LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1994

A Concurrent Abstract Interpreter

STEPHEN WEEKS (*weeks@cs.cmu.edu*)
Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA
SURESH JAGANNATHAN (*suresh@research.nj.nec.com*)
NEC Research Institute, 4 Independence Way, Princeton, NJ
JAMES PHILBIN (*philbin@research.nj.nec.com*)
NEC Research Institute, 4 Independence Way, Princeton, NJ

Keywords: Abstract Interpretation, Control-flow Analysis, Concurrency, Multi-threaded Computing

Abstract. Abstract interpretation [6] has been long regarded as a promising optimization and analysis technique for high-level languages. In this article, we describe an implementation of a *concurrent abstract interpreter*. The interpreter evaluates programs written in an expressive parallel language that supports dynamic process creation, first-class locations, list data structures and higher-order procedures. Synchronization in the input language is mediated via first-class shared locations. The analysis computes intra- and inter-thread *control* and *dataflow* information.

The interpreter is implemented on top of Sting [12], a multi-threaded dialect of Scheme that serves as a high-level operating system for modern programming languages.

1. Introduction

Abstract interpretation [6] has been long regarded as a promising optimization and analysis technique for high-level languages. However, since abstract interpretation typically involves manipulation of an approximate program state, it has been difficult to realize linear-time implementations when the approximate state is complex. On the other hand, an interpreter that manipulates a simple abstract state in which many significant details of the exact program state are collapsed may be efficient to implement, but may not be very useful as an optimization tool.

One way of making sophisticated abstract interpreters useful in practice is to transform their internal structure into a form suitable for parallelization. In such an implementation, many components of a program interpretation will occur in parallel. This implementation strategy has the potential to scale well as input program size increases; since the input to an abstract interpreter is a source program, opportunities for useful and significant concurrency increase with program size. However, parallelization of programs such as abstract interpreters would be difficult to achieve in parallel dialects of languages such as C or Fortran. This is because abstract interpreters generate data dynamically, the data sets consist of objects of irregular type and structure, and data dependencies are difficult to detect statically.

In this article, we describe an implementation of a *concurrent abstract interpreter* in a parallel dialect of Scheme. The interpreter evaluates programs written in an expressive parallel language that supports dynamic process creation, first-class locations, list data structures and higher-order procedures. Synchronization in the input language is mediated via first-class shared locations. The design of the language was motivated by our desire to have a simple framework capable of expressing many interesting parallel programming abstractions and paradigms. For example, the essential characteristics of futures [9], distributed data structures [4], concurrent objects [10], and synchronization primitives such as *replace-and-op* [2, 7] are all expressible using the primitives provided in this language. Thus, we consider a concurrent implementation of an abstract interpreter whose input language itself supports parallel constructs. In the description given here, the implementation of the concurrent interpreter is not expressed in the input language it evaluates, but there is no reason *a priori* why this could not be done.

The analysis computes intra- and inter-thread *control* and *dataflow* information. Such information can be used to facilitate a number of important optimizations that would otherwise be realized via *ad hoc* analyses; lifetime and sharing analysis, static scheduling, prefetching, test for presence, and process/data mappings, are a few important examples we have considered. Our discussion does not develop the formal construction of the abstract interpretation, details of which are provided in [14]. The ability to effectively analyze high-level parallel languages statically has important implications for implementing high-performance parallel systems well-suited for symbolic processing.

Broadly speaking, we can regard the abstract interpreter as a constraint based evaluator. The abstract state is represented as a directed graph in which nodes correspond to expressions and edges represent the flow of data. Each node stores an abstract value which corresponds to the value of the expression at that node. Each edge can be viewed as a (subset) constraint on the abstract values stored at the nodes it connects. The evaluation of an expression can change the abstract value stored at a node, which in turn may violate a number of constraints (i.e., edges) emanating from that node. This may cause the evaluation of further expressions, the violation of further constraints, etc. as a further complication, because the input language contains data structures and higher-order procedures, edges may be added dynamically. The interpreter terminates when all constraints are satisfied.

Despite its complexity, structuring an interpreter in this way has the important benefit of making the interpreter amenable to a parallel implementation. Because the interpreter is organized in terms of a collection of local rules (constraints between pairs of nodes), many of its components may be able to evaluate concurrently.

One obvious concurrent implementation of the interpreter would create a new lightweight thread for each constraint that is violated; thus, the resulting runtime behavior would be very fine-grained, especially if little computation is required per node. We rejected this implementation, however, because it is not amenable to runtime optimizations such as lazy task creation [18], or thread absorption [12] that can be effectively used in certain instances to obviate the cost of fine-grained parallelism on stock multiprocessor platforms. These optimizations reduce the overhead of creating many execution contexts for fine-grained tasks by allowing threads that exhibit manifest data dependencies with one another to share the same execution context (i.e., stack, registers, etc.); these techniques thus effectively increase thread granularity transparently at runtime.

The threads generated by a fine-grained concurrent implementation of the concurrent interpreter, however, will often not exhibit any manifest data dependencies with any other thread. Because not all dependencies in the graph are known statically, and because constraints generated by one thread must often be propagated to many nodes in the abstract state, a fine-grained concurrent implementation of the interpreter will typically entail the creation of many more execution contexts than can be efficiently implemented on multiprocessor platforms that do not provide explicit hardware support for fine-grained concurrency [1, 11].

We, therefore, choose a more coarse-grained implementation strategy. In this version, the abstract state is partitioned, and a thread is assigned to manage each partition. The runtime organization of a program's abstract state is given in terms of shared data structures that are concurrently updated by these threads as new abstract values are generated and refined. This implementation places significant communication and memory management burdens on Sting[12, 13], the runtime kernel on which the interpreter is implemented, and serve to highlight a number of interesting issues in programming dynamic and irregular parallel computations.

