

A Provably Time-Efficient Parallel Implementation of Full Speculation

JOHN GREINER

Rice University

and

GUY E. BLELLOCH

Carnegie Mellon University

Speculative evaluation, including leniency and futures, is often used to produce high degrees of parallelism. Understanding the performance characteristics of such evaluation, however, requires having a detailed understanding of the implementation. For example, the particular implementation technique used to suspend and reactivate threads can have an asymptotic effect on performance. With the goal of giving the users some understanding of performance without requiring them to understand the implementation, we present a provable implementation bound for a language based on speculative evaluation. The idea is (1) to supply the users with a semantics for a language that defines abstract costs for measuring or analyzing the performance of computations, (2) to supply the users with a mapping of these costs onto runtimes on various machine models, and (3) to describe an implementation strategy of the language and prove that it meets these mappings. For this purpose we consider a simple language based on speculative evaluation. For every computation, the semantics of the language returns a directed acyclic graph (DAG) in which each node represents a unit of computation, and each edge represents a dependence. We then describe an implementation strategy of the language and show that any computation with w work (the number of nodes in the DAG) and d depth (the length of the longest path in the DAG) will run on a p -processor PRAM in $O(w/p + d \log p)$ time. The bounds are work efficient (within a constant factor of linear speedup) when there is sufficient parallelism, $w/d \geq p \log p$. These are the first time bounds we know of for languages with speculative evaluation. The main challenge is in parallelizing the necessary queuing operations on suspended threads.

Categories and Subject Descriptors: D.2.8 [Software Engineering]: Metrics; D.3.2 [Programming Languages]: Language Classifications—*Data-flow languages*; F.1.2 [Theory of Computation]: Modes of Computation—*Parallelism and concurrency*; F.3.1 [Theory of Computation]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: Abstract machines, parallel languages, profiling semantics, speculation, threads

This research was supported in part by the National Science Foundation (NSF) CCR 92-58525 and CCR 97-06572.

Authors' addresses: J. Greiner, Rice University, Department of Computer Science, 3118 Duncan Hall, Houston, TX 77251; email: greiner@cs.rice.edu; G. E. Blelloch, Carnegie Mellon University, Department of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: guyb@cs.cmu.edu.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 1999 ACM 0164-0925/99/0300-0240 \$5.00

1. INTRODUCTION

Futures, lenient languages, and several implementations of graph reduction for lazy languages all use speculative evaluation (*call-by-speculation* [Hudak and Anderson 1987]) to expose parallelism. The basic idea of speculative evaluation, in this context, is that the evaluation of a function body can start in parallel with the evaluation of the corresponding function arguments. The evaluation of a variable in the body is then *blocked* if it references an argument that is not yet available and *reactivated* when the argument becomes available. With futures in languages such as Multilisp [Halstead 1985; 1989; Osborne 1989] and MultiScheme [Miller 1987], the programmer explicitly states what should be evaluated in parallel using the *future* annotation. In lenient languages, such as Id [Traub 1988; Nikhil 1991] and pH [Nikhil et al. 1995], by default all subexpressions can evaluate speculatively. With parallel implementations of lazy graph reduction [Hudak and Mohr 1988; Peyton Jones 1989; Joy and Axford 1992] speculative evaluation is used to overcome the inherent lack of parallelism of laziness [Kennaway 1994; Tremblay and Gao 1995].

Although call-by-speculation is a powerful mechanism that can achieve high degrees of parallelism, it is often very difficult or impossible to predict its performance characteristics without understanding details of the implementation. Requiring such an understanding is not only a burden on the users, but is often hard to specify, can be ill defined (nondeterministic timing changes can potentially cause serious differences in performance), and is implementation dependent (a new release might require changes in coding style or even algorithm design to generate efficient code). Furthermore, the performance effects of an implementation can go beyond constant factors, and potentially affect the running time asymptotically. For example, code that appears parallel to the users and should improve asymptotically with the number of processors could be fully serialized due to some obscure feature of an implementation.

The question we would like to address is whether it is possible to supply a model to users from which they can get at least some prediction of the performance of their code without having to understand the particulars of the implementation being used. For this purpose we use the idea of *provably efficient implementations* [Greiner 1997], which consist of

- (1) defining an abstract cost model from which the users can analyze their code;
- (2) specifying mappings from these costs to running times on various machine models; and
- (3) describing an implementation that guarantees that these mappings can be achieved.

The model and the mapping can be thought of as part of the language specification (i.e., something that would be included in the manual). An implementation is then considered correct if it is not only true to the semantics of the language specification, but is also true to the performance guarantee. Needless to say, given the many complicated features with hardware (e.g., caches and pipelines), trying to derive exact predictions of performance from an abstract model would be futile even for the simplest sequential languages. We therefore limit ourselves to guaranteeing

Machine Model	Time
Hypercube	$O(w/p + d \log p)$
CRCW PRAM	$O(w/p + d \log p / \log \log p)$

Fig. 1. The mapping of work (w) and depth (d) in the Parallel Speculative λ -calculus to running time, with high probability, on various machine models with p processors. The results assume that the number of independent variable names in a program is constant. We assume the hypercube can communicate over all wires simultaneously (multiport version). A butterfly with $p \log_2 p$ switches would have equivalent results as the hypercube.

asymptotic bounds. Although these asymptotic bounds cannot precisely predict performance, they can, for example, tell the users how runtime will scale with problem size or number of processors, and might be used by the users to guarantee that there will not be anomalous effects that cause asymptotic slowdowns.

To define our cost model we use a *profiling semantics*, which is an operational semantics augmented with information about costs [Rosendahl 1989; Sands 1990]. For every computation our semantics defines a directed acyclic graph (DAG), henceforth referred to as a computation graph, in which each node represents a single operation, and each edge represents either a data or control dependence between two operations. The computation graph can be thought of as an abstract trace of the computation and is independent of the implementation. The number of nodes in the DAG corresponding to a computation is referred to as the *work* of that computation, and the longest path is referred to as the *depth*. In the model, users can determine the work and depth of computations either analytically by calculating asymptotic bounds, or experimentally by having an implementation track the work and depth. The *cost mappings* we provide then relate the work and depth costs to running time on various machine models. This relationship is specified in terms of asymptotic bounds (see Figure 1) and includes all costs of the computation, except for garbage collection. With sufficient parallelism (i.e., when the first term dominates) these bounds are work efficient—the machine does no more than a constant factor more work (processor \times time) than required.

We use queues to maintain the threads that are blocked on each variable, and to achieve our bounds we present the first implementation strategy that fully parallelizes these queues allowing threads both to be enqueued when blocked and dequeued when reactivated in parallel. Our solution is based on using a dynamically growing array for each queue. The basic idea is to start with an array of fixed size. When the array overflows, we move the elements to a new array of twice the number of elements in the queue. Adding to the array, growing of the array, and dequeuing from the array can all be implemented in parallel using a *fetch-and-add* operation [Gottlieb et al. 1983; Ranade 1989]. To account for the cost of growing the array, we amortize it against the cost of originally inserting into the queue. Once the queues are parallelized, we use known scheduling results [Blumofe and Leiserson 1993] to achieve our overall time bounds.

The idea of including a cost model in a language definition, along with a proof of a mapping of that model to runtimes on parallel machines, was introduced in the NESL language [Blleloch 1992], and used in the Cilk [Blumofe et al. 1995] language. The original approach taken in NESL was somewhat informal, however, in that it did not have a well-specified profiling semantics. This led to some ambiguities in

the definition having to do with the cost of passing in large arguments (nonconstant size) to a parallel call. The risk of such ambiguities lead us to the conclusion that it is important to formalize the cost model in the semantics. Although one might find the level of detail we go through in this article unnecessary, we believe it is quite important in getting both the model and mappings correct. We have found several subtle ambiguities and problems in the process, one of which we describe briefly in the second example below. In addition to this article, we have used the approach of provably efficient implementations based on a profiling semantics to model call-by-value parallelism [Blelloch and Greiner 1995], and to more formally model NESL including space usage [Blelloch and Greiner 1996].

1.1 Motivating Examples

As an example of how an implementation can have asymptotic effects on performance, consider the following pseudocode:

```

y = future(exp)
in parallel for i = 1 to n
  a[i] = b[i] + y

```

which adds the result of expression *exp* to each element of **b**. The **future** allows the *n* threads from the parallel **for**-loop to start up while *exp* is still evaluating. If *exp* requires more time to compute than the time to fork the *n* threads, then these threads will need to block waiting for *y*. One possibility is to have the threads spin and keep checking if *y* is ready yet. Such *spin waiting* can be very inefficient, since the processor will be tied up doing no useful work. In fact, without having a fair schedule it can cause deadlock, since the thread that is computing *y* might never be scheduled. To avoid these problems most implementations will suspend a thread that reads a variable that is not ready by adding it to a queue associated with the variable [Miller 1987; Ito and Matsui 1989; Osborne 1989; Chandra et al. 1990; Kranz et al. 1989; Goldman and Gabriel 1988; Nikhil 1990; Feeley 1993] (some implementations will spin for a fixed amount of time and then suspend [Callahan and Smith 1990]). This modification will have an asymptotic effect on performance on the example code, since the thread computing *y* can use a full processor (the other threads will suspend) while in the spin-waiting implementation, assuming fair scheduling, *n* + 1 threads will need to share the *p* processor, giving the thread computing *y* an effective $p/(n + 1)$ fraction of a processor. Alternatively, one might consider a spin-waiting implementation that places a higher priority on computing *y* than on the *n* spinning threads. In full generality, however, this scheme is at least as complicated as suspending threads.

