely on such implementation details, however, in writing TS/SCHEME code.

2.2. Immutable Tuple Spaces

A tuple-space T can be "closed" via the operation (close-ts T). A closed tuple-space is an immutable object – no further tuples can be deposited into it, and no tuples may be removed from it. In terms of implementation, no synchronization is required to access the elements of a closed tuple-space.

An operation that attempts (a) to read a non-existent tuple from a closed tuple-space, or (b) to write a tuple into a closed tuple-space induces a runtime error. Furthermore, tasks blocked in a tuple that becomes closed will never become unblocked.

2.3. Bindings

A significant point of departure of TS/SCHEME from other distributed data structure systems is its fundamental support for *bindings* as a basic unit of communication. Tuple-spaces as described thus far encapsulate an anonymous collection of data values retrieved as a consequence of a pattern-match operation. While useful for expressing recursive, self-relative or associative data structures (*e.g.,* lists, vectors, streams, queues, etc.), the tuple repository abstraction is a poor fit for building other kinds of named structured objects such as modules, classes, or records. Despite obvious differences in their semantics, these objects all fundamentally manipulate *bindings* of labels to values. Modular programming in general, and symbolic computing in particular, makes heavy use of such structures. Consequently, TS/SCHEME seriously supports the construction and access of bindings in its coordination substrate.

Binding tuples and templates are defined using the binding constructor, $\{\ldots\}$. These objects are manipulated by the same tuple operations described above.

Bindings are orthogonal to ordinary data and reside in a tuple-space's binding repository. When used as an argument to a `put` or `spawn` operation, a binding tuple has the form:

$\{$ `(`x_1 E_1`)` `(`x_2 E_2`)` ... `(`x_n E_n`)` $\}$

Each of the x_i are labels, and each of the E_i are expressions. When used in conjunction with a `put` operation, for example, the expressions are first evaluated to yield values v_1 , v_2 , ... , v_n. Each label/value pair, (x_i , v_i) is then deposited into the specified tuple-space's binding repository.

For example, the expression:

```
(let ((bar 2)
      (bam 3))
  (put ts { (foo 1) (bar bar) (bam 4) }))
```

atomically deposits three bindings into `ts` – the first binds the identifier `foo` to `1`; the second binds the identifier `bar` to its value in the lexical environment; the third binds the identifier `bam` to 4. Bindings for these labels deposited by the above `put` supersede previously established bindings for these same labels.

We can similarly deposit process-valued bindings:

```
(spawn ts { (foo (f x)) (bar (g y)) })
```

This expression deposits two bindings into tuple-space initially bound to two tasks (computing `(f x)` and `(g y)` resp.) The values yielded by these tasks ultimately replace the references to the tasks themselves; thus when the task computing `(f x)` completes with value *v*, the binding of `foo` to the task object is replaced with a binding of `foo` to *v*.

A binding template takes the form,

```
{ ?x ?y ... ?z }
```

where `x`, `y`, ... `z` are labels. When evaluated in the context of a `rd` or `get` operation, *e.g.,*

`(rd ts {` ?x_1 ?x_2 ... ?x_n `}` E`)`

each label x_i acquires a value as defined by its binding in `ts`'s binding repository; these bindings are then used in the evaluation of E. E is evaluated only after all formals in the binding template have acquired a value, *i.e.,* only after bindings for all the x_i have been deposited into `ts`. The order in which bindings in a tuple are established is unspecified.

For example, evaluating the expression,

```
(rd ts { ?foo ?bam } (+ foo bam))
```

introduces a new scope with two names, `foo` and `bam`. The binding values for these names are the values extant at the time the name-lookup in `ts`'s binding repository occurs. Either the binding for `foo` or `bam` can be established first. During the interval between the establishment of the first and second bindings, other threads can access and manipulate `ts`.

Using bindings, one can build concurrent analogues of sequential named structures. For example, the following program fragment creates a tuple-space that behaves as a shared record object:

```
(let ((ts (make-ts)))
  (put ts { (foo 1) (bar 2) })
  ts)
```

If this object is named R, the expression,

```
(rd R { ?foo } foo)
```

behaves as a field selection operation, and the expression,

```
(get R { ?foo }
  (put R { (foo new-value) }))
```

deposits a new binding for foo into R. Tasks that read bindings from R block if the desired bindings are absent.

Bindings add two important pieces of functionality to a coordination model. First, they acknowledge the importance of named objects in symbolic computing, and elevate such structures as *bona fide* shared coordination entities. While named structures can be simulated using just tuples, *e.g.*,

```
(put ts ['foo 1])
(put ts ['bar 2])
   ...
(rd ts ['foo ?x] x)
(rd ts ['bar ?y] y),
```

supporting bindings directly simplifies implementation and adds expressive power. Using symbols as above to represent bindings provides no protection against name clashes, and requires a global protocol that fixes the representation of bindings in terms of tuples.