The remainder of this article is organized as follows. Section 2 describes the target language for the interpreter. Section 3 introduces the data and control-flow problem in this context. Section 4 gives a brief description of the exact and abstract semantics for a kernel language we develop. Section 5 describes the rules used by the interpreter to implement the abstract semantics. Sections 6 discusses a sequential implementation. Section 7

$$
\begin{array}{lcl}
e & \in & \mathit{Exp} \\
c & \in & \mathit{Const} = \mathit{Int} + \mathit{Bool} \\
x & \in & \mathit{Var} \\
f & \in & \mathit{Func} \\
p & \in & \mathit{Prim} = \{\mathbf{cons}, \mathbf{car}, \mathbf{cdr}, \mathbf{mk\text{-}loc}, \mathbf{write}, \mathbf{read}, \mathbf{remove}, \ldots\} \\
\\
e & ::= & c \\
 & | & x \\
 & | & f \\
 & | & p \\
 & | & e_1\,(e_2,\ e_3,\ \ldots,\ e_n) \mid (e) \\
 & | & \mathbf{if}\, e\ \mathbf{then}\, e_1\ \mathbf{else}\, e_2 \\
 & | & \mathbf{letrec}\ x_1 = f_1\ \ldots\ x_n = f_n\ \mathbf{in}\ e \\
 & | & \mathbf{spawn}\, e \\
\\
f & ::= & \lambda x_1\ x_2\ \ldots\ x_n.\, e
\end{array}
$$

Figure 1: The kernel language.

describes a parallel extension to the sequential implementation. Section 8 presents some benchmark timings, and Section 9 gives conclusions.

2. The Target Language

Our abstract interpreter is concerned with computing control-flow and dataflow information for an expressive higher-order parallel language. Our language (see Fig. 1) has constants, variables, procedures, primitives, call-by-value procedure applications, conditionals, mutual recursion, and a process creation operation. The primitives include operations for creating and accessing pairs and shared locations. Constants include integers and booleans.

Since processes can be instantiated dynamically, and since synchronization is mediated via shared locations, this language can be used to describe a number of interesting synchronization and communication paradigms.

The language's sequential core defines a higher-order functional language with list data structures; its parallel extensions provide functionality found in parallel Lisps [9, 16], ML threads [19], or Linda [3]. Concurrency is introduced through a **spawn** operation that creates a lightweight thread to evaluate its input argument. Synchronization and communication is realized exclusively through shared variables. There are four operations over shared locations:

1. (**mk-loc**) creates a new location and marks it unbound.
2. **read**(x) returns the value at location x, blocking if x is empty.
3. **write**(x, v) writes v into the location denoted by x.
4. **remove**(x) behaves like **read**, except that in addition to reading the value stored at location x, it also marks x as unbound.

To illustrate the language's utility, we show the following translation of the MultiLisp [9] expression, (**future** e)

let *loc* = (**mk-loc**)
in begin
    spawn write(*loc*, e)
    loc
  end

The MultiLisp synchronization operation (**touch** v) is equivalent to,

if location? (v)
  then read(v)
  else v

(Note that "**let**" and "**begin**" are syntactic sugar for simple and nested application, respectively.)

3. Control and DataFlow Analysis

Optimizing compilers for high-level sequential languages typically perform aggressive control and dataflow analysis which can be used to facilitate a number of important optimizations; notable examples include lifetime and escape analysis, type recovery [23, 24], efficient closure representation [17, 22], constant folding, and copy propagation.

In languages such as Scheme or ML, however, inter-procedural analysis is complicated. The difficulties arise primarily because of higher-order procedures and polymorphic data structures. Consider the example shown in Fig. 2. A useful control-flow analysis of this program would reveal a number of interesting properties about this program fragment:

$$
\begin{array}{l}
\textbf{let } f = \ \lambda\ (x\ y) \\
\qquad\qquad \textbf{if null? } x \\
\qquad\qquad\quad \textbf{then } \lambda\ ()\ y \\
\qquad\qquad\quad \textbf{else let } z = (x) \\
\qquad\qquad\qquad\quad \textbf{in} \quad \ldots\ z \ldots \\
\textbf{in let } g = \textbf{cons } (\textbf{let } v = \mathit{exp} \textbf{ in } \ \lambda\ ()\ f\ (\textbf{nil } v), \\
\qquad\qquad\qquad\quad \textbf{nil}) \\
\quad \textbf{in } \ f\ (\textbf{car}(g),\ w)
\end{array}
$$

Figure 2: Higher-order programs have non-trivial control-flow and dataflow properties.

1. v escapes outside the call in which it occurs.

2. the closure passed to **cons** is applied in f's body indirectly via **car**.

3. the application of x in the conditional's **else** branch returns a closure corresponding to λ `()` y where the binding for y is either the value of *exp* or the value of w.

4. x is bound to **nil** or the procedure,

   $\lambda\ ()\ f\ (\textbf{nil},\ v)$

To obtain this kind of information, a control-flow analysis must be capable of selectively collapsing environment information associated with multiple instantiations of the same procedure, and recording dataflow into and out of complex data structures such as lists.

In the presence of concurrency, the control-flow analysis problem becomes exacerbated. In addition to recording useful closure information, an analysis is also faced with the task of computing dynamic interleaving information among threads. In order to be practical, however, the interpreter cannot afford to generate *all* potential interleavings; certain interleaving information must be collapsed. Furthermore, an analysis that operates over a language of the kind described in the previous section must also track the movement of shared locations since optimizations relating to synchronization and communication depend on knowing how and where these locations are used.