Perhaps less obvious is that the suspending implementation can still give performance that is asymptotically worse than one would expect. In the given code a large number of threads will try to suspend on a single variable almost simultaneously. All the implementations cited above serialize this process by using a linked list for the queue. This can have the effect of fully serializing the code even though it looks perfectly parallel. Although such serializing is likely to be much less common than problems caused by spin-waiting, it might be more insidious for the users, since it would only show up under certain conditions. Subtle timing differences in whether *y* is computed before the threads try to read it, for example, might have very seri-

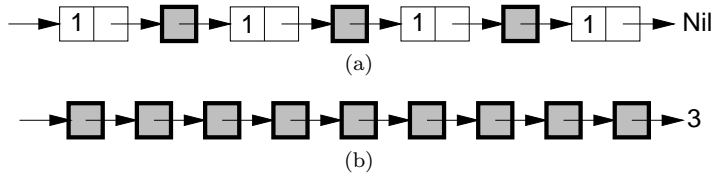


Fig. 2. The structures created by `func1` (a) and `func2` (b) in a standard implementation of futures. The shaded squares are future cells that start empty and are filled in when the future computation completes.

ous effects on performance. It is these asymptotic and often unpredictable effects in performance that motivate the use of a provably efficient implementation.

As an example of the subtleties of the model and why it can be important to be formal about the costs, we consider the following code fragment, written in Multilisp.

```
(defun func1 (n rest)
  (if (= n 1)
      (cons 1 rest)
      (let ((nhalf (floor (/ n 2))))
        (func1 nhalf (future (func1 (- n nhalf) rest))))))
  (func1 n nil))
```

This function creates a length- n list of all 1's. The `future` creates a process to evaluate one recursive call which can start in parallel with the other recursive call. If analyzed in the model described in this article, it leads to $\Theta(n)$ work and $\Theta(\log n)$ depth, since it forks off a tree of parallel processes of logarithmic depth. In fact, it is somewhat curious that the function can create a list of length n in $\Theta(\log n)$ depth in a side-effect-free language. We now make a simple change to the code by removing the `cons` in the base case.

```
(function func2 (n rest)
  (if (= n 1)
      rest
      (let ((nhalf (floor (/ n 2))))
        (func2 nhalf (future (func2 (- n nhalf) rest))))))
  (+ 2 (func2 n 3)))
```

This new function just returns `rest`, so the final result is 5 independent of n . The subtle point is that this use of `func2` now has depth $\Theta(n)$ instead of $\Theta(\log n)$ if analyzed in our model, even though the depth of recursive calls is still only $\Theta(\log n)$. The $\Theta(n)$ depth is consistent with all implementation strategies of futures we know of, including our own, since the strategies either create a chain of n futures (see Figure 2) or do not allow pointers to future cells to be written into other future cells. In the first case the addition requires $\Theta(n)$ time, since it needs to traverse the chain to find the 3 at the end, and in the second case `func2` is fully sequentialized, requiring $\Theta(n)$ time. Having the semantics correctly model this as a $\Theta(n)$ -depth computation is subtle. Roe [1991], for example, modeled this as a $\Theta(\log n)$ -depth

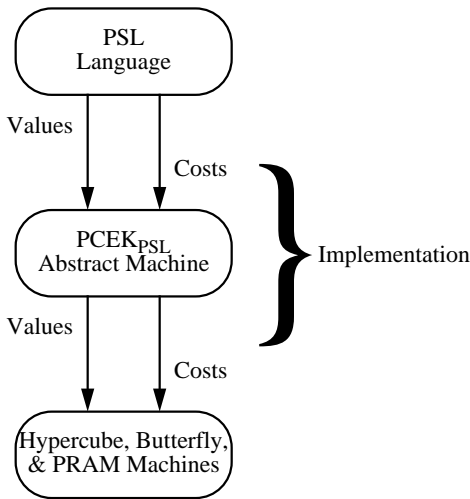


Fig. 3. The implementation maps values and costs of the profiling semantics (Section 2) to those of the machine (Section 4). The implementation is staged via an abstract machine model (Section 3). Its effects on costs are summarized by cost mappings.

computation, making it unlikely that one could prove any useful time bounds based on that model. Subtleties like this are easily missed or ill defined without having a formal semantics that account for costs.

1.2 Outline of the Model and Implementation

The article is organized by first describing the model and then describing the implementation. We use the pure λ -calculus with some arithmetic primitives as our basic language model, since it allows us to abstract away from language specifics and simplify the exposition. We use a profiling semantics to augment the language so that it returns directed acyclic graphs (DAGs) which abstractly represent the performance, and call this augmented language the *parallel speculative λ -calculus* (PSL). Section 2.1 describes the structure of the graphs returned by the semantics; Section 2.2 defines the semantics; and the Appendix describes how several common language features can be added to the language with only constant overheads.

Our implementation and bounds are based on simulating the PSL on the target machines. We stage this simulation into two parts to simplify the mapping, as shown in Figure 3. In Section 3.1 we define an intermediate model using an abstract machine called the $P\text{-CEK}_{\text{PSL}}^q$. The intermediate machine is a parallel version of the CESK machine [Felleisen and Friedman 1987], one of many variants of the original SECD machine for implementing the λ -calculus [Landin 1964]. Our machine maintains a set of states that are ready to execute. On each step the machine executes one step on q of these ready states in parallel, transforming each to 0, 1, 2, or m new ready states (0 if terminating or suspending, 1 if continuing, 2 if forking, and m if reactivating a set of m suspended states). The parameter q can be thought of as the number of processors. We prove that a derivation in the PSL model that returns a computation graph g can be simulated in the $P\text{-CEK}_{\text{PSL}}^q$

model in $W(g)/q + D(g)$ or fewer steps, where $W(g)$ is the work of the graph (number of nodes), while $D(g)$ is the depth (longest path).

The second half of the simulation is an implementation of the intermediate machine onto the target machines. In Section 4 we show that each step of a P-CEK_{PSLf} ^{$p \log p$} machine can be implemented in $O(\log p)$ amortized time with high probability on a p processor hypercube or CRCW PRAM. The amortization comes from how we grow the set of ready states, and the high probability time bound comes from cost of simulating a fetch-and-add or memory access on these machines [Ranade 1989; Matias and Vishkin 1991; 1995; Gil and Matias 1994]. Since we have a bound on the number of steps required by the machine, we can simply plug in these results and bound the total running time for these machines (see Figure 1).

We distinguish between a fully speculative implementation, in which it is assumed that the body and argument are always evaluated, and a partially speculative implementation for which speculation is limited. The call-by-speculation semantics of the PSL and the main results in this article assume a fully speculative implementation, but Section 5 considers partially speculative implementations that can either kill inaccessible tasks or only selectively execute tasks, thus reducing the work over call-by-value semantics.

1.3 Issues of Practicality

It is important to realize that the “implementation” we specify in this article is meant as a proof that the runtime bounds can be achieved within a constant factor (asymptotically). As often done in proofs of asymptotic bounds, many simplifications are made to the implementation that make the proofs much simpler at the cost of increasing constant factor overheads. We would expect that an actual implementation would maintain the asymptotic bounds while making many optimizations to reduce the constant factors. Here we list some of the techniques and assumptions we make, justify why we make them, and discuss how they can be modified to make a more practical implementation.

- (1) We define our model in terms of the pure λ -calculus with only arithmetic primitives added. We do not include even standard constructs such as local bindings (e.g., `let`), data structures (e.g., cons-cells), or conditionals as primitives directly, but instead assume they are defined using the pure λ -calculus. This may seem like an overly simplified language, but in regards to our goal of showing asymptotic bounds these other constructs can all be simulated in constant time, making them unnecessary as primitives, as shown elsewhere [Greiner 1997]. Leaving them out of the language greatly simplifies our semantics and reduces the number of proof cases. Needless to say, any real implementation of call-by-speculation should directly implement many of these constructs. Assuming that the direct implementation is at least as efficient as our implementation based on the primitives, these will improve the constants without affecting asymptotic bounds.
- (2) We assume that all arguments are evaluated speculatively. This assumption is made so that we do not need a separate set of rules to define the nonspeculative evaluation, and corresponding cases in the proof. A nonspeculative evaluation

c	\in	<i>Constants</i>	
x, y, z	\in	<i>Variables</i>	
e	\in	<i>Expressions</i>	$::= c \mid x \mid$
			$\lambda x. e \mid$ abstraction
			$e_1 e_2$ application

Fig. 4. Basic λ -calculus expressions.

construct should certainly be included in a practical implementation, since the constant overhead for creating threads and synchronization on variables can be high, especially since in many cases it might not be necessary.

- (3) We assume that every unit of computation is scheduled independently. This simplifies the mapping from the graph to the abstract machine and simplifies the abstract machine. In a real implementation there is no reason to break up threads to this fine level of granularity, and, in fact, any greedy schedule [Blumofe and Leiserson 1993] (one that makes sure that no processor sits idly when there is a ready thread) will achieve the same bounds. In practice, scheduling larger blocks of computation is likely both to reduce scheduling overhead and to increase cache hit ratios within each thread.
- (4) Related to scheduling units of computation, we use a highly synchronous implementation. Again, the tight synchronization is not necessary for the time bounds as long as the schedule is close to greedy.
- (5) We assume that when the array maintaining a queue needs to be copied because it overflows, all processors can become involved in the copy. This does not make much sense in an asynchronous or coarse-grained thread scheduler since it would involve interrupting the processors. Instead the implementation could add a set of jobs to the front of the thread queue each of which copies some block of entries to the new array (i.e., the copying of a queue can be scheduled by the same scheduler as the execution of the threads themselves).
- (6) Our machine models assume that any memory location can be read or written concurrently by all the processors. In the CRCW (concurrent read concurrent write) PRAM this assumption is made directly in the model, and in the hypercube we assume the network supports combining within the switches to support these concurrent accesses [Ranade 1989]. Since our language assumes parallel threads can access the same values, concurrent reads seem to be difficult to avoid in a general way. The concurrent writes are only used to implement the fetch-and-add operation, which is used to add threads to the suspension queues. Again we see no general way of avoiding them within the same time bounds. In more recent work we have shown that by restricting the language it is possible to implement call-by-speculation with purely exclusive access [Blelloch and Reid-Miller 1997]. It is an interesting question of whether the concurrent memory access capability is needed in the general case.