Second, given first-class tuple-spaces, the ability to inject and read bindings provides the necessary flexibility to build a robust object system with tuple-spaces as the representation structure for abstract user-defined objects. We elaborate on this point in Section 2.6.

2.4. Types

The tuple repository component of a tuple-space may be represented as semaphores, sets, bags, vectors, protected variables, streams, queues or tasks. The type specification is a reflection of the intended use of the repository; in the absence of any type annotation, the elements in a tuple repository are represented via associative hash tables.

Regardless of the annotation affixed to a tuple-space, its interface remains uniform although the structure of tuples applied to tuple operations may be restricted. Thus, if TS has type type/var,

```
(define TS (make-ts type/var))
```

the operation,

```
(rd ts [100] ...)
```

is ill-defined since protected variables are not intended to be accessed by content, whereas the operation

```
(rd ts [?x] ...)
```

is well-defined, and returns the value of the protected variable.

Similarly, tuples deposited into a tuple-space represented as a vector must be of the form `[i, v]` where `i` is an integer index. `Rd` and `get` operations must operate over templates of the form, `[i, ?v]`; in the default case, these operations block if no value has been deposited into index `i` of the corresponding vector. During compilation, annotations are used to help type-check tuple operations in certain well-defined instances; thus, the following expression raises a compile-time error provided `TS` is not assigned in the `let` body:

```
(let ((TS (make-ts type/var)))
  ...
  (put TS [1 2 3])  ;;; ill-formed use of TS
  ...)
```

2.5. Attributes

A tuple-space can be associated with a set of attributes that specify properties of its implementation. There are several important attributes and attribute classes over tuple-spaces available in TS/SCHEME:

1. *size*: A size attribute defines an upper-bound on the number of tuples and bindings that can occupy a tuple-space. A tuple operation whose evaluation would violate this bound signals a runtime error.

2. *parent*: A parent attribute augments the semantics of tuple access by enabling tuple-spaces to follow an inheritance protocol on match failure. The expression:

   ```
   (define TS (make-ts (ts/parent Parent)))
   ```

   creates a tuple-space `TS` with parent `Parent`. `Rd` or `get` operations on `TS` which fail to find a suitable matching tuple (or binding) perform their search on `Parent`; a match (or binding) failure in `Parent` causes search to resume in its parent and so forth. The executing task blocks in `TS` if all tuple-spaces in a parent chain have been examined, and no matching tuple is encountered.

   Allowing tuple-spaces to be linked together in this fashion elevates their utility to that of full-fledged objects that can be made to share common data. As we describe in Section 2.6, such capability permits the realization of a concurrent object-based system using tuple-spaces as the basic object constructor.

3. *Daemon attributes*: A daemon attribute is a procedure that is invoked whenever a specified tuple operation is applied on a tuple-space. These attributes are of the form: `(ts/`*type constructor proc*`)` where *type* is one of `put, rd, get` or `spawn`, *constructor* is either `data` or `binding`, and *proc* is an expression that yields a procedure of one argument.

   Attributes are evaluated only after the appropriate operation has completed; thus a `ts/put` daemon is invoked only after a tuple has been deposited; daemon attributes are always applied to a tuple representation structure.

   For example, the following program fragment creates a tuple-space `TS` that implicitly maintains a copy of all binding tuples that are deposited into its tuple repository:

```
(define copy (make-ts))
(define TS (make-ts (ts/put binding (lambda (tuple) (put copy tuple)))))
```

Although other expressions may `rd` or `get` binding tuples from `TS`, `copy` serves as a log of all binding tuples deposited into `TS`; such tuples extracted from `TS` are not automatically extracted from `copy`.

Daemon attributes are also useful for building simple extensions to the core set of tuple operations. For example, the following fragment maintains a counter of the number of tuples currently found in `TS`; since tasks have exclusive access to a tuple-space while depositing a tuple or establishing bindings, no synchronization is required to access this counter:

```
(define TS-count 0)
(define TS (make-ts
             (ts/put (lambda (tuple) (set! TS-count (+ TS-count 1))))
             (ts/get (lambda (tuple) (set! TS-count (- TS-count 1))))))
```

Attributes are defined when a tuple-space is created. Certain attributes (such as *size*) are immutable, but others (such as *parent*) can be side-effected.

2.6. Examples

Figures 1 and 2 give two examples of TS/SCHEME programs.

The first program describes an implementation of I-structures[4]. As found in Id, I-structures are write-once arrays; an attempt to read an unwritten element of an I-structure causes the accessing expresssion to block until a value is written. The use of tuple-spaces as the representation object for I-structures lets us generalize their functionality in some important respects; the implementation described here is similar in some respects to the definition of M-structures discussed in [5]. The implementation provides four operations on I-structures:

1. `(make-I-structure` *size*`)` creates an I-structure vector. An I-structure vector is represented as a tuple-space with two bindings. The first binds `elements` to a tuple-space that will hold I-structure values. The second binds `status` to a tuple-space vector of length `size` used to determine if an I-structure cell has a value.