4. Semantics

A detailed description of the exact and approximate semantics for our kernel language is given in [14]; we provide a brief outline here.

Both the exact and abstract semantics are defined in terms of labels. There are three kinds of labels: *value*, *environment*, and *spawn*, denoted by θ_v, θ_e, and θ_s, respectively. Every expression in the program is assigned a unique value label, which is used to store the value of the expression. Every lambda and spawn expression is assigned a unique environment label. Additionally, each spawn expression is assigned a unique spawn label. In the exact semantics, we say an expression is *being evaluated in spawn label* θ_s if the thread which is evaluating the expression was created by the spawn expression labeled with θ_s. Note that this is a dynamic notion, not a syntactic one.

In the abstract semantics, a pair $\langle \theta_v, \theta_s \rangle$ serves as an index into the abstract state. Associated with this index is an abstract value that "combines" or "joins" all values to which the expression with label θ_v may evaluate in spawn label θ_s. Similarly, a pair $\langle \theta_e, \theta_s \rangle$ is an index to an abstract environment that "combines" all environments which may exist at label θ_e while evaluating in spawn label θ_s.

In both the exact and abstract semantics, labels serve a dual role as components of values. In the exact semantics, pointers and heap entries are created for cons cells, shared locations and closures. Similar "abstract pointers" are created in the abstract semantics. For example, an "abstract pointer" to a cons cell is represented by a pair $\langle \theta_v, \theta_s \rangle$, where θ_v is the label of the cons expression and θ_s is the label of the thread which created the cell. Intuitively, this abstraction joins the values of all cons cells which were created by the cons expression labeled with θ_v while being evaluated in θ_s.

An abstract value is simply a set of abstract pointers. Abstract environments are joined componentwise; in other words, an abstract environment is a function from variables to abstract values.

5. Rules

An abstract state consists of two maps, one from pairs $\langle \theta_v, \theta_s \rangle$ to abstract values and one from pairs $\langle \theta_e, \theta_s \rangle$ to abstract environments. Because an abstract environment is a function from variables to abstract values, we can generalize further and view an abstract state as single map from indices to abstract values, where an *index* is either a pair $\langle \theta_v, \theta_s \rangle$ or a triple $\langle \theta_e, \theta_s, x \rangle$; x represents a program variable in the latter case.

The abstract semantics is defined by a transition function, T, that maps abstract states to abstract states. The abstract transition function for a particular program is given by a collection of rules, one for each index, where the rule for a particular index i takes an abstract state and describes how to compute the abstract value that arises at i after taking one "step" in the program. There are fairly complex rule schemas which describe how to define the rule for each index for a particular program; [14] provides details.

We use the metavariable i to range over indices and S to range over abstract states. We use $S(i)$ to denote the abstract value at index i in abstract state S. Finally, viewing a rule as a function from abstract states to abstract values, we use $R_i(S)$ to indicate the abstract value at index i after taking a step in S. For the purposes of this article, the details of the rules are unimportant. The rules can, however, be broadly classified into one of four categories; each rule is given as union over some set of values in the abstract state:

1. **Constant.** Constant rules are used for indices whose values are syntactically apparent. Indices which correspond to numbers, cons expressions, **mk-loc** expressions, and lambda expressions fall in this category. Note that for the latter three, a singleton set containing an abstract pointer is the value.

2. **Static.** Static rules are used to compute values where the control flow is apparent syntactically. For example, the value of an if-expression is the join (union) of the values of the consequent and alternative branches, whose indexes are syntactically apparent. A similar type of rule describes the value of an expression which references a variable since that variable is stored in the enclosing abstract environment.

3. **Dynamic From Pointers.** These rules describe information that flows from a pointer when it arrives at a particular index. For example, the value of the index of a **car** expression depends on the cons pointers that constitute the value of its subexpression. A similar rule describes **cdr** expressions and **read** expressions on locations.

4. **Dynamic To Pointers.** These rules describe information that flows back to a pointer based on the indices whose value contain the pointer. For example, the values that flow into an abstract location depend on the set of indices corresponding to expressions that write to that location.

$S_{new} \leftarrow S_0$
repeat
  $S_{old} \leftarrow S_{new}$
  $S_{new} \leftarrow T(S_{old})$
until $S_{old} = S_{new}$

Figure 3: Naive Algorithm

6. Implementation

Abstract values are ordered by set inclusion, abstract states are ordered pointwise. If S_0 abstracts the set of initial program states, then the least fixed point we seek is the limit of the chain:

$$S_0, T(S_0), T^2(S_0), \ldots T^n(S_0), \ldots$$

The least fixed point may be computed by the algorithm in Figure 3.

This algorithm has a straightforward implementation. At each iteration of the loop, this implementation (1) creates a new state, (2) computes the value at each index of the new state as given by the rule at that index, and (3) tests for equality between the new state and the previous state. Unfortunately, this implementation suffers from many inefficiencies. First, it requires the construction of a new state for each iteration of the loop. It would be more efficient to keep a single state, and update it destructively. Second, there may be many indices i such that $S_{new}(i) = S_{old}(i)$; e.g., the abstract values produced by constant and static rules are invariant across iterations in the fixpoint computation. We would like to eliminate the unnecessary recomputation of these unchanged values, and only compute the entries which might change. Third, the test for equality requires comparing each entry of the new state with the corresponding entry in the old state (a set comparison). As before, we would like to remove as much redundant computation as possible from this test, and only compare entries that may have changed.