2. LANGUAGE AND PROFILING SEMANTICS

The core syntax of the λ -calculus we use is given in Figure 4, where the set of constants is defined later, and the set of variables is countably infinite. To keep

$ \begin{aligned} i &\in \text{Integers} \\ c \in \text{Constants} &::= i \mid \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \mathbf{lt} \mid \text{numeric constants} \\ &\quad \mathbf{add}_i \mid \mathbf{sub}_i \mid \mathbf{mul}_i \mid \mathbf{div}_i \mid \mathbf{lt}_i \end{aligned} $
--

Fig. 5. Basic λ -calculus constants.

the number of syntactic forms, and thus the number of semantic rules and the number of proof cases, to a minimum, we only added constructs to the language if they asymptotically affect the work and depth costs. In particular we did not include local bindings (e.g., `let`), recursion, conditionals, or built-in data structures (e.g., lists), since they can each be simulated in constant work and depth, as the Appendix discusses. However, we do include the integers and arithmetic operations as constants, since, although numerous schemes exist for encoding numerals in the λ -calculus, at best they require polylogarithmic time to simulate addition and multiplication [Parigot 1988].

Also to reduce the number of syntactic forms, we do not syntactically distinguish unary and binary functions. Instead, a binary function such as addition takes its arguments one at a time, i.e., $\mathbf{add} \ 1 \ 2 \equiv (\mathbf{add} \ 1) \ 2$. Thus the core language uses the constants defined in Figure 5. For example, \mathbf{add} represents binary addition, and \mathbf{add}_i represents the unary addition of the number i . While the unary functions are redundant with the corresponding binary functions, we include them in the syntax of expressions to simplify the presentation of the semantics (the unary functions are the result of applying the corresponding binary functions to their first argument).

The free variables of an expression, $FV(e)$, are defined as usual, where function abstraction, $\lambda x.e'$, is the only variable-binding construct.

2.1 PSL Computation Graphs

The PSL model is the λ -calculus together with a profiling semantics, which describes the parallelism by specifying a computational graph for every computation. This section describes the computation graphs generated by the PSL model. The evaluation of every expression generates a subgraph, and these subgraphs are composed by the semantics. In every subgraph we distinguish two nodes (potentially the same node): its source ns and minimum sink nt . The source represents the start of a computation, and the minimum sink represents when it returns a value. Edges represent control or data dependences in the program execution.

Definition 1. A PSL *computation graph* is a triple (ns, nt, NE) of its source, minimum sink, and a mapping from nodes to ordered sets of other nodes (representing the outgoing edges from each node).

We use ordered sets to represent the outgoing edges, since the graphs are ordered DAGs. When illustrating computation graphs in this article we will draw the outgoing edges for a node ordered from left to right and directed downward. We will also often draw graphs as triangles with their source at the top and their minimum sink at the bottom left, as shown in Figure 6. The bottom right corner represents the node (or possibly nodes) with the longest path from the source and will be referred to as the maximum sink(s). This rendering with the right bottom

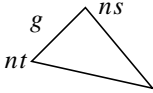

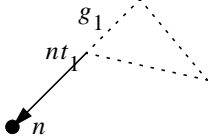
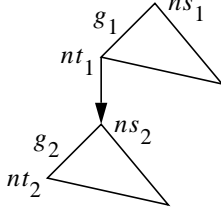
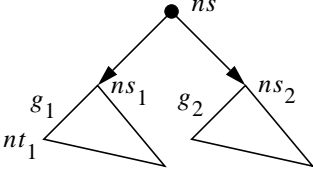
Graph g :	$\mathbf{1}$	$\mathbf{1} \leftarrow g_1$	$g_1 \oplus g_2$
(ns, nt, NE)	(n, n, \cdot)	$(n, n, [nt_1 \mapsto [n]])$	$(ns_1, nt_2, (NE_1 \uplus NE_2)[nt_1 \mapsto [ns_2]])$
	unique n 	unique n 	
$W(g)$:	1	1	$W(g_1) + W(g_2)$
Graph g :	$g_1 \wedge g_2$		
	$(ns, nt_1, (NE_1 \uplus NE_2)[ns \mapsto [ns_1, ns_2]])$		
	unique ns		
			
$W(g)$:	$W(g_1) + W(g_2) + 1$		

Fig. 6. The definition of combining operators for PSL computation graphs and work. Note that the work of the $\mathbf{1} \leftarrow g_1$ graph does not count the work of g_1 , since this operator is only used when g_1 is present elsewhere in the overall graph.

corner lower than the left is meant to illustrate that the maximum sink is at least as deep as the minimum sink.

The following operators, defined in Figure 6, will be used by the profiling semantics for building PSL computation graphs.

- $\mathbf{1}$ represents a singleton node.
- $g_1 \oplus g_2$ places the two graphs in series by returning all edges from both graphs along with an additional dependence edge from the minimum sink of g_1 to the source of g_2 .
- $\mathbf{1} \leftarrow g$ creates a new singleton node and an edge to this node from the minimum sink of g . It differs from $g \oplus \mathbf{1}$ in that it does not return the edges from g , and the new node rather than the source of g is the source of the returned graph.
- $g_1 \wedge g_2$ places the two graphs in parallel by returning all edges from both graphs along with an additional node with edges to the sources of both graphs. The minimum sink of g_1 becomes the minimum sink of the new graph.

The operators use \uplus to combine mappings that represent the neighbors:

$$NE_1 \uplus NE_2 = \bigcup_{n \in N} \begin{cases} [n \mapsto NE_1(n)] & n \in \text{dom}(NE_1), n \notin \text{dom}(NE_2) \\ [n \mapsto NE_2(n)] & n \notin \text{dom}(NE_1), n \in \text{dom}(NE_2) \\ [n \mapsto NE_1(n) \# NE_2(n)] & n \in \text{dom}(NE_1) \cap \text{dom}(NE_2) \end{cases}$$

where $N = \text{dom}(NE_1) \cup \text{dom}(NE_2)$, and $X \# Y$ appends two arrays or mappings.

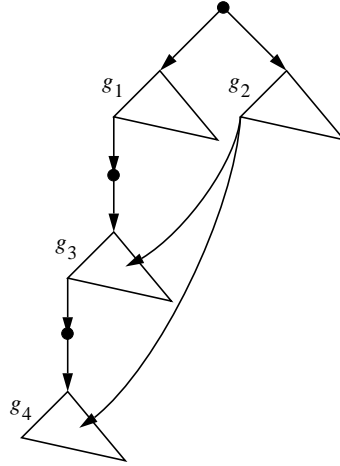


Fig. 7. Illustration of a case where combined computation graphs share edges from the same node. This shows the graph $(g_1 \wedge g_2) \oplus \mathbf{1} \oplus g_3 \oplus \mathbf{1} \oplus g_4$, where each of g_3 and g_4 contains a subgraph $\mathbf{1} \leftarrow g_2$.

$l \in \text{Locations}$			
$v \in \text{Values}$	$::= c \mid l$		
$sv \in \text{StoreValues}$	$::= \mathbf{cl}(\rho, x, e)$	closure	
$\rho \in \text{Environments}$	$= \text{Variables} \xrightarrow{fin} \text{Values} \times \text{Graphs}$		
$\sigma \in \text{Stores}$	$= \text{Locations} \xrightarrow{fin} \text{StoreValues}$		

Fig. 8. PSL values, stores, and environments.

The domains of the mappings may overlap if they each contain an edge from the same node. This may occur if either graph joined by an operator includes a data edge created by $\mathbf{1} \leftarrow g$, for some g , as in Figure 7.

We define the depth of a node in a graph as the longest path to that node. We define the work of a graph, $W(g)$, as the total number of nodes in the graph, and the depth of a graph, $D(g)$, as the longest path in the graph, i.e., the maximum depth of any node in the graph.

2.2 Semantics

We now define the PSL model using a profiling semantics. The semantics is defined recursively in terms of a *judgment*, or relation, describing the evaluation of expression e to value v . We explicitly manage the memory via stores, since this eases comparison with the implementation which also uses a store. In this model, environments map each variable to both a value and the computation graph describing the evaluation to that value. These values, stores, and environments are defined in Figure 8. A variable’s graph is used to describe when that value has been computed. The profiling semantics is then given by Definition 2.

Definition 2. In the PSL model, starting with the environment ρ and store σ ,

$\rho, \sigma \vdash c \xrightarrow{\text{PSL}} c, \sigma; \mathbf{1}$	(CONST)
$\rho, \sigma \vdash \lambda x.e \xrightarrow{\text{PSL}} l, \sigma[l \mapsto \mathbf{cl}(\rho, x, e)]; \mathbf{1}$ where $l \notin \sigma$	(LAM)
$\frac{\rho(x) = v; g}{\rho, \sigma \vdash x \xrightarrow{\text{PSL}} v, \sigma; \mathbf{1} \leftarrow g}$	(VAR)
$\frac{\rho, \sigma \vdash e_1 \xrightarrow{\text{PSL}} l, \sigma_1; g_1 \quad \rho, \sigma_1 \vdash e_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; g_2 \quad \sigma_2(l) = \mathbf{cl}(\rho', x, e_3) \quad \rho'[x \mapsto v_2; g_2], \sigma_2 \vdash e_3 \xrightarrow{\text{PSL}} v_3, \sigma_3; g_3}{\rho, \sigma \vdash e_1 e_2 \xrightarrow{\text{PSL}} v_3, \sigma_3; (g_1 \wedge g_2) \oplus \mathbf{1} \oplus g_3}$	(APP)
$\frac{\rho, \sigma \vdash e_1 \xrightarrow{\text{PSL}} c, \sigma_1; g_1 \quad \rho, \sigma_1 \vdash e_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; g_2 \quad \delta(\sigma_2, c, v_2) = v_3, \sigma_3; g_3}{\rho, \sigma \vdash e_1 e_2 \xrightarrow{\text{PSL}} v_3, \sigma_2 \cup \sigma_3; (g_1 \wedge g_2) \oplus (\mathbf{1} \leftarrow g_2) \oplus g_3}$	(APPC)

Fig. 9. The profiling semantics of the PSL model using the definition of δ in Figure 10, and the graph combining operators of Figure 6.