2. `(I-ref` *rep i*`)` returns the value of the i^{th} I-structure component of *rep*, blocking if the structure is empty.

3. `(I-set` *rep i v*`)` atomically sets the i^{th} element of *rep* to *v* provided that this element has not already been previously written. The tuple-space bound to `status` is used to synchronize multiple writes to an I-structure.

4. `(I-remove` *rep v*`)` is an operation that demonstrates the use of tuple-spaces as content addressable data structures.

   Given a value *v* and an I-structure *rep*, `I-remove` creates a new thread that re-initializes all I-structure elements that contain *v*. The thread repeatedly removes a tuple in *rep*'s

```
(define (make-I-structure size)
  (let ((rep (make-ts)))
    (put rep { (elements (make-ts)) })
    (put rep { (status (make-ts (type/vector size))) })
    (let -*- ((i 0))
         (cond ((= i size) rep)
               (else (rd rep { ?status }
                       (put status [i '#f]))
                       (-*- (+ i 1)))))))

(define (I-ref rep i)
  (rd rep { ?elements }
      (rd elements [i ?v] v)))

(define (I-set rep i v)
  (rd rep { ?status }
    (get status [i ?state]
         (cond (state (error ``I-structure currently written''))
               (else (rd rep { ?elements }
                        (put elements [i v]))
                     (put status [i '#t]))))))

(define (I-remove rep v)
  (spawn (make-ts)
    [(rd rep { ?elements ?status }
         (let -*- ()
              (get elements [?i v]
                   (get status [i _]
                        (put status [i '#f])))
              (-*-)))]))
```

Figure 1. A generalized implementation of I-structures using first-class tuple-spaces.

`elements` tuple-space that contains *v*, setting the appropriate status field to false. In the implementation shown, these threads run forever, but it is straightforward to augment their functionality such that they terminate upon a user-specified condition.

The program shown in Fig. 2 is a concurrent object-based definition of a `point` and `circle` abstraction. Instances of these abstractions are represented as tuple-spaces that hold bindings for all relevant methods. A `point` object defines two methods, one to compute the distance of the point from the origin, and another to determine whether the current point is closer to the origin than a point provided as an input.

Circles inherit the `ClosertoOrig` method found in points, but redefine a new `DistfromOrig` procedure. Note that a call to `ClosertOrig` with an instance of a circle as the `self` argument will cause the definition of `DistfromOrig` found in circles, not the one defined for points, to be used. Thus, this implementation realizes a late-binding semantics found in object-based systems such as Smalltalk[14] or Self[34].

These definitions define a concurrent program[1], [3], [36] – there may be many methods in `point` and `circle` instances evaluating simultaneously. For example, the following fragment returns a procedure which when applied to a circle instance creates several concurrent threads manipulating both circle instance `C` and point instance `P`:

```
(let ((C (make-circle a b r))
      (P (make-point x y)))
  (lambda (my-C)
    (spawn (make-ts) [ (f (send C DistfromOrig))
                       (g (send C ClosertoOrig my-C))
                       (send P Update-x new-x)
                       (send P Update-y new-y)])))
```

Methods in `C` and `P` may all evaluate simultaneously with one other; thus, `C`'s distance from origin, and its distance relative to `my-C` are computed in parallel. Similarly, `P`'s `x` and `y` coordinate are updated concurrently as well. As with ordinary tuples, bindings are accessed and manipulated in well-defined atomic steps.

The system naturally supports dynamic inheritance[8], [26]. Given a point instance, we can deposit a new method simply by applying a tuple operation to the object:

```
(put point-instance { (new-method-name new-method) })
```

Operations which send messages to *new-method-name* block until the depositing task executes the above tuple operation.

Simple analyses of TS/SCHEME program structure can reduce the overheads involved in manipulating bindings considerably. One important optimization that may be applied occurs if the set of bindings deposited into a binding repository is statically known. For example, consider the following expression:

```
(let ((TS (make-ts)))
  (put TS {(a 1) (b 2)})
  (spawn (make-ts) [(f TS) (g TS)]))
```

If neither `f` nor `g` deposit bindings into `TS` other than `a` or `b`, we might implement `TS`'s binding repository as a two-field record with labels for `a` and `b`. `Get` operations on `a` or `b` reinitialize the corresponding record slot; subsequent `rd` or `get` operations on this slot block until a new binding-value is deposited.

```
(define (make-point x y)
  (let ((obj (make-ts (ts/parent default-object))))
    (put obj { (x x) (y y)
                (Update-x
                  (lambda (self new-x) (get self {?x}
                    (put self {(x new-x)}))))
                (Update-y
                  (lambda (self new-y) (get self {?y}
                    (put self {(y new-y)}))))
                (DistfromOrig
                  (lambda (self)
                    (rd self { ?x ?y }
                        (sqrt (- x y)))))
                (ClosertoOrig
                  (lambda (self p)
                    (< (send self DistfromOrig)
                       (send p DistfromOrig))))) })
    obj))

(define (make-circle a b r)
  (let* ((super (make-point a b))
         (obj (make-ts (ts/parent super))))
    (put obj { (radius r)
               (Update-radius
                 (lambda (self new-radius)
                   (get self { ?radius }
                       (put self {(radius new-radius)}))))
               (DistfromOrig
                 (lambda (self)
                   (max (- (send super DistfromOrig)
                           (rd self { ?radius } radius))
                        0))) })
    obj))

(send Object Method . args) ≡
(rd Object { ?Method } (apply Method Object args))
```

Figure 2. A concurrent object program using tuple-spaces as the object representation.