These concerns can be addressed by an implementation based on a different view of the abstract transition function. Instead of viewing this function as defining a set of rules that describe how to compute entries of a new abstract state, we can view the rules as *constraints* on the current abstract state that must be satisfied. Let R_i be the rule that exists at index i in a program. By the definition of T, there there exists an index i in abstract state S such that $S(i) \neq R_i(S)$, if and only if $S \neq T(S)$. The algorithm in figure 4 is based on explicitly maintaining a set of such indices.

$S \leftarrow S_0$
$I \leftarrow I_0$
while $I \neq \phi$
  remove some index i from I
  if $S(i) \neq R_i(S)$
    then $S(i) \leftarrow R_i(S)$
      $I \leftarrow I \cup D_i$

Figure 4: Constraint Algorithm

It maintains the invariant that at the beginning of each iteration, the index set I contains all indices i such that $S(i) \neq R_i(S)$.

For each iteration, the algorithm chooses an index i from I, computes $R_i(S)$, and if new information is found, updates S to reflect the new information. This modification of the state may in turn violate other constraints, and new indices may need to be added to I to maintain the invariant. An efficient implementation of this algorithm requires being able to determine which indices must be recomputed when $S(i)$ changes. The simplest solution is to associate this information with each index. Let D_i denote the set of indices that depend on $S(i)$. Initially, the D_i's are given by the static rules of T. As the fixed point is computed, and abstract values arrive at certain indices, the D_i's are augmented to reflect additional dependencies induced by the dynamic rules of T. To do so requires that a small amount of additional information be stored for certain abstract pointers.

When the algorithm terminates, we have for all indices i that $S(i) = R_i(S)$, hence $S = T(S)$. It can also be proven by induction that at each iteration, S is less than or equal to the least fixed point of T. The proof relies on the monotonicity of T, which is evident from its definition.

One inefficiency of the constraint algorithm is that for each $i' \in D_i$, the entire union as given by $R_{i'}$ will be recomputed upon changing $S(i)$. Much of this computation is, in fact, redundant, and can also be removed. By the monotonicity of T we know that any change to an abstract state S at index i must be an increase. Hence, we only need to compute the effect of the *new* information on the union. To be more precise, let P be a set that is to be added to S at index i, and let P_{diff} be $S(i) - P$. The only information to be propagated to indices i' which depend on i is P_{diff}; it is not necessary to recompute $R_{i'}$. The algorithm in Fig. 5 explicitly maintains a list of pairs $\langle i', P \rangle$ where P is a set containing new information that must be added to the union at $S(i')$. L_0 contains sets of pairs associated with indices that

$S \leftarrow S_0$
$L \leftarrow L_0$
while $L \neq \phi$
  remove some pair $\langle i, P \rangle$ from L

  let P_{diff} be $P - S(i)$
  in if $P_{\mathit{diff}} \neq \phi$
    then $S(i) \leftarrow S(i) \cup P_{\mathit{diff}}$
      for each $i' \in D_i$
        add $\langle i', P_{\mathit{diff}} \rangle$ to L
end

Figure 5: Dependency Algorithm

have static or constant rules.

Essentially, this algorithm views the abstract state and the rules as a graph, where the nodes are indices, and the adjacency list for node i is given by D_i. An edge (i, i') represents the fact that information flows from index i to i'. Thus, if there is an edge from i to i', then in a fixed point of T, $S(i) \subseteq S(i')$. Whenever new information is added to a node, this new information is propagated along all outgoing edges of the node. The algorithm is similar in spirit to the "worklist" approach of Hall and Kennedy [8].

7. The Parallel Interpreter

In the following, we use *thread* to refer to a lightweight asynchronous process defined as part of the interpreter implementation.

We parallelize the dependency algorithm of the previous section by partitioning the abstract state. Thus, we define a function $\pi : \mathit{Index} \rightarrow \{1, 2, \ldots, K\}$, where K is the number of partitions. For each partition, k, a worker thread W_k is created which is responsible for computing $S(i)$ for all i such that $\pi(i) = k$. Worker W_k is also responsible for communicating any new information it computes to other partitions, as dictated by its outgoing dependency edges. The program structure resembles a master-slave computation. The master is described in Fig. 6.

Each worker thread has a both a local queue (L) and a remote queue (R) of $\langle i, P \rangle$ pairs that contain additional information that must be joined to nodes in its partition. The structure of a worker is very similar to the

$$
\begin{array}{l}
S \leftarrow S_0 \\
\text{for } k = 1 \text{ to } K \\
\quad L_k \leftarrow L_{k0} \\
\quad R_k \leftarrow \phi \\
\\
\text{for } k = 1 \text{ to } K \\
\quad \text{create worker } W_k
\end{array}
$$

Figure 6: Master

$$
\begin{array}{l}
\text{while } L_k \neq \phi \text{ or } R_k \neq \phi \\
\quad \text{let } \langle i, P \rangle \text{ be} \\
\qquad\qquad \text{if } L_k \neq \phi \\
\qquad\qquad\quad \text{then } \mathit{dequeue}(L) \\
\qquad\qquad\quad \text{else } \mathit{dequeue}(R)) \\
\qquad P_{\mathit{diff}} \text{ be } P - S(i) \\
\quad \text{in if } P_{\mathit{diff}} \neq \phi \\
\qquad \text{then } S(i) \leftarrow S(i) \cup P_{\mathit{diff}} \\
\qquad\qquad\quad \text{for each } i' \in D_i \\
\qquad\qquad\qquad \text{if } \pi(i') = k \\
\qquad\qquad\qquad\quad \text{then } \mathit{enqueue}(L_k, \langle i', P_{\mathit{diff}} \rangle) \\
\qquad\qquad\qquad\quad \text{else } \mathit{enqueue}(R_{\pi(i')}, \langle i', P_{\mathit{diff}} \rangle) \\
\text{end}
\end{array}
$$

Figure 7: Code for Worker k

dependency algorithm of the previous section. When new information is found, however, it must be added to the appropriate worker's queue. The code for a worker is shown in Fig. 7.