c	v	$\delta(\sigma, c, v)$			if/where
		v'	σ'	g'	
add	i_1	add $_{i_1}, \cdot$	\cdot	$\mathbf{1}$	
add $_{i_1}$	i_2	$i_1 + i_2, \cdot$	\cdot	$\mathbf{1}$	
sub	i_1	sub $_{i_1}, \cdot$	\cdot	$\mathbf{1}$	
sub $_{i_1}$	i_2	$i_1 - i_2, \cdot$	\cdot	$\mathbf{1}$	
mul	i_1	mul $_{i_1}, \cdot$	\cdot	$\mathbf{1}$	
mul $_{i_1}$	i_2	$i_1 * i_2, \cdot$	\cdot	$\mathbf{1}$	
div	i_1	div $_{i_1}, \cdot$	\cdot	$\mathbf{1}$	
div $_i$	i'	$[i'/i], \cdot$	\cdot	$\mathbf{1}$	
lt	i_1	lt $_{i_1}, \cdot$	\cdot	$\mathbf{1}$	
lt $_{i_1}$	i_2	$l, [l \mapsto \mathbf{cl}(\cdot, x, \lambda y.x)]$	\cdot	$\mathbf{1}$	$i_1 < i_2, l \notin \sigma$
lt $_{i_1}$	i_2	$l, [l \mapsto \mathbf{cl}(\cdot, x, \lambda y.y)]$	\cdot	$\mathbf{1}$	$i_1 \geq i_2, l \notin \sigma$

Fig. 10. The δ function defining constant application. The values resulting from the **lt** $_i$ application are standard λ -calculus encoding of booleans.

the expression e evaluates to value v and the new store σ' with computation graph g , or

$$\rho, \sigma \vdash e \xrightarrow{\text{PSL}} v, \sigma'; g,$$

if it is derivable from the rules of Figure 9.

We are primarily interested in the “top level” case where the evaluation starts with an empty environment, i.e., $\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g$ represents the evaluation of program e to its result value v .

As usual, a constant evaluates to itself; an abstraction evaluates to a *closure* containing the current environment; and a variable evaluates to the value found for that variable in the current environment. An application evaluates both the function and argument and

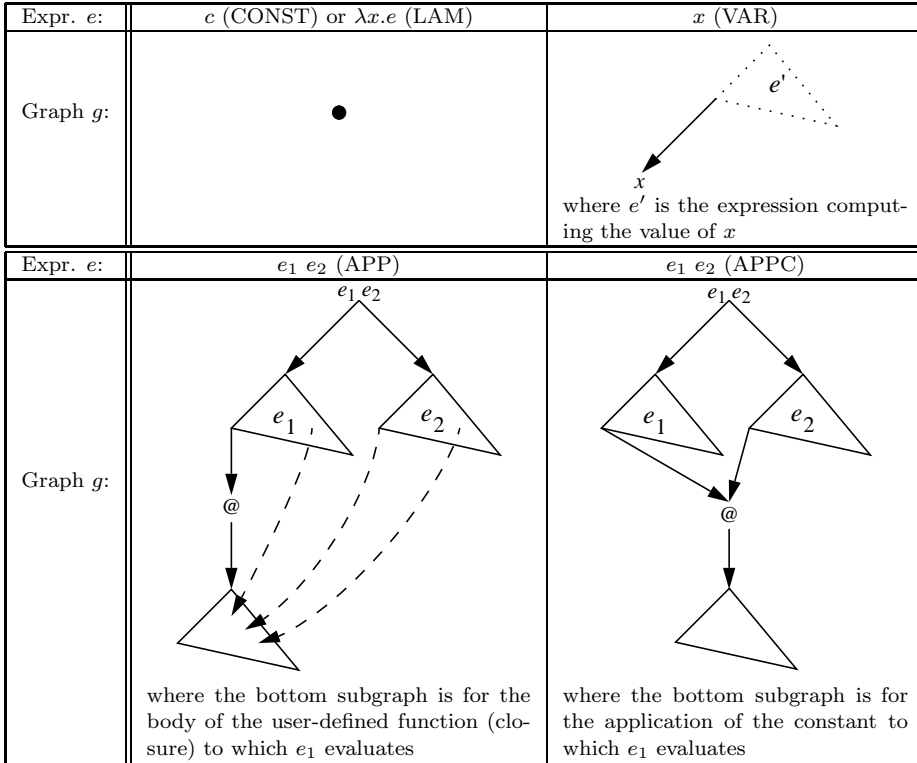


Fig. 11. Illustration of computation graphs for the PSL model. The dashed lines represent possible dependencies. Where applicable, nodes are labeled with the expression whose evaluation is represented by the subsequent graph. Nodes labeled “@” represent the initiation of an application and are used to simplify the later correspondence with the machine of Section 3. It corresponds to the **1** in the APP and APPC rules.

- if the function value is a closure, it evaluates the closure body using the closure’s defining environment or
- if the function value is a constant, it evaluates the constant application as defined by δ .

Using the APPC rule with the δ function is a convenient way to define the application of most constants—in particular, those that do not depend on the general evaluation relation. Alternatively, we could define a separate semantic application rule for each constant.

Figure 11 illustrates the computation graphs that are formed by the rules of the semantics. Evaluating a constant or abstraction creates a single node. Evaluating a variable also creates a single node but adds an edge from the sink of the subgraph that evaluated the value of the variable to its use. Recall that the environments used in the semantics store along with every value the subgraph that describes its evaluation. Evaluating an application $e_1 e_2$ evaluates the function e_1 and argument e_2 in parallel, and then sequentially after evaluating e_1 it evaluates the function body or constant application.

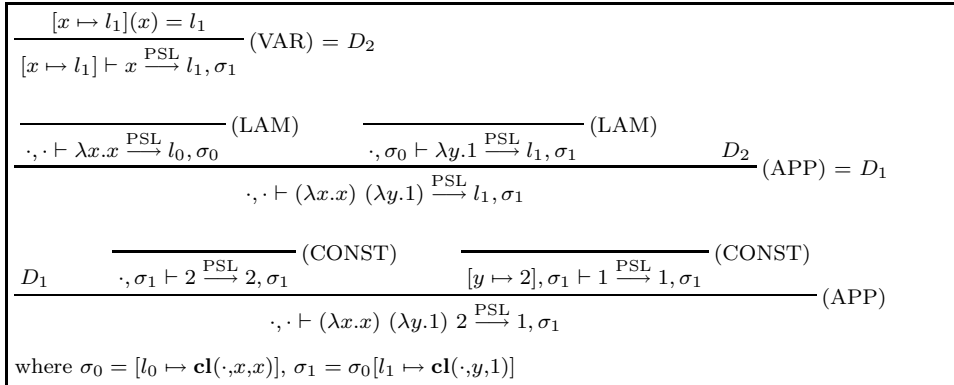


Fig. 12. PSL profiling semantics derivation for Example 1, omitting the computation graphs. For readability, the derivation tree is broken into three subtrees.

We choose a consistent sequential ordering on the evaluation of subexpressions, placing the function’s subgraph before that of the argument. This choice makes the branching in computation graphs resemble that of the corresponding syntax tree. As explained later, this ordering will reflect the execution ordering when there are not enough processors to parallelize all computation.

Example 1. Consider the evaluation of the following expression:

$$(\lambda x.x) (\lambda y.1) 2.$$

Figures 12 and 13 show the derivation tree (excluding costs) and the overall computation graph, respectively. The left spine including the root represents the main thread; the other two nodes are separate threads, only one of which synchronizes.

The reader may be concerned whether the computation graphs actually represent real costs. First, the nodes may represent significantly different amounts of real work, as, for example, synchronization would take significantly more time than evaluating a constant. Second, either or both of variable lookup or closure creation (i.e., evaluating a variable or a function) could take nonconstant time, but they are both represented by single nodes. It is important to realize, however, that the computation graphs represent the time costs as defined abstractly in the language model, and these costs must still be mapped to the machine model. Furthermore to achieve more accurate cost predictions it would not be difficult to refine the model by, for example, adding weights to the nodes.

We now describe the form of PSL computation graphs to get a better intuitive grasp of speculative computation. There is a single node to start the application, with edges to the subcomputations, but there is not necessarily a single node for synchronization. There is also an edge from the function’s graph to that of the function body, but the existence of edges from the argument’s graph depends on whether its value is accessed. In an application, edges may connect the interior nodes of the subgraphs, from either the function or argument subgraph to the function body subgraph. What edges exist depend on the expressions:

—Edges from the argument value (i.e., the argument’s minimum sink) connect to

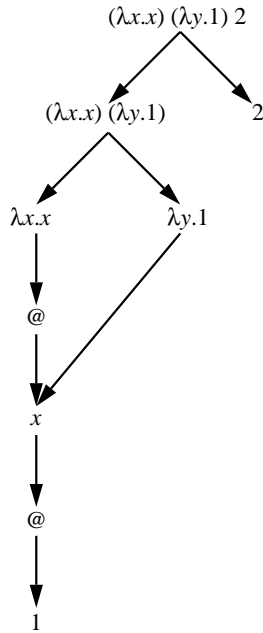


Fig. 13. PSL computation graph for Example 1. For an application expression, the “@” node between the function and the function body represents the application of the function value and a placeholder for its argument.

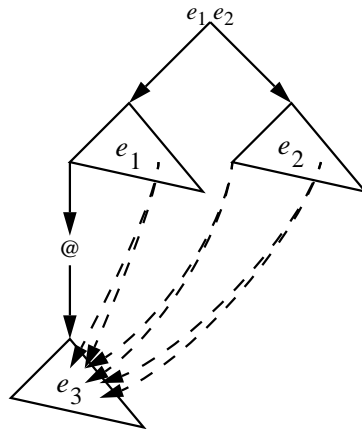


Fig. 14. PSL computation graphs may have multiple edges from nodes.

each of its uses within the function body. Note that there may be multiple such edges, as Figure 14 illustrates, although other figures show only one edge to avoid clutter.