For object-based programs in which bindings are only created at the time the object is instantiated and the parent hierarchy is not modified once established, compiling binding repositories as records can lead to other optimizations. Consider the following program fragment:

```
(define O (make-ts (ts/parent P)))
(put O (M1 method 1)
       (M2 method 2)
          ...
       (Mn method n))
```

Deciding an efficient representation for `O`'s binding repository requires knowledge of the set of potential bindings in `P` and `O`. For example, if the threads accessing `O` manipulate only bindings for M_1, M_2, ..., M_n, `rd` and `get` operations on `O`'s bindings are translated into expressions that use offsets into a record representation containing these bindings. If the same analysis is applied to `O`'s ancestors, references to ancestor methods sent in messages to `O` can be compiled into offsets in the appropriate record containing the method definition; such records define the representation of the binding repository for tuple-spaces such as `O` and `P`. This approach of customizing the representation of binding repositories based on their use is similar to strategies employed in other optimizing compilers for late-binding languages[8], [31].

3. Runtime Support

Efficient runtime implementation of first-class tuple-spaces requires at the minimum (a) cheap creation and management of lightweight tasks, (b) flexible scheduling policies that can be dynamically instantiated to support different programming paradigms, (c) support for fine-grained parallel computing via efficient process throttling and scheduling protocols, and (d) storage management capabilities sensitive to tuples and tuple-spaces.

STING is an operating system kernel for high-level symbolic programming languages that provides these capabilities. Details of the system are given in [22], [23]. The system is implemented entirely in Scheme with the exception of a few operations to save and restore thread state, and to manipulate hardware locks. Tuple-space operations translate to operations on threads and ordinary Scheme data structures.

Threads and *virtual processors* are two novel abstractions in the STING design that support distributed data structures.

Threads define a separate locus of control. They are represented as a small Scheme object closed over a procedure which executes as a separate lightweight task. Storage for threads is deferred until the thread is actually about to execute; delaying storage allocation improves program locality and permits some important runtime optimizations which we describe below. When a thread is an *evaluating* state, it is associated with an evaluation context or *thread control block* (TCB). TCBs contain dynamic state information, and storage (*e.g.,* stacks and heaps). All evaluating threads execute using separate TCBs unless data dependencies warrant otherwise.

Besides threads, STING supports the construction of first-class virtual processors (VPs). Virtual processors abstract all scheduling, migration and policy decisions associated with the management of threads. Virtual processors combine to form virtual machines which define a single address space in which threads may execute. Virtual machines themselves are mapped to physical machines which are faithful abstractions of an underlying architectural platform. STING assumes its physical processors execute on top of a shared memory or shared virtual memory[25] substrate.

Thread control blocks are created in a LIFO pool associated with each VP. When a thread terminates, its TCB is restored into the TCB pool of its VP. Every VP has a TCB threshold; if exceeded, a certain fraction of TCBs are migrated from the VP to a global VP pool associated with the virtual machine. Similarly, if a VP has no available TCBs, and one is required, the global pool is first consulted before a new TCB is created. Recycling TCBs improves runtime locality and minimizes overheads in running new threads.

Policy decisions such as thread scheduling, migration, load-balancing, etc. are encapsulated inside a *thread policy manager*. Virtual processors are closed over potentially distinct policy managers. Since virtual processors are first-class, and since all policy managers provide a standard interface[3], programmers can dynamically close individual virtual processors over distinct policy managers without loss of efficiency. Customizing policy decisions permits applications with different runtime requirements to execute under scheduling protocols tailored toward their specific needs. In particular, TS/SCHEME programmers can build virtual machines whose processors are closed over policy managers best suited for the paradigm being implemented (*e.g.,* master-slave programs may execute on VPs which schedule threads using a FIFO queueing discipline with preemption, fine-grained result-parallel programs may execute under a LIFO non-preemptive scheduler, etc.).

3.1. Efficient Support for Fine-Grained Active Tuples

STING provides support for fine-grained parallel symbolic computing by permitting threads to share evaluation contexts whenever data dependencies warrant. In a naive implementation, a thread T which accesses the value produced by another must block until the value is generated. This behavior is sub-optimal – cache and page locality is compromised, bookkeeping information for context switching increases, and processor utilization is not increased since the original task must block until the new task completes.