Partitioning the input across threads reduces contention on shared data structures. Since each thread has exclusive access to the portion of the abstract state corresponding to its partition, no synchronization operations need to be executed by threads when refining or adding a local constraint. Synchronization costs occur entirely in the management of remote queues. A thread sending work to a remote queue belonging to another thread must have exclusive access to the tail of the queue in the interval in which the message transmission occurs. Work sent to remote threads are implemented as thunks that are evaluated by the receiving thread upon receipt; this communication paradigm resembles *active messages* [25].

In our implementation, work found on local queues is done before work found on remote ones. A thread examines its remote queue only when all local work has been completed. Evaluating work found on a remote queue may enable further (local) updates on a partition; these are evaluated before the next item on the remote queue is examined. If there is no work on either the local or remote queues, the thread either spins or blocks; a blocked thread W is resumed by another thread W' when W' deposits a new piece of work on W's remote queue. There is one top-level thread T that remains blocked and is resumed only when the program reaches a fixpoint. This happens when all workers become blocked because their local and remote queues are empty. Prior to blocking, a worker increments a counter that is decremented when the thread resumes; when the counter becomes equal to the number of threads initially created, T is resumed. T then terminates all workers and returns the current abstract state.

The parallel program structure in this problem exhibits a great deal of locality – shared data is exclusively owned by a thread (data and process layout is thus SPMD-style), communication occurs exclusively through messages sent on remote queues, and it is assumed that there is a one-to-one correspondence between processors and threads. Depending on the input program and the effectiveness of the partitioning strategy, however, there may be significant communication via remote queues among different tasks. The overheads incurred in message transmission, blocking, and resuming threads is determined partially by the efficiency of the Sting implementation, and partially by the granularity exhibited by the interpreter's sequential component.

8. Benchmarks

Our benchmarks were implemented on a 16 processor Silicon-Graphics cache-coherent shared memory multiprocessor. Each node consists of a MIPS R4400 150 Mhz processor; the total memory on the system is 512 megabytes. Each processor has a 16Kbyte data and 16Kbyte instruction on-chip primary cache, and a one megabyte unified off-chip secondary cache. The operating system used is Irix V Release 5.1.1.1.

The benchmarks were executed using compiled versions of both Sting and the abstract interpreter. The definition of the interpreter follows the description of the abstract semantics given in Sections 4, 5 and 7.

8.1. Input Programs

We benchmarked two different applications on the concurrent abstract interpreter. The first is a parallel parser implemented for a significant subset of Scheme [5] written in a functional style, adapted from [20]. In this code, a parser is a function that takes a stream of tokens and a location and writes the resulting parse tree in that location. There are *parser constructors* for the grammar operations "or" (*alt*), "concatenation"' (*seq*), and "option" (*optional*). The only parallelization of the parser is in the function *alt*, the parser constructor that implements the *or* grammar operation. It takes two parsers and an input stream and returns the resulting parse tree if either parser succeeds. In the parallel case, threads are spawned to evaluate both parsers. When the first thread is complete, its value is returned, if it succeeds; if it fails, the value of the second parser is returned. Note that even if the first thread completes successfully, the evaluation of the second thread is not aborted; the value returned by the second thread is simply ignored.

The second benchmark, self-application, is shown in Fig. 8. Since a significant fraction of the interpreter's complexity lies in managing higher-order procedures, and propagating new information derived by a function application to other call sites of the applied function, we constructed a benchmark to exercise these facets. While many higher-order parallel programs may not exhibit this kind of structure, we expect more complicated abstract interpreters to exhibit many of the same characteristics shown by our interpreter on this benchmark on demonstrably simpler applications [23].

8.2. Parameters

The benchmarks shown in the following sections measure the time taken to run the abstract interpreter on the input programs described above. We parameterized the structure of the interpreter along three dimensions:

1. *Partitioning Strategy.* We consider two possible ways of allocating indices in the abstract state to threads. The first is *random* allocation; in this strategy, the set of nodes managed by a thread is determined randomly. The second is *block* allocation; in this strategy, a node is allocated to a given partition if it contains a known dependency with some other node in that partition, provided that its inclusion would not cause the number of elements in the partition to exceed the partition's maximum allowed size.

   The random partition strategy suffers from poor locality effects – nodes adjacent in the dependency graph may be allocated to different threads running on different processors. It does have the advantage,

$$
\begin{array}{l}
\textbf{let } \mathit{thunk} \;=\; \lambda\,().\\
\qquad\qquad\qquad \begin{array}{l}
\textbf{let } p_1 \;=\; \lambda\, f.f(\,f)\\
\quad\;\; p_2 \;=\; \lambda\, f.f(\,f)\\
\quad\;\; \vdots\\
\textbf{in let } i_1 \;=\; \lambda\, x.x\\
\qquad\;\;\; i_2 \;=\; \lambda\, x.x\\
\qquad\;\;\; \vdots\\
\quad \textbf{in } p_1(\,i_1)\\
\qquad\; p_1(\,i_2)\\
\qquad\; \vdots\\
\qquad\; p_2(\,i_1)\\
\qquad\; p_2(\,i_2)\\
\qquad\; \vdots
\end{array}\\
\textbf{in spawn}(\,\mathit{thunk})\\
\quad \textbf{spawn}(\,\mathit{thunk})\\
\quad \vdots
\end{array}
$$

Figure 8: A complex higher-order program. The benchmark used evaluated a program with the above structure using 20 self-application nodes, 20 identity procedures, and 20 syntatically distinct spawn expressions.

however, of being less susceptible to poor load balancing; a more even partition of work to threads is likely to occur using random partitioning for abstract states that have irregular or non-uniform structure.