—If the argument’s value is a closure, it may communicate the name of another

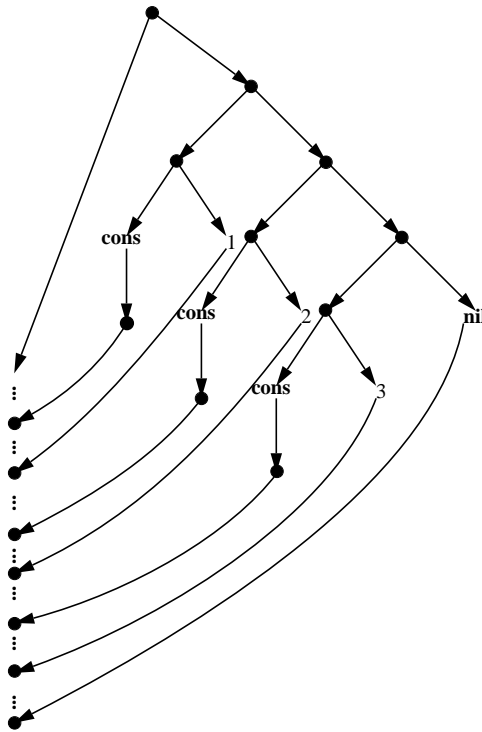


Fig. 15. Illustration of PSL computation graph where the function branch accesses a list’s elements in order and the argument branch creates the list. To access each element, the function must first access the cons-cell containing the element. The graph is simplified, with some nodes consolidated, but still representing constant amounts of computation.

value being computed within the argument’s subgraph. If that value is used in the function body, there is an edge from within the argument subgraph to its use, somewhere below the node linked to the argument’s value. For example, if the argument constructs a list, its value represents the first cons-cell. Edges would exist for each accessed element and cons-cell of the list, as illustrated in Figure 15.

—Similarly, the edge from the function to the function body may also communicate names of other values being computed within the function. These names are in the environment of the closure to which the function evaluates.

For an application expression, the “@” node between the function and the function body represents the application of the function value and a placeholder for its argument. It could be omitted with a resulting constant factor difference in the work and depth. Also, note that the computation graphs are not compositional in terms of their subgraphs.

While each edge represents a control dependence and allows for the flow of data, it can be intuitively helpful to distinguish two classes of edges. The edges for applications can be thought of as “control” edges, and the edges for variable lookups

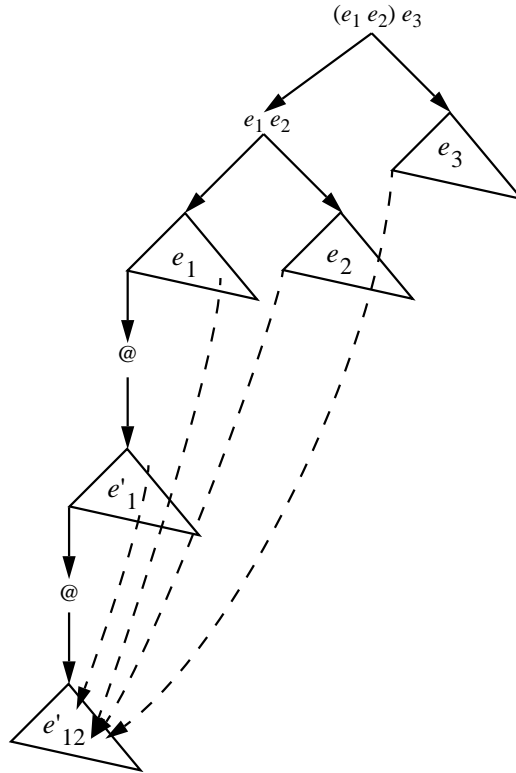


Fig. 16. PSL computation graph of nested applications $(e_1 e_2) e_3$, where e_1 evaluates to $\lambda x.e'_1$, and $e_1 e_2$ evaluates to $\lambda y.e'_{12}$.

would be “data” edges.

The asymmetric nature of these graphs leads to a useful notion of threads, which is formally defined in the abstract machine. The thread evaluating an application expression

- spawns a new thread to evaluate the argument,
- evaluates the function, and
- evaluates the function body.

Recursively, the evaluation of both the function and function body generally use additional threads. For example, in Figure 16, evaluating $(e_1 e_2) e_3$, the initial thread spawns a new thread for e_3 , then evaluates the inner application, spawning a new thread for e_2 , and then evaluates e_1 . The same thread then evaluates the function bodies e'_1 and e'_{12} . Although not shown, additional data edges may come into the graph for the inner application body e'_1 from outside the graph for the application $e_1 e_2$. As in the following example, threads follow the leftmost control edges until encountering a parent thread or until the thread simply ends.

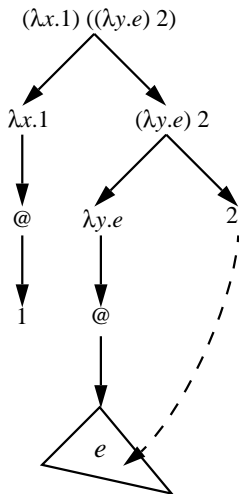


Fig. 17. PSL computation graph for Example 2.

Example 2. Consider the evaluation of the following expression:

$$(\lambda x.1) ((\lambda y.e) 2).$$

The computation graph for this expression is given in Figure 17. The leftmost edges including the root represent the main thread; the middle branch, another thread; and the rightmost node, another. The evaluation of e may use additional threads. No thread synchronizes with the main thread, but the rightmost thread might need to synchronize with the evaluation of e .

3. FULLY SPECULATIVE INTERMEDIATE MODEL

Section 2 introduced an abstract semantic model which we now show how to implement in a more concrete machine model and prove performance bounds for the implementation. As in a compiler, staging the implementation via an intermediate language or model frequently simplifies the problem. Here we stage the implementation using a parallel abstract machine model called the $\text{P-CEK}_{\text{PSL}f}^q$. The CEK in the name comes from the fact that each state consists of Control information, an Environment, and a “K”ontinuation. The “P” indicates that there are multiple states that run in parallel, the “PSL f ” that it evaluates the fully speculative PSL model, and the q specifies the maximum number of states evaluated on each step.

In this section we describe the $\text{P-CEK}_{\text{PSL}f}^q$ model, give some background on computation graph traversals, and then use this to prove a relationship between work and depth in the PSL model and the number of steps of the $\text{P-CEK}_{\text{PSL}f}^q$. In Section 4 we describe the implementation of the intermediate $\text{P-CEK}_{\text{PSL}f}^q$ machine model on more standard parallel machine models.

3.1 The Model

The $\text{P-CEK}_{\text{PSL}f}^q$ machine performs a series of steps, each transforming an ordered set of states (technically substates) and a store into a new ordered set of states

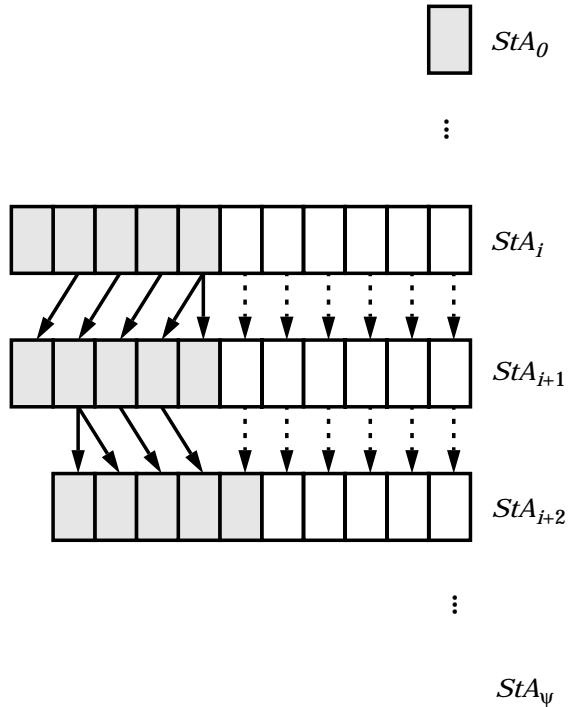


Fig. 18. Illustration of $\text{P-CEK}_{\text{PSLf}}^q$ active states during an evaluation. It starts with one active state representing the entire program and ends with no active states. The states are kept in a stack. At most q states are selected each step. Here $q = 5$, and these states are shaded. These can create zero or more new states (solid arrows). Unselected states are still active in the next step (dashed arrows).

and new store for the next step. Figure 18 illustrates this process. The states that can be transformed on a given step are called the current *active states*, each representing a thread of computation which can be performed in parallel. The machine starts with a single active state representing the entire computation, and it ends when there is no active state left. Each state uses a location for its eventual result value, so the program’s result is accessible through the initial state once it has been evaluated. Each state is used for computation on only one step—that step creates new states to perform any successive computation, i.e., we think of creating new states and discarding old states rather than modifying states. Each step also uses a global store to not only keep track of the program’s store contents, but also to record partial results of the computation. Each state consists of a control string C , consisting of the expression to be evaluated and a thread identifier (discussed below), an environment ρ , and a continuation κ .

Since the intermediate model is machine-like, different costs are of interest than in the profiling semantics. Here we track two costs: Q , the total number of states processed, and ψ , the number of parallel steps. We will relate these costs to the computation graphs of the profiling semantics.

The currently active states represent those states whose computation can be performed now. Other states can be suspended waiting for a variable value. On each step we *select* and use at most q active states (q is a parameter of the machine and is fixed during a computation). In other models, this is necessary for space efficiency [Blelloch and Greiner 1996; Greiner 1997], as it bounds the numbers of active states and thus the space to store them. Here, it provides no such bound in general, but does reduce the space in many typical examples. We later relate q to the number of processors available on the machine.

We formalize the intuitive notion of a thread as a central data structure of the implementation. A thread represents a series of states over time computing the same value, as described previously. A *thread* τ is a pair of locations that contain the information common to these states:

- (1) The first location contains either the resulting value v of the thread or a marker *Noval* indicating the value has not been computed yet. Since a thread provides a pointer to its result value, environments now map variables to the threads evaluating them, rather than to their values. Of course, a more realistic implementation would also cache the value in the environment once computed, to avoid the extra indirection.
- (2) The second location contains a set of *suspended states* waiting for this thread's value. When this thread finishes, these states are reactivated so that they can continue with the value.

A thread's two components are selected with π_1 and π_2 .