Consider the following optimization: a procedure t associated with a thread, T that has not yet started evaluating can be evaluated within the evaluation context of a thread S that demands t's value without violating the program's semantics. The rationale is straightforward: since T has not been allocated storage and has no dynamic state information associated with it, t can be treated as an ordinary procedure and evaluated using the TCB already allocated for S. In effect, S and T share the same evaluation context. We refer to such an operation as *absorption*. A thread is absorbed if its thunk and dynamic state execute using the stack and heap associated with another already evaluating thread.

Thread absorption is therefore tantamount to a dynamic demand-driven evaluation strategy for scheduled threads – any thread that requires the value of another that has not yet started evaluating can directly demand its value without blocking. A scheduled thread is not constrained to be demanded before it can evaluate, however; such a thread can execute independently of its relationship with any other thread provided that resources are available.

Consider a task P that executes the following expression:

```
(rd TS [ x1 x2 ] E)
```

where `x1` and `x2` are non-formals. Assume furthermore that an active tuple is deposited into `TS` as a consequence of the operation,

```
(spawn TS [ E1 E2 ])
```

This operation schedules two threads (call them T_{E_1} and T_{E_2}) responsible for computing E_1 and E_2. If both T_{E_1} and T_{E_2} complete, the resulting (passive) tuple contains two determined threads. Recall that because there is no distinction in TS/SCHEME between active tuples (*i.e.,* tuples with process-valued fields) and passive tuples (*i.e.,* tuples containing only Scheme values), an active tuple is a valid argument for a match operation.

If T_{E_1} has not yet begun evaluating at the time P executes, however, P is free to absorb it, and then determine if its result matches `x1`. If a match does not exist, P may proceed to search for another tuple, leaving T_{E_2} still in a scheduled state. Another task may subsequently examine this same tuple and absorb T_{E_2} if warranted. Similarly, if T_{E_1}'s result matches `x1`, P is then free to absorb T_{E_2}. If either T_{E_1} or T_{E_2} are already evaluating, P may choose to either block on one (or both) thread(s), or examine other potentially matching tuples in `TS`. The semantics of tuple-spaces impose no constraints on the implementation in this regard.

Thread absorption is similar in some respects to load-based inlining[16], [35] and lazy task creation[28] insofar as it is a mechanism to throttle the creation of new lightweight threads of control. Unlike load-based inlining, thread absorption can never cause processor starvation or avoidable deadlock since it is applied only when a manifest data dependency (*i.e.,* a producer/consumer relationship) exists between two threads. Lazy task creation is an optimization found in Mul-T[24] that inlines potential threads, but permits the inlining operation to be revoked if processors becomes idle. In this scheme, a thread is created only when a processor becomes available to evaluate it. Unlike thread absorption, the decision to revoke an inlined thread is not based on runtime dataflow constraints, but exclusively on processor availability.

STING's combination of first-class threads and thread absorption allows us to implement quasi-demand driven fine-grained (result) parallel programs using shared data structures. In this sense, the thread system attempts to minimizes any significant distinction between structure-based (*e.g.,* tuple-space) and dataflow style (*e.g.,* future/touch[16]) synchronization. In this respect, TS/SCHEME addresses a major criticism of Linda and other related systems insofar as programmers may ignore many details concerning process granularity and work distribution in their programs without compromising performance. In implementations of systems such as C.Linda, the inability to minimize the number of actual evaluation contexts created in a program by exploiting data dependencies between active tuple writes and reads severely limits the kind of algorithms which can be efficiently supported; a C.Linda program that spawns n tasks will evaluate them in n separate contexts. No thread absorption will be performed, and threads that attempt to read the tuples holding these threads will block until they complete.

```
(define (queens n d)
  (let try ((d d)
            (rows-left n)
            (free-diag1 -1)
            (free-diag2 -1)
            (free-cols (- (ashl 1 n) 1)))
    (let ((free (logand free-cols
                        (logand free-diag1 free-diag2))))
      (let loop ((col 1))
        (let ((my-try (lambda (d)
                        (try d
                             (- rows-left 1)
                             (+ (ashl (- free-diag1 col) 1) 1)
                             (ashr (- free-diag2 col) 1)
                             (- free-cols col)))))
          (cond ((> col free) 0)
                ((= (logand col free) 0)
                 (loop (* col 2)))
                ((= rows-left 1)
                 (+ 1 (loop (* col 2))))
                (else
                 (let ((Q (cond ((> d 0)
                                  (let ((Q (make-ts type/var)))
                                    (spawn Q  [(my-try (- d 1))])
                                    Q))
                                 (else (my-try 0))))
                        (other-solns (loop (* col 2))))
                   (+ (cond ((zero? d) Q)
                            (else (rd Q [?v] v)))
                        other-solns)))))))))
```

Figure 3. A fine-grained tree-structured TS/SCHEME program.