The block partition strategy exploits locality in the dependency graph, at the expense of uneven load balancing for non-uniform, irregularly structured graphs.

2. *Waiting Strategy.* Recall that a thread with no work in its local queue examines its remote one. In the case where the remote queue is also empty, a thread may either choose to busy wait or explicitly block. Busy waiting is obviously the preferred alternative when the number of evaluating threads is equivalent to the number of processors being used since no useful work will be performed by a processor executing a thread that chooses to block. Blocking, on the other hand, is conceptually cleaner and more general since it decouples the choice of how many threads (or partitions) are created from the number of processors actually used.

3. *Traversal Strategy.* Nodes may be examined using either a breadth-first or a depth-first traversal strategy. In a breadth-first implementation, a thread deposits all new work into local and remote queues before examining the contents of these queues. As a result, the order in which information is added to the graph is unpredictable. In the depth-first case, a new piece of work is immediately evaluated before any other; thus, there is no need for local queues, and a stack is sufficient. In this case, information percolates along a given spine in the graph until it crosses a partition boundary or it encounters a node that does not generate any new information.

8.3. Execution Times

8.3.1. Sequential Times

As a base case, we measured the time taken by the abstract interpreter to evaluate the input programs in a purely sequential setting. We compiled the interpreter using Orbit [17]; the interpreter was written in T [21] release 3.1. The sequential benchmarks uses no partitioning or waiting strategy; it is a faithful implementation of the algorithm shown in Fig. 5. We measure the times using both breadth-first and depth-first traversal. In a sequential implementation, the traversal strategy only dictates the order in which nodes in the dependency graph are examined; there are no remote queues manipulated. Since our intent was to measure only sequential running

	Parser	Self-Apply
T	56.5 (BFS)	100.9 (BFS)
	66.4 (DFS)	74.3 (DFS)

Figure 9: Sequential times (in seconds) for the abstract interpreter written in T.

times, no part of the Sting runtime was involved in these tests. The times are shown in Fig. 9 and do not include garbage collection costs.

8.3.2. Parallel Execution

For the parallel implementation, we used the Sting runtime system to handle all concerns related to thread scheduling and context-switching. The Sting virtual machine used for the benchmarks implemented a single global FIFO thread queue. None of the benchmarks triggered garbage collection.

The benchmark results are shown in Fig. 10. The number of block operations executed in runs using a block waiting strategy is shown in Fig. 11. Note that the single processor times for both the parser and self-apply are slightly faster than the sequential times; memory and register optimizations performed by Sting that are not utilized by T are the primary reasons for this difference. As a related point, lock acquire and releases on the R4400 are implemented in terms of two simple machines instructions, `load-linked` and `store-conditional` [15]; the overhead in acquiring locks in the single processor case is thus negligible.

With 16 processors, using a breadth-first traversal, block partition, and spin-waiting, the evaluation of the abstract interpretation on the parser took 7.1 seconds; compared with the single processor time of 53.8 seconds, this program executes at roughly 47% efficiency; on 8 processors, the same program executes with 58% efficiency.[1] We suspect the drop in efficiency is due to lack of available parallelism in the program.

Note however that the block partition strategy clearly outperforms random partitioning which achieved no speedup regardless of the traversal or waiting strategy used. This is because updating nodes across partition boundaries entails manipulating remote queues which are serialized. More importantly, locality is compromised; on tightly-coupled cache-coherent shared memory machines, failure to achieve cache locality can lead to significant performance degradation. With random partitioning, a thread will likely perform many updates to shared data structures not cached on its

[1] If m is the running time of a program P on one processor, and n is its execution time on k processors, the efficency of P on k processors is defined to be $\frac{\frac{m}{k}}{n} \times 100$.