Each state records which thread it belongs to by including the thread in its control string C . This information is passed from state to state, for example, from application state to the state representing the function branch, and eventually to the state representing the function body. Only one state of a thread is accessible (i.e., active or suspended) in the machine at any given time. Thus, while we describe the machine in terms of states, we could describe it equivalently in terms of threads.

When a thread τ_1 evaluates an expression $e_1 e_2$ a new thread τ_2 is created to evaluate e_2 , and a pointer to it is placed in the continuation of τ_1 . The thread τ_1 then evaluates e_1 , and assuming this results in $\lambda x.e$, it removes τ_2 from its continuation and evaluates e with x bound to τ_2 in the environment. The continuation is used only to record the thread of its argument so that this information can be passed to the function body. When a thread finishes (has computed and written its result) it can simply die.

The initial state and thread of the computation uses a location l_{res} for the thread's eventual result. When the machine finishes, the result value v is in that location, and there are no active states left.

The above descriptions lead us to the definition of domains for the intermediate machine given in Figure 19, where the ellipsis represents the basic λ -calculus expressions of Figure 4. Note that we also introduce an expression not in the profiling semantics. The expression $@ v \tau$ represents the decision point after a function has evaluated and is about to be applied as to whether its value v is a constant or a closure.

$l \in \text{Locations}$		
$v \in \text{Values}$	$::= c \mid l$	
$sv \in \text{StoreValues}$	$::= \mathbf{cl}(\rho, x, e)$	closure
$\tau \in \text{Threads}$	$::= (l, l)$	
ValueOpts	$::= \mathbf{Noval} \mid v$	optional value
$e \in \text{Expressions}$	$::= \dots \mid @ v \tau$	application
$C \in \text{Controls}$	$::= (e, \tau)$	
$\rho \in \text{Environments}$	$= \text{Variables} \xrightarrow{fin} \text{Threads}$	
$\kappa \in \text{Continuations}$	$::= \bullet \mid$ $\mathbf{fun}(\tau \kappa)$	thread finishing function finishing
$st \in \text{States}$	$::= (C, \rho, \kappa)$	
$St \in \text{StateArrays}$	$::= \vec{st}$	
$I \in \text{IntermediateStates}$	$::= St \mid$ $\mathbf{Fin}(v \tau)$	new/reactivated states finishing state
$\sigma \in \text{Stores}$	$= \text{Locations} \xrightarrow{fin} (\text{StoreValues} +$ $\text{ValueOpts} +$ $\text{StateArrays})$	

Fig. 19. P-CEK_{PSLf}^q domains. The ellipsis represents the expressions of Figure 4.

Definition 3. A step i of the P-CEK_{PSLf}^q machine, written

$$StA_i, \sigma_i \xrightarrow{\text{PSLf}, q} StA_{i+1}, \sigma_{i+1}; Q_i,$$

is defined in Figure 20. It starts with a stack of active states StA_i and a store σ_i and produces a new stack and store for the next step. This step processes Q_i states.

Definition 4. In the P-CEK_{PSLf}^q machine, the *evaluation of expression e to value v , starting in the environment ρ and store σ_0 , ends with store σ_ψ and processes Q states, in ψ steps*, or

$$\rho, \sigma_0 \vdash e \xrightarrow{\text{PSLf}, q} v, \sigma_\psi; Q, \psi.$$

For each of these $i \in \{0, \dots, \psi - 1\}$ steps,

$$StA_i, \sigma_i \xrightarrow{\text{PSLf}, q} StA_{i+1}, \sigma_{i+1}; Q_i,$$

such that

- the machine starts with one active state and one thread for the whole program:
 $StA_0 = [(e, \tau), \rho, \bullet]$, $\sigma_0 = [l_{res} \mapsto \mathbf{Noval}][l'_{res} \mapsto []]$;
- the machine ends with zero active states and the result value: $StA_\psi = []$,
 $\sigma_\psi(l_{res}) = v$; and
- the total number of states processed is $Q = \sum_{i=0}^{\psi-1} Q_i$.

Each step consists of one substep for computation and two substeps for communication and synchronization. The computation substep's transition $\xrightarrow{\text{PSL}}_{comp}$ performs a case analysis on the state's expression and generates up to two new states or a special intermediate state for use in the following substeps:

Computation substep, $\xrightarrow{\text{PSL}}_{\text{comp}}$:		
st	I	if/where
$((c, \tau), -, \kappa) -$	$\xrightarrow{\text{PSL}}_{\text{comp}} \text{throw}(c, \tau, \kappa)$	·
$((x, \tau), \rho, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{comp}} \text{throw}(v, \tau, \kappa)$	· $\sigma(\pi_1(\rho(x))) = v$
$((\lambda x.e, \tau), \rho, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{comp}} \text{throw}(l, \tau, \kappa)$	$[l \mapsto \text{cl}(\rho', x, e)] \quad l \notin \sigma$
$((e_1 e_2, \tau), \rho, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{comp}} [((e_1, \tau), \rho, \text{fun}\langle \tau' \kappa \rangle), ((e_2, \tau'), \rho, \bullet)]$	$[l \mapsto \text{Noval}, \tau' = (l, l'), \quad l, l' \notin \sigma$ $l' \mapsto []]$
$((@ l \tau', \tau), \cdot, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{comp}} [((e, \tau), \rho[x \mapsto \tau'], \kappa)]$	· $\sigma(l) = \text{cl}(\rho, x, e)$
$((@ c \tau', \tau), \cdot, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{comp}} \text{throw}(v', \tau, \kappa)$	σ' $\sigma(\pi_1 \tau') = v,$ $\delta(\sigma, c, v) = v', \sigma'; -$
where $\text{throw}(v, \tau, \bullet) = \text{Fin}\langle v \tau \rangle$ $\text{throw}(v, \tau, \text{fun}\langle \tau' \kappa \rangle) = [(@ v \tau', \tau), \cdot, \kappa]$		
Reactivation and blocking substeps, $\xrightarrow{\text{PSL}}_{\text{react}}$ and $\xrightarrow{\text{PSL}}_{\text{block}}$:		
I	St	
$\text{Fin}\langle v \tau \rangle$	$\sigma \xrightarrow{\text{PSL}}_{\text{react}} \sigma(\pi_2 \tau)$	$[\pi_1 \tau \mapsto v]$ (reactivate)
\vec{st}	$- \xrightarrow{\text{PSL}}_{\text{react}} []$	· (ignore)
st	St	if/where
$((x, \tau), \rho, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{block}} []$	$\sigma[\pi_2(\rho(x)) \mapsto [st] \# \sigma(\pi_2(\rho(x)))]$ $\sigma(\pi_1(\rho(x))) = \text{Noval}$ (block)
$((@ c \tau', \tau), \cdot, \kappa) \sigma$	$\xrightarrow{\text{PSL}}_{\text{block}} []$	$\sigma[\pi_2 \tau' \mapsto [st] \# \sigma(\pi_2 \tau')]$ $\sigma(\pi_1 \tau') = \text{Noval}$ (block)
st	$\sigma \xrightarrow{\text{PSL}}_{\text{block}} [st]$	σ otherwise (ignore)
Step, $\xrightarrow{\text{PSL}f, q}$:		
$StA, \sigma \xrightarrow{\text{PSL}f, q} (\# \vec{st}) \# (\# \vec{st}') \# [st_{q'}, \dots, st_{k-1}], \sigma''; q'$		
if	$StA = [st_0, \dots, st_{k-1}]$	
	$q' = \min(q, k)$	select at most q states per step
st_i, σ	$\xrightarrow{\text{PSL}}_{\text{comp}} I_i, \sigma_i$	$\forall i \in \{0, \dots, q' - 1\}$ $\sigma' = \sigma \cup (\bigcup \vec{\sigma})$
I_i, σ'	$\xrightarrow{\text{PSL}}_{\text{react}} St_i, \sigma'_i$	$\forall i \in \{0, \dots, q' - 1\}$ $\sigma''_0 = \sigma' \cup (\bigcup \vec{\sigma'})$
\vec{st}'	$= [st \mid st \in I_0, \dots, I_{q'-1}]$	collect newly created states
k	$= \vec{st}' $	
st'_j, σ''_j	$\xrightarrow{\text{PSL}}_{\text{block}} St'_j, \sigma''_{j+1}$	“sequentially” $\forall j \in \{0, \dots, k - 1\}$ $\sigma''' = \sigma''_k$

Fig. 20. Definition of the P-CEK $^q_{\text{PSL}f}$ abstract machine step. Assume all new locations of the computation step are chosen or renamed to be distinct. Note that $\#$ and $\#$ are array append operators.

- The cases for constants and abstractions correspond to those in the profiling semantics.
- A variable lookup must get the value from the appropriate thread. Note that the variable lookups are guaranteed not to block because active states correspond to ready nodes, i.e., a previous step’s use of the third substep guarantees this value is available when requested.
- Evaluating an application $e_1 e_2$ creates two states, one to evaluate the function e_1 and one to evaluate argument e_2 . Later steps can evaluate the function and argument in parallel. The continuation of each new state indicates which branch it is. Also, the state for function and the state eventually created for the function body are considered to be the same thread as the application. The state for the argument starts a new thread.
- Evaluating an expression $@ v \tau$ initiates evaluation of the function body or performs a constant application, as appropriate.

Each step uses the $\overset{\text{PSL}}{\hookrightarrow}_{comp}$ transition in parallel for each of the selected active states.

The communication substeps finish and suspend states, respectively. For each intermediate state $\text{Fin}(v \tau)$, representing a thread that is finishing this step, the second substep uses the $\overset{\text{PSL}}{\hookrightarrow}_{react}$ transition to reactivate the states suspended on thread τ and store its result value v . Reactivated states correspond to ready nodes, since the value that they need is now available.

For each newly created state (i.e., we do not need to consider the reactivated states), the third substep uses the $\overset{\text{PSL}}{\hookrightarrow}_{block}$ transition to check whether it would block and, if so, add it to the set of suspended states owned by the thread on which it would block. The blocked thread reactivates in a later instance of the second substep once the value is available. While Figure 20 describes the semantics sequentially for simplicity, the following section shows that the instances of this transition can be parallelized so that all these states suspend at once. Synchronization is required between the two communication substeps to ensure that a thread’s result value is found in the last substep if stored in the second substep.