Fig. 3 shows a fine-grained implementation of the well-known n-queens problems[4]. In this formulation, a queen is placed on one row of the chessboard at a time. Threads are then spawned to find all solutions stemming from the current configuration; threads are not created after a specified cutoff depth in the tree is exceeded. Bit vectors are used to build a compact representation, leading to fine thread granularity. Since there exist manifest data dependencies among spawned threads in this example (a queen on one row needs to know the positions of queens on all other rows), many scheduled threads can be absorbed, limiting the overall storage requirements needed by this program. We provide benchmark results for this program in Section 4.

3.2. Storage Management

STING threads are closed over a private heap, private stack, and a shared heap. Data allocated on private heaps is local to a thread, and can be garbage collected without requiring synchronization with other threads[30]. Data which is shared among threads is allocated on a shared heap. Whenever a reference to an object in a local heap escapes that heap, the object is copied to a shared heap. Heaps and stacks are first-class objects.

The TS/SCHEME implementation allocates a shared heap for every virtual processor. A thread executing on virtual processor V, allocates tuples on V's associated heap; no locking is required if preemption is disabled during the allocation. Unlike private heaps, this shared heap is garbage collected synchronously with respect to the group of threads which access it. Any thread which attempts to access tuples found on this heap during a garbage collection blocks until the collector is finished. In the abstract, when a tuple references an object already allocated on some thread's private heap, the object is copied to some shared heap; this heap may be the virtual processor heap holding tuples, or a shared heap associated with the thread's group [22].

The runtime overhead associated with copying objects from local to shared heaps can be reduced by having objects pre-allocated on a shared heap if it can be determined that they will subsequently be referenced within a tuple operation. For example, in the following code fragment,

```
(let ((TS (make-ts))
      (v (make-vector 1000)))
  ...
  (put TS [v])
  TS)
```

it is reasonable to allocate `v` on a shared heap since `v` is referenced within the tuple deposited into `TS`, and the lifetime of the vector bound to `v` is tied to the lifetime of `TS`. If the vector were allocated on the private heap of the thread executing this fragment, `v` would be copied to `TS`'s shared heap during evaluation of the `put` operation; efficiency would be compromised in this case. Compile-time analysis of data locality in threads is an important component of future TS/SCHEME implementations.

4. Benchmarks

The sequential timings shown in Table 1 were measured on a 50 MHz Silicon Graphics MIPS R4000 workstation with 96 megabytes of memory; the R4000 has 8 kilobyte primary instruction and data caches, and a unified 1 megabyte secondary cache.

The multiprocessor timings were measured on an eight processor Silicon Graphics PowerSeries shared-memory machine; each node consists of a 75 Mhz MIPS R3000 processor. Because of STING's aggressive treatment of data locality, its separation of evaluation contexts from threads which use them, its use of thread absorption to increase dynamically increase thread granularity, and the lightweight costs of tuple operations, we expect these programs to scale well even on larger processor ensembles.

Table 1. Baseline times for tuple-space operations.

Operation	Time (μ-seconds)
Create	40 100
Put	61 400
Rd	73 91
Put/Get	136 235
Ping-pong	750

Each tuple operation was timed on both a general fully associative tuple-space as a well as a specialized one (for the timings, tuple-spaces created with a `type/var` annotation were used); the smaller times shown for each operation are the times recorded for the specialized operations.

Put/Get measures the cost of creating two tuple-spaces T_1 and T_2, and synchronously depositing and removing a singleton tuple from them. Ping-pong creates two tasks closed over two `type/var` tuple-spaces. Task A deposits a singleton tuple into tuple-space T_1 and waits for a tuple to be deposited into T_2; Task B waits for a tuple to be deposited into T_1, and then deposits a tuple into T_2. The time measured is the cost for (a) creating and scheduling STING threads for tasks A and B, (b) creating tuple-spaces T_1 and T_2, and (c) depositing a tuple into T_1 and removing a tuple from T_2.

Tables 2 and 3 shows timings and relevant statistics for five benchmarks. The tuple operations shown in Table 3 gives an indication of how much synchronization and communication occurs in the corresponding benchmark. The benchmarks were chosen to exercise different aspects of both the language and runtime system. All benchmarks are structured not to trigger garbage collection. The super-linear speedups seen in the Hamming benchmark are attributable to significant cache effects in the shared memory system since doubling the number of processors effectively doubles the size of the primary cache. In benchmarks which are storage intensive and which exhibit significant locality (such as Hamming), cache behavior can be a dominating factor in the recorded timings.

N-Queens is the n queens problem shown in Fig. 3. The timings are shown for a 14x14 chessboard with a thread cutoff depth of $\log n + 1$ where n is the number of processors.

Alpha-Beta is a parallel implementation of a game tree traversal algorithm. The program does α/β pruning[11], [18] on this tree to minimize the amount of search performed. The program creates a fixed number of threads; each of these threads communicate their α/β values via tuple-spaces. The current "best" values for α and β cutoffs are consulted by these threads up to some fixed depth in the search tree. The input used consisted of a tree of depth 10 with fanout 8; the depth cutoff for communicating α and β values was 3. To

Table 2. Timing and efficiency ratios for a benchmark suite. The first row for each benchmark are wallclock times measured in seconds; the second row are efficiency ratios. The efficiency of a program P on N processors is defined as the percentage ratio of P's ideal performance on N processors over P's realized performance. We define P's ideal performance on N processors to be P's single processor time divided by N.