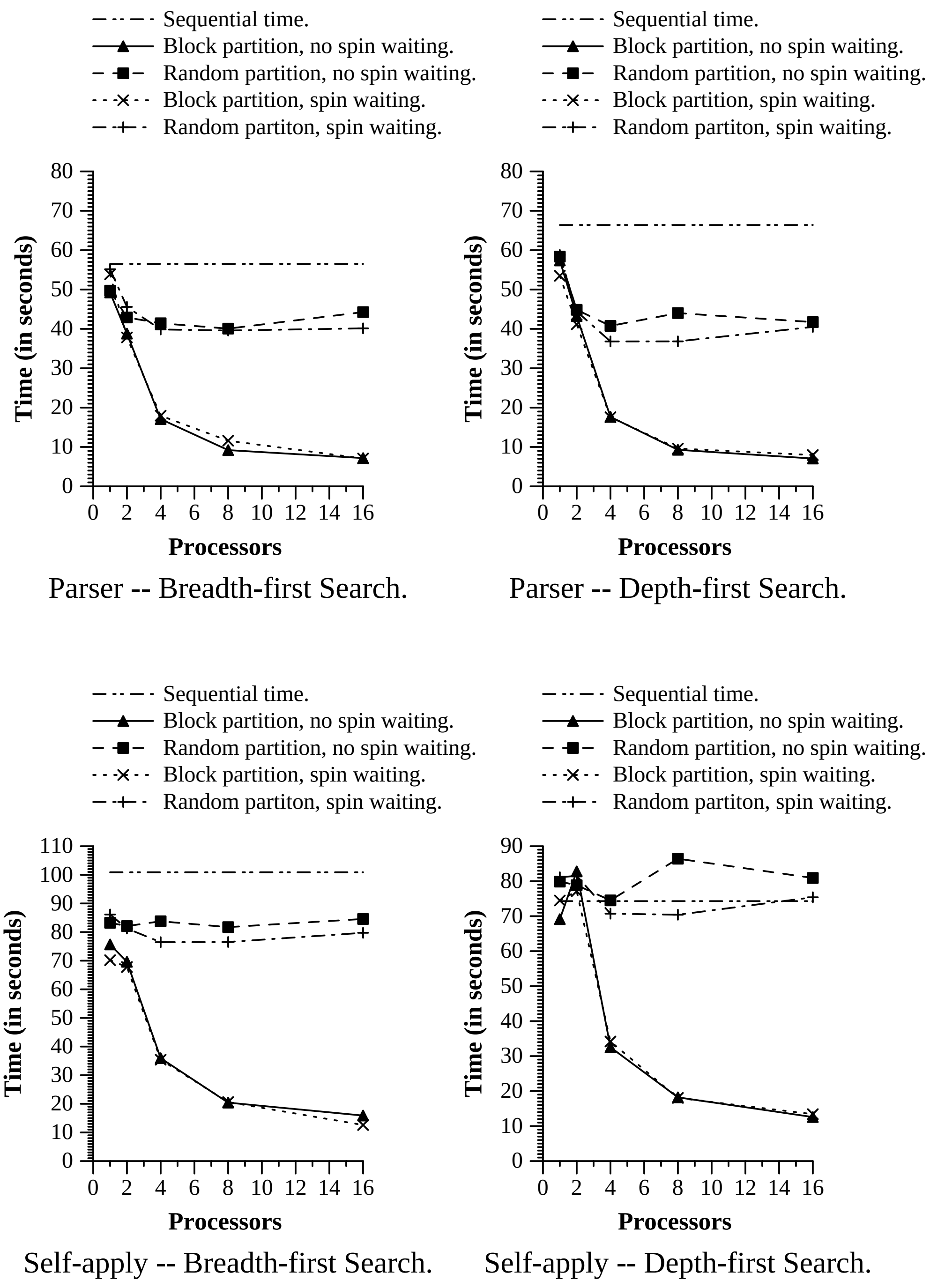
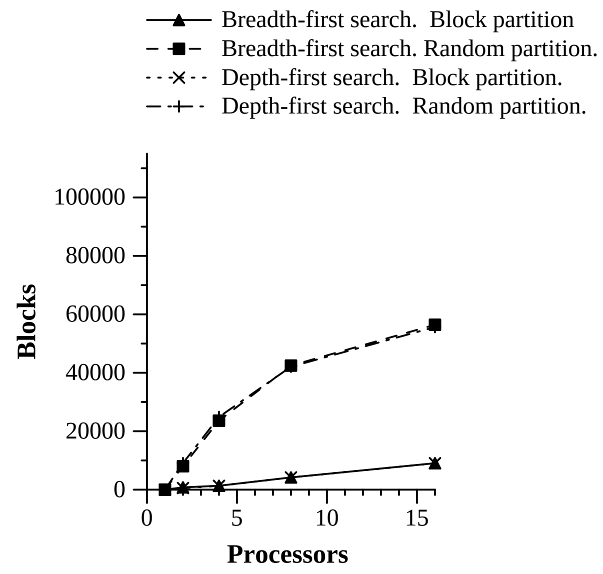
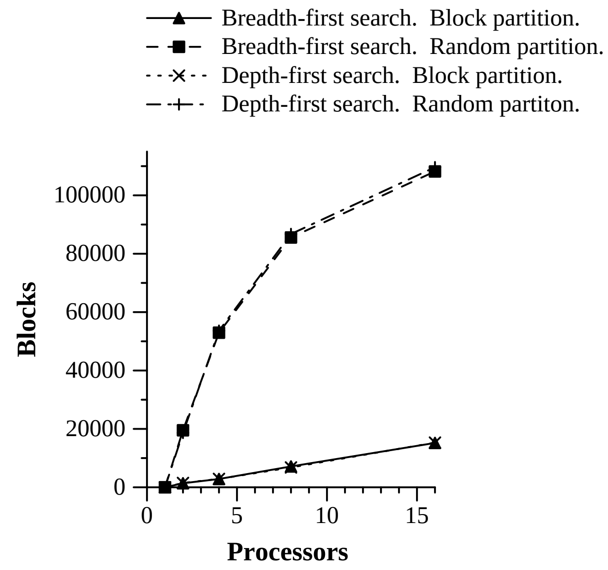


Figure 10: Wallclock times measuring the execution of the concurrent abstract interpreter on the input programs, Parser and Self-apply described in Section 8.1.



Parser Block Counts.



Self-Apply Block Counts.

Figure 11: Thread block counts incurred under different partitioning and traversal strategies for the input programs described in Section 8.1 when evaluated by the concurrent abstract interpreter.

processor. Poor memory locality coupled with extra synchronization costs make random partitioning an inferior strategy to block partitioning.

With block partitioning, the choice of spin waiting or blocking appears to have little impact on the overall execution times. Under a breadth-first traversal with automatic blocking, running the concurrent interpreter on the parser resulted in approximately 9,000 thread blocks/unblocks. The cost of executing these extra blocks is roughly the cost of simply spinning; the overheads introduced by Sting to resume blocked threads does not appear to entail any performance penalty; in fact, on 4 and 8 processors, a blocking waiting strategy using block partitioning slightly outperforms the spin waiting version.