The step ends by adding the new states that have not blocked and the reactivated states to the active states stack. In the $\text{P-CEK}_{\text{PSLf}}^q$, the active states do not need to be treated as a stack, but that is one way to ensure determinacy.

Since each transition represents constant work, the total work for the step is defined as q' .

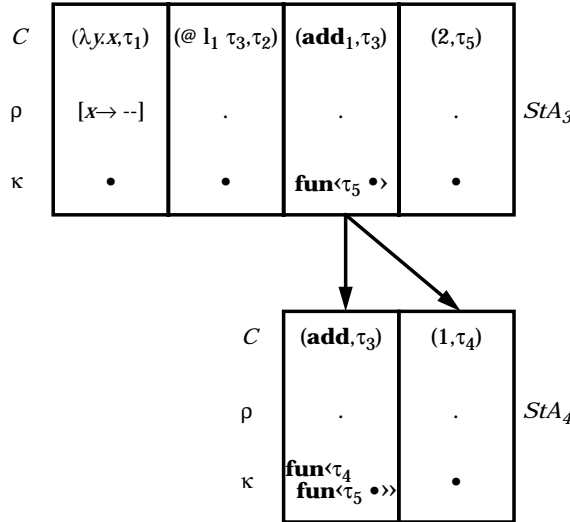
Example 3. As an example of $\text{P-CEK}_{\text{PSLf}}^q$ execution, Figure 21 shows the active states at the beginning of each step of evaluating $(\lambda x. \lambda y. x) ((\lambda z. z) (\mathbf{add} \ 1 \ 2))$. Figure 22 shows additional detail of one step. For lower values of q , the evaluation might take more steps, but it processes the same total number of states. The computation graph of the corresponding profiling semantics evaluation is shown for comparison in Figure 23, using the appropriate states’ expressions as node labels. Observe that each of these executions is a greedy q -traversal of the graph.

Example 4. If we apply the previous example program to another argument, the main thread must then wait for the computation of the value 3, as Figure 24 shows.

Step i	$q \geq 4$		$q = 2$	
	expressions in StA_i	q'	expressions in StA_i	q'
0	$(\lambda x.\lambda y.x) ((\lambda z.z) (\mathbf{add} \ 1 \ 2))$	1	$(\lambda x.\lambda y.x) ((\lambda z.z) (\mathbf{add} \ 1 \ 2))$	1
1	$\lambda x.\lambda y.x, (\lambda z.z) (\mathbf{add} \ 1 \ 2)$	2	$\lambda x.\lambda y.x, (\lambda z.z) (\mathbf{add} \ 1 \ 2)$	2
2	$@ \ l_0 \ \tau_1, \lambda z.z, \mathbf{add} \ 1 \ 2$	3	$@ \ l_0 \ \tau_1, \lambda z.z, \mathbf{add} \ 1 \ 2$	2
3	$\lambda y.x, @ \ l_1 \ \tau_2, \mathbf{add} \ 1, \ 2$	4	$\lambda y.x, @ \ l_1 \ \tau_2, \mathbf{add} \ 1 \ 2$	2
4	$\mathbf{add}, \ 1$	2	$\mathbf{add} \ 1 \ 2$	1
5	$@ \ \mathbf{add} \ \tau_3$	1	$\mathbf{add} \ 1, \ 2$	2
6	$@ \ \mathbf{add}_1 \ \tau_4$	1	$\mathbf{add}, \ 1$	2
7	z	1	$@ \ \mathbf{add} \ \tau_3$	1
8			$@ \ \mathbf{add}_1 \ \tau_4$	1
9			z	1
	States processed:	15	States processed:	15

where l_0 contains $\mathbf{cl}(-, x, \lambda y.x)$,
 l_1 contains $\mathbf{cl}(-, z, z)$, and
 τ_1 and τ_2 are defined in Figure 23.

Fig. 21. P-CEK_{PSLf}^q evaluations for Example 3. The underlined expressions correspond to the selected states of each step.



where τ_3 has state $((z, \tau_2), [z \mapsto -], \bullet)$ in its suspended queue at the start of step 4.

Fig. 22. Active states for step 3 ($q \geq 4$) of Example 3.

3.2 Graph Traversals

The following definitions and theorems about traversals are either standard graph terminology or are from Blumofe and Leiserson [1993] or Blelloch et al. [1995]. Note that the definition of a graph traversal is somewhat different than is standard in that it requires all nodes to be visited.

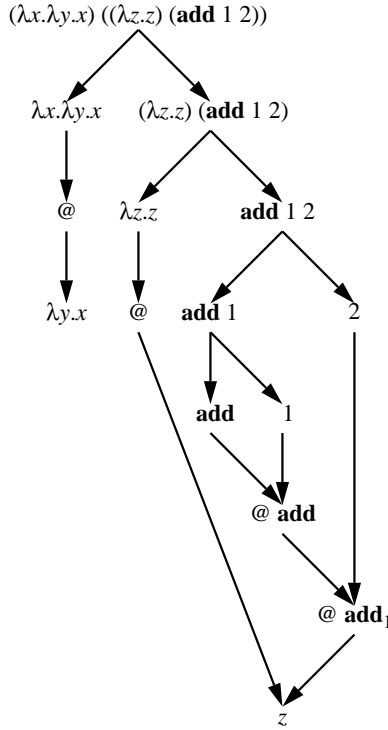


Fig. 23. PSL computation graph for Example 3. The threads are numbered left to right, i.e., the leftmost spine is τ_0 , the next leftmost is τ_1 , etc. Note that this is not the order of creation of the threads in this example. Also note that the value (a location pointing to the closure for $\lambda y.x$) is obtained at the end of step 3. The remainder of the computation is for the value of x if this closure were applied.

Definition 5. A *parallel traversal* of a graph g is a sequence of $k \geq 1$ steps, where each step i , for $i = 0, \dots, k - 1$, defines a set of nodes, V_i (that are *visited*, or *scheduled*, at this step), such that the following two properties hold:

- (1) Each node appears exactly once in the schedule: the sets V_0, \dots, V_{k-1} partition the nodes of g .
- (2) A node is scheduled only after all its ancestors have been: if $n' \in V_i$ and n is an ancestor of n' , then $n \in V_k$ for some $k < i$.

Definition 6. A q -*traversal* of a graph g , for $q \geq 1$, is a parallel traversal such that each step schedules at most q nodes.

Consider a traversal $T = V_0, \dots, V_{k-1}$ of g . A node n of g is *scheduled* prior to a step i in T if it appears in the traversal prior to step i , i.e., $n \in V_0 \cup \dots \cup V_{i-1}$. An unscheduled node n is *ready* at step i in T if all its ancestors (equivalently, all its parents) are scheduled prior to step i . Any sequence $P = V_0, \dots, V_i$, for $i < k$, is a *prefix* of T .

Definition 7. A *greedy q -traversal* T_q of a graph g is a traversal that schedules q

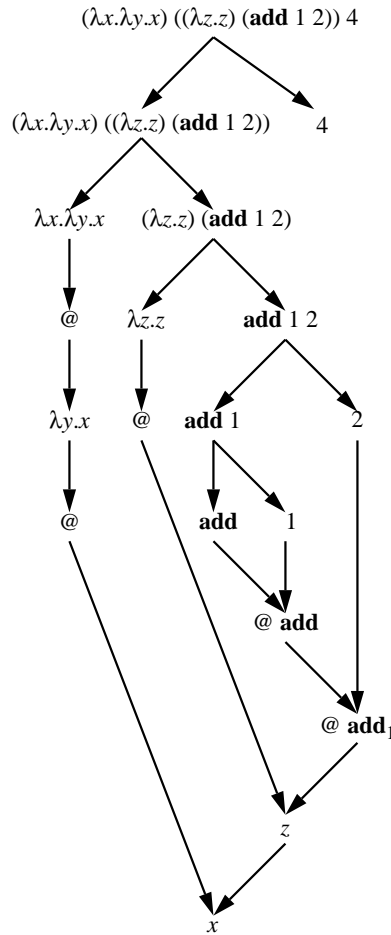


Fig. 24. PSL computation graph for Example 4.

ready nodes on each step (or all the ready nodes when there are fewer than q).

Computation graphs are *dynamically unfolding* in that

- initially, only the source node is revealed;
- when a node is scheduled, its outgoing edges are revealed; and
- when all of the incoming edges of a node are revealed, the node is revealed and available for scheduling.

We consider only online scheduling algorithms for these graphs, i.e., each step’s scheduling decision is based on only the revealed graph nodes and edges. In other words, computation is scheduled at run time, not compile time.

THEOREM 3.2.1. *A greedy q -traversal of graph g takes at most $W(g)/q + D(g)$ steps [Blumofe and Leiserson 1993].*

3.3 Equivalence of the Language and Intermediate Models

This section relates the PSL profiling semantics to the $\text{P-CEK}_{\text{PSLf}}^q$ abstract machine. We show that the machine executes a greedy q -traversal of the computation graph. This provides a bound on the total states processed and the number of steps taken by the machine. This also shows that the models compute the same result, although details of the extensional equivalence are omitted [Greiner 1997].

Since the profiling semantics and abstract machine do not use the same domains, we first define an equivalence between their values and environments in Definition 8.

Definition 8. A PSL value and store v, σ correspond with a $\text{P-CEK}_{\text{PSLf}}^q$ value and store v', σ' ,

$$v, \sigma \approx v', \sigma',$$

if the values are the same constant, or if they are closures with syntactically identical bound variables and expressions and corresponding environments and stores.

A PSL environment and store ρ, σ correspond with a $\text{P-CEK}_{\text{PSLf}}^q$ environment and store ρ', σ' ,

$$\rho, \sigma \approx \rho', \sigma',$$

if they have the same domains, and for each variable x in the environments' domain, $\sigma(\pi_1(\rho(x))), \sigma \approx \sigma'(\pi_1(\rho'(x))), \sigma'$.

To show extensional equivalence between the profiling semantics and the intermediate model (Theorem 3.3.1), we use the more general, but more complicated, Lemma 3.3.1. To allow induction on the structure of a profiling semantics derivation, it includes the following:

- It describes multiple evaluations performed in parallel.
- The context of each evaluation may be any environment and store.
- To describe the evaluation of variables, it requires assumptions about the behavior of any suspended threads.

LEMMA 3.3.1. *If, for some number of expressions $n \geq 1$, and where j ranges over $\{1, \dots, n\}$, and*

- the expressions evaluate in the profiling semantics, for each j ,*

$$\rho_j, \sigma'_j \vdash e_j \xrightarrow{\text{PSL}} v_j, \sigma''_j; g_j,$$

- some step i of the $\text{P-CEK}_{\text{PSLf}}^q$ selects states corresponding to the start of these evaluations, for each j ,*

$$\text{St}A_i = [st_1, \dots, st_n] \# \text{St} \quad st_j = ((e_j, \tau_j), \rho'_j, \kappa_j)$$

for some St , and

- the machine has computed or will compute values corresponding to those in the semantics' environments, for each j , $\text{dom}(\rho_j) = \text{dom}(\rho'_j)$, and for each $x \in \text{FV}(e_j)$, there exists some step m' (before or after i) that starts with store $\sigma_{m'}$ and there exists value v such that $\pi_1(\rho_j(x)), \sigma'_j \approx v, \sigma_{m'+1}$, when the machine calls*

$$\text{throw}(v, \rho'_j(x), \bullet),$$

then the machine computes the corresponding values and reactivates any threads suspended on these states: for each j , for some step $m'_j \geq i$ that starts with store $\sigma_{m'_j+1}$ there exists value v'_j such that $v_j, \sigma'_j \approx v'_j, \sigma_{m'_j}$, when the machine calls

$$\text{throw}(v'_j, \tau_j, \kappa_j).$$

PROOF. We prove this by simultaneous induction on the structure of the profiling semantics derivations. Then for each of the j expressions, we perform the following case analysis on the form of the expression. We apply the induction *in order*, $1, \dots, n$, on the expressions to show that the previous expressions eventually terminate so that any states spawned by st_j are eventually selected, so that induction may be applied to those spawned states.

cases CONST, LAM ($e_j = c$, $e_j = \lambda x.e'$). The corresponding value is computed in a single step ($m'_j = i$).

case VAR ($e_j = x$). By the third assumption, the value of x is computed on step m' . The machine computes the corresponding value in a single step, although it may have to wait for it to be computed ($m'_j = \max(i, m' + 1)$).

case APP ($e_j = e' e''$). As previously described, the two states for evaluating e' and e'' are eventually selected, so that induction may be applied to those evaluations. This gives evaluations terminating on some future steps m' and m'' , respectively. The former results in a state for evaluating $@ l \tau'$, for some location l containing a closure with body e''' and where τ' is the thread for e'' .

At some step after m' , the “@” application state is selected, generating in one step a state to execute e''' . Then at some step after that, the state for e''' is selected, and induction on its semantics derivation applies.

The conclusion then holds, since the value of e''' is the value of the application and, since the thread and continuation of the function body’s initial state are also those of st_j .

case APPC ($e_j = e' e''$). We apply induction on e' and e'' as in the APP case. Evaluation of the resulting “@” application state is also like the APP case, except that it also requires the value of the argument, and thus may have to wait for it to be computed, like the VAR case. That extra condition corresponds to the extra edge in g_j relative to the APP case.

Again the conclusion holds, since the value of the call to δ is the value of the entire application and, since the thread and continuation of the “@” application state are also those of st_j . \square

Theorem 3.3.1 is the special case of extensional equivalence between the profiling semantics and abstract machine about which we are usually concerned.

THEOREM 3.3.1. *If e evaluates in the profiling semantics*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g,$$

then it evaluates in the abstract machine

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSLf}, q} v', \sigma'; Q, \psi$$

such that it computes a corresponding value, $v, \sigma \approx v', \sigma'$.

PROOF. This follows from Lemma 3.3.1 by limiting it to a single profiling semantics derivation of empty context, and by the definition of evaluation in the abstract machine. \square

THEOREM 3.3.2. *If e evaluates in the profiling semantics*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSLf}} v, \sigma; g,$$

then it also evaluates in the abstract machine

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSLf}, q} v', \sigma'; Q, \psi,$$

such that $v, \sigma \approx v', \sigma'$, and the machine executes a greedy q -traversal of g , i.e.,

—the selected states and visited nodes correspond at each step and

—the active states and ready nodes correspond at each step.

PROOF. First, Theorem 3.3.1 shows that the abstract machine computes the same result as the profiling semantics, so in the remainder of this proof, we gloss over the extensional equivalences. To show that computation is a greedy q -traversal, we show by induction on the number of abstract machine steps that the appropriate states and nodes correspond at each step. Let this traversal be $T = V_0, \dots, V_{k-1}$, and let its prefixes be $P_i = V_0, \dots, V_{i-1}$ for $i \in \{0, \dots, k\}$.

For the base case, we observe that the machine starts with a single active state corresponding with the source node of g (which is trivially ready).

Inductively we show that on step i of the P-CEK_{PSLf} ^{q} machine, if

- (1) it starts with active states

$$StA_i = [st_1, \dots, st_{q'}] \uplus StA$$

where $q' = \min(q, |StA_i|)$,

- (2) the active states StA_i correspond 1-1 with the ready states of the prefix $P_i = V_0, \dots, V_{i-1}$ (which also means that the nodes of P_i have been visited),
- (3) it selects states st_1, \dots, st_n , which correspond 1-1 with the nodes of V_i , and
- (4) any node with two parents, one visited prior to this step (in P_i) and one visited on this step (node n , in V_i), corresponds to a suspended state that is reactivated on this step by a state corresponding to n ,

then

- (1) the active states StA_{i+1} correspond 1-1 with the ready states after the prefix P_{i+1} (i.e., any nonsuspending newly created state or reactivating state corresponds 1-1 with a ready state) and
- (2) any state suspending on this step corresponds to a node with two parents, one visited on this step (in V_i , corresponding to the state that created this state) and one not yet visited (in V_{i+1}, \dots, V_{k-1}).

By definition, the unselected states (StA) are part of StA_{i+1} , and the conclusion holds for them by the first assumption. So we perform a case analysis on each of $st_1, \dots, st_{q'}$ to show that one of the two mutually exclusive conclusions holds for each state it creates.

For the case analysis, let $st_j = ((e, \tau), \rho, \kappa)$ (where $j \in \{1, \dots, q'\}$), and let n be the node to which this state corresponds.

case $e = c$. This state corresponds to the unit graph of the CONST rule.

If $\kappa = \bullet$, then n is the second parent node as described in the fourth assumption for zero or more children. This step reactivates those children, adds them to StA_{i+1} , and which by the fourth assumption those children are ready on the next step.

If $c = \text{fun}\langle\tau' \kappa'\rangle$, this step creates a state $((@ c \tau'), \cdot, \kappa')$ which corresponds 1-1 with the unit graph (n') of each possible constant application (*cf.* the APPC rule and Figure 10). Node n' has a second parent for the constant function's argument. If the argument value is not available (that node has not been visited), this created state suspends, and the second conclusion holds by the fourth assumption. If the argument value is available, the created state is added to StA_{i+1} , so the first conclusion holds, since n' is ready.

case $e = x$. This state corresponds to the unit graph of the VAR rule. By definition of the machine, the value is available (since the requesting state would have blocked when first created, if the value was unavailable then) from a state corresponding to n 's second parent.

If $\kappa = \bullet$, the second conclusion follows like the $e = c$ case.

If $\kappa = \text{fun}\langle\tau' \kappa'\rangle$, one of the conclusions follows as in either the $e = c$ or $e = \lambda x.e'$ case, depending on the looked-up value.

case $e = \lambda x.e'$. This state corresponds to the unit graph of the LAM rule.

If $\kappa = \bullet$, the second conclusion follows like the $e = c$ case.

If $\kappa = \text{fun}\langle\tau' \kappa'\rangle$, this step creates a state $((@ l \tau'), \cdot, \kappa')$, for an appropriate l , which corresponds 1-1 with the unit graph (n') in the APP rule. Node n' has only the one parent, so it is ready, and the created state is added to StA_{i+1} , so the first conclusion holds.

case $e = e_1 e_2$. This state corresponds to the source node of the APP rule's graph. This step creates two new states st'_1 and st'_2 , corresponding 1-1 with the two child nodes n_1 and n_2 , respectively, of n .

For each of $j' \in \{1, 2\}$, if the expression in $st'_{j'}$ is a variable, $n'_{j'}$ has a second parent, and the appropriate suspending or nonsuspending conclusion holds as in the second part of the $e = c$ case. Otherwise, $n'_{j'}$ has only the one parent, so it is ready, and the created state is added to StA_{i+1} , so the first conclusion holds.

case $e = @ l \tau'$. This state corresponds to the unit graph of the APP rule's graph. This step creates a state corresponding to the source node of the appropriate function body graph. The conclusion then holds following the same analysis done for the new states of the $e = e_1 e_2$ case.

case $e = @ c \tau'$. This state corresponds to the unit graph of each of the constant function applications in the APPC rule. By definition of the machine, the argument value is available from a start corresponding to n 's second parent.

If $\kappa = \bullet$, the second conclusion follows like the $e = c$ case.

If $\kappa = \text{fun}\langle\tau' \kappa'\rangle$, one of the conclusions follows as in either the $e = c$ or $e = \lambda x.e'$ case, depending on the constant value. \square

COROLLARY 3.3.1. *If e evaluates in the profiling semantics*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g,$$

then it also evaluates in the abstract machine

$$\cdot, \cdot \vdash e \xRightarrow{\text{PSLf}, q} v, \sigma'; Q, \psi$$

such that

- the number of states processed by the machine equals the work of the profiling semantics, $Q = W(g)$, and*
- the number of steps in the machine is bounded by a function of the work and depth of the profiling semantics, $\psi \leq W(g)/q + D(g)$.*

PROOF. The first conclusion follows from