Benchmark	**Timings and Efficiency**			
	1	2	4	8
N-Queens	103.8	52.6	26.4	14.7
	100	98.6	98.2	88.2
Alpha-Beta	108	57.6	31.9	16.7
	100	93.7	84.6	80.8
Hamming	39.9	16.74	7.1	3.4
	100	119.1	140	146
N-Body	751	363	251	137
	100	96.7	75	69
Primes	219	112.6	58.1	34.1
	100	97.4	94.4	81

Table 3. Tuple Operations on 8 processors.

Benchmark	**Tuple Operations**			
	Put	Get	Rd	Spawn
N-Queens	0	0	11167	11167 11108 (absorbed)
Alpha-Beta	23	20	597	11
Hamming	10367	7417	17405	8
N-Body	198630	191620	1165492	49
Primes	86752	43418	1819380	8

make the program realistic, we introduced a 90% skew in the input that favored finding the best node along the leftmost branch in the tree.

Hamming computes the extended hamming numbers upto 1,000,000 for the first five primes. The extended hamming problem is defined thus: given a finite sequence of primes *A, B, C, D,* ... and an integer n as input, output in increasing magnitude without duplication all integers less than or equal to n of the form

$$A^i \times B^j \times C^k \times D^l \times \ldots$$

We structure this problem in terms of a collection of mutually recursive lightweight threads that communicate via streams; each stream is implemented as a tuple-space. This program is the most communication sensitive of the benchmarks presented, and also exhibits the most thread locality.

N-body simulates the evolution of a system of bodies under the influence of gravitational forces. Each body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. This benchmark ran the simulation on 3500 bodies over six time-steps.

We implemented the two-dimensional version of the Barnes-Hut algorithm. The algorithm exploits the idea that the effect of a cluster of particles at a distant point can be approximated by one body whose location is the center of mass of the cluster and whose mass is the sum of the masses in the cluster. It thus assumes that all bodies are contained in a fixed sized cube (in the 3-dimensional case) or in a square (in the 2-dimensional case).

This program generates a number of relatively coarse-grained threads independent of the actual input. Barrier synchronization is used to ensure that all forces are computed before a new iteration is initiated. To optimize storage usage, a new set of threads is created on every iteration; storage generated during one iteration becomes available to threads created in subsequent ones; the overall aggregate storage requirements are thus minimized.

Primes computes the first million primes using a parallel master/slave version of the Sieve of Erasthosenes algorithm. Workers compute primes within a given range, repeatedly fixing on new ranges via synchronization objects represented as distributed data structures; discovered primes are also recorded in a distributed data structure (represented as a vector tuple-space) that is also used to determine the primality of numbers in chunks subsequently examined.

5. Related Work and Conclusions

Besides C.Linda and TS/SCHEME, distributed data structures are available in a number of explicitly parallel programming languages. Some notable examples are the blackboard object in Shared Prolog[2], stream abstractions in Flat Concurrent Prolog[33], Concurrent Smalltalk's distributed objects[17] and its closely related variant Concurrent Aggregates[9], and the I-structure in Id[4].

TS/SCHEME is distinguished from these other efforts in several important respects. By way of comparison, synchronization in other related languages takes place either through process constructor primitives (*e.g.,* future and touch in Mul-T[24], lightweight threads in ML Threads[10], [29]) and shared objects (*e.g.,* read-only variables in Concurrent Prolog, constraint-based stores in Herbrand[32], process closures in Qlisp[12]). Task communication in TS/SCHEME, on the other hand, is decoupled from task instantiation. This attribute makes it possible for tasks to initiate requests for the value of objects even if the object itself has not yet been created, and to collectively contribute to the construction of shared objects.

Concurrent object-oriented languages such as ABCL/1[36] or Actors[1] permit many messages to be handled simultaneously by an object, but do so in a specialized framework. In ABCL/1, concurrent objects behave as monitors insofar as an object may handle only a single message at a time, and the state of an object is inaccessible to any task other than its creator. Unlike monitors, ABCL/1 objects do not adhere to a bi-directional message passing protocol – the sender of a message does not need to wait for a reply from the recipient before continuing. The Actor model permits multiple threads of control or *actors* to exist, but assumes all messages sent to an actor are handled in FIFO order. Like the systems described above, and unlike TS/SCHEME, both ABCL/1 and Actors require objects to be created before messages can be sent to them. This is because synchronization structures in these systems are defined to be implicitly part of a task's state. Both systems also assume a message-passing copy protocol; all shared data must be encapsulated inside a concurrent object. Furthermore, neither ABCL/1 nor Actors easily support an inheritance structure among objects; such a structure is conveniently expressed in TS/SCHEME since programmers can construct inheritance trees using tuple-spaces.