Under a random partition strategy, the block counts increase dramatically. On 16 processors, using breadth-first traversal interpretation of the parser resulted in roughly 56,400 thread blocks. A similar number of blocks occurs under a depth-first traversal.

Surprisingly, there is little difference between a depth-first or a breadth-first traversal of the abstract state. With 16 processors, using block partitioning and without busy waiting, interpreting the parser using a depth-first traversal took 7.06 seconds versus 7.17 seconds using a breadth-first strategy. With spin-waiting, the breadth-first version took 7.07 seconds versus 7.9 seconds under a depth-first strategy. We suspect the dependency graph for the parser has small depth but is bushy; in this case, its evaluation takes little advantage of locality benefits afforded by a depth-first traversal.

The performance curves for the self-application benchmark are roughly the same as the parser. As with the parser, the dominant parameter in the execution times is the partitioning strategy; self-apply on 16 processors with a block partition scheme takes between 13.4 seconds (with depth-first search and spin-waiting) and 13.6 seconds (with breadth-first search and spin-waiting). The execution times with a random partition is roughly constant regardless of the number of processors used. Given a single processor time of 74.4 seconds with depth-first search, block partitioning and spin-waiting, the interpreter executes self-apply with an efficiency of approximately 34% on 16 processors; on 8 processors the efficiency improves to a little over 51%. Again, the reason for the dropoff is presumably lack of available parallelism; a larger input program would presumably improve the overall efficiency provided garbage collection remains relatively infrequent.

As with the parser, self-apply shows the importance of spatial locality on tightly-coupled parallel systems. A random partitioning strategy exhibits poor locality that results in a significantly larger number of thread blocks; under a depth-first traversal, applying the interpreter to self-apply on 16 processors resulted in over 109,000 thread blocks, roughly a factor of seven more than the number of blocks executed using block partitioning. Here also, the traversal strategy chosen seems to be less important than the partitioning strategy. On 16 processors, a depth-first traversal under a block partition with busy waiting takes 13.41 seconds compared with 13.59 seconds under a similar configuration with a breadth-first traversal. The non-busy waiting numbers are roughly the same. The structure of the dependency graph for self-apply again presumably lacks sufficient height to highlight differences between the two traversal strategies.

Regardless of the underlying machine, communication costs incurred in depositing work on remote queues, lack of "parallel slackness", and uneven load balancing appear to be dominating factors as we scale the number of processors for both benchmarks. Significant queue contention and poor cache locality are important factors in making remote queue access costly relative to the cost of executing the interpreter's serial component. Moreover, the small sequential grain size defined by the local unit of work performed by each thread is small enough to exaggerate the costs of inter-thread communication. In both benchmarks, threads communicate frequently, and perform relatively little work compared to the cost of recording a piece of work on another thread's remote queue. Increasing grain size by computing a more sophisticated analysis, or combining other analyses simultaneously would help mask some of these overheads. Furthermore, regardless of whether a thread blocks or spins, the lack of other tasks to execute when a thread has no work in either queue undoubtedly results in some performance degradation. Unfortunately, simply making the partition sizes smaller, and thus the program more fine-grained will not necessarily improve performance because contention for remote queues would increase correspondingly, thus eliminating any benefits derived from extra available parallelism. Allowing nodes to dynamically migrate, and thus permitting dynamic repartitioning of the abstract state would be a more realistic solution. Since not all nodes perform equal amount of work, using static (and even random partitions) is unlikely to lead to even load-balancing among worker threads. Allocating threads to "interesting" regions of the abstract state may lead to a more efficient implementation.

9. Conclusions

An efficient concurrent abstract interpreter is a challenging and important symbolic application. Like many other applications in this category, abstract interpreters generate data dynamically, work on inputs with highly irregular structure, manipulate objects of many different types, and do not lend themselves to a trivial parallel formulation. Constructing an efficient concurrent implementation, however, has several important benefits. Most significantly, it enables the incorporation of a number of sophisticated inter-procedural compile-time optimizations as part of a general compiler toolbox for Scheme and other higher-order symbolic programming languages.

References

1. Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th IEEE Conference on Computer Architecture*, pages 104–114, 1990.

2. George Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, 1991.

3. Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.

4. Nick Carriero and David Gelernter. Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler. In *Second Workshop on Languages and Compilers for Parallelism.* MIT Press, August 1989.

5. William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

6. Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *ACM 4th Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

7. Allan Gottlieb, B. Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

8. Mary Hall and Ken Kennedy. Efficient Call Graph Analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.

9. Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

10. Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

11. William J. Dally and *et. al.* Architecture of a Message-Driven Processor. In *Proceedings of the 14^{th} IEEE Conference on Computer Architecture*, pages 189–196, 1987.

12. Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 55–67, June 1992.

13. 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*, pages 345–357, June 1992.

14. Suresh Jagannathan and Stephen Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994.

15. G. Kane. *MIPS RISC Architecture.* Prentice-Hall, 1989.

16. 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.

17. David Kranz, R. Kelsey, Jonathan Rees, Paul Hudak, J. Philbin, and N. Adams. ORBIT: An Optimizing Compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986.

18. Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.

19. 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.

20. Chris Reade. *Elements of Functional Programming.* Addison Wesley Press, 1989.

21. Jonathan A. Rees and Norman I. Adams. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

22. Zhong Shao and Andrew Appel. Space Efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994.

23. Olin Shivers. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation.* MIT Press, 1990.

24. Olin Shivers. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, 1991.

25. Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Eric Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19^{th} Annual International Symposium on Computer Architecture*, pages 256–266, 1992.