TS/SCHEME is an attempt to increase the flexibility of distributed data structures in the context of a highly-parallel symbolic programming environment. First-class tuple-spaces contribute to modularity and fit naturally in the programming style supported by Scheme and related higher-order languages. Permitting bindings to reside as elements in a tuple-spaces elevates tuple-spaces to the status of a first-class (parallel) environment[20] without affecting the semantics of data tuples and repositories. Attributes and type information give programmers flexibility to control the behavior of tuple-space objects, add greater protection, and significantly simplify implementation. Permitting read and remove operations on tuple-spaces to fail and default to a specified parent allows programmers to build concurrent object-based programs. We argue that these extensions add only slight conceptual overhead to the basic model, but provide significant expressivity advantages, and present a very efficient platform for building modular asynchronous parallel programs.

Acknowledgments

The STING kernel on which TS/SCHEME runs was implemented by James Philbin. Monika Rauch implemented several of the benchmarks, and has been a major user of the system; Henry Cejtin contributed several useful suggestions to improve the quality of this paper.

Notes

1. A *task* or *thread* is a lightweight process that executes in the same address space as its parent.
2. We consider a program to be *fine-grained* if it defines tasks whose execution cost, in the absence of any compile-time or runtime optimizations, is roughly proportional to the execution cost of the operations needed to create them.
3. The interface must provide operations to insert a thread into a thread queue, get a new runnable thread, etc..
4. This solution is adapted from [27].

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
2. V. Ambriola, P Ciancarini, and M. Danelutto. Design and Distributed Implementation of the Parallel Logic Language Shared Prolog. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 40–49, March 1990.
3. Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented System. In *Proceedings of the European Conf. on Object-Oriented Programming*, pages 234–242, 1987.
4. Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.
5. Paul Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Proceedings of the ACM Symposium on Functional Programming and Computer Architecture*, pages 538–568, August 1991. Published as Springer Verlag LNCS 523.
6. Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.
7. Nick Carriero, David Gelernter, and Jerry Leichter. Distributed Data Structures in Linda. In *Proceedings of the* 13^{th} *ACM Symposium on Principles of Programming Languages*, January 1986.
8. Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 146–160, June 1989.
9. Andrew Chien and W.J. Dally. Concurrent Aggreates (CA). In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 1990.
10. Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.
11. Raphael Finkel and John Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19(1):89–106, 1982.
12. R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 25–44, August 1984.
13. David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of the Conf. on Parallel Languages and Architectures, Europe*, pages 20–27, 1989.
14. Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Press, Reading, Mass., 1983.
15. Ron Goldman, Richard Gabriel, and Carol Sexton. QLisp: An Interim Report. In *Parallel Lisp: Languages and Systems*, pages 161–182. Springer-Verlag, LNCS 441, 1990.
16. Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
17. Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 101–109, June 1989.
18. Feng-hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess*. PhD thesis, Carnegie-Mellon University, 1990. Published as Technical Report CMU-CS-90-108.

19. Takayasu Ito and Manabu Matsui. A Parallel Lisp Language PaiLisp and its Kernel Specification. In *Parallel Lisp: Languages and Systems*, pages 58–100. Springer-Verlag, LNCS 441, 1990.
20. Suresh Jagannathan. *A Programming Language Supporting First-Class, Parallel Environments*. PhD thesis, Massachusetts Institute of Technology, December 1988. Published as LCS-Technical Report 434.
21. Suresh Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Proceedings of the Conf. on Parallel Languages and Architectures, Europe*, pages 254–276. Springer-Verlag, Springer-Verlag, LNCS 506, June 1991.
22. Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, June 1992.
23. Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 Conf. on Lisp and Functional Programming*, June 1992.
24. David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.
25. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
26. Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of the 1986 Conf. on Object-Oriented Programming, Systems, Languages and Architectures*, pages 214–223, 1986.
27. Rick Mohr. *Dynamic Partitioning of Parallel Lisp Programs*. PhD thesis, Yale University, 1991. Technical Report YALEU/DCS/RR-869.
28. Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Computing*, 2(3):264–280, July 1991.
29. J. Gregory Morrisett and Andrew Tolmach. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 198–207, 1993.
30. James Philbin. *An Operating System for Modern Languages*. PhD thesis, Dept. of Computer Science, Yale University, May 1993.
31. William Pugh and Grant Weddell. Two-Directional Record Layout for Multiple Inheritance. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 85–92, 1990.
32. Vijay Saraswat and Martin Rinard. Concurrent Constraint Programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232–246, 1990.
33. Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–60, August 1986.
34. David Ungar and Randall Smith. Self: The Power of Simplicity. In *Proceedings of the 1987 Conf. on Object-Oriented Programming, Systems, Languages and Architectures*, pages 227–241, 1987.
35. M. Vandevoorde and E. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.
36. A. Yonezawa, E Shibayama, T Takada, and Y. Honda. Object-Oriented Concurrent Programming – Modelling and Programming in an Object-Oriented Concurrent Language, ABCL/1. In *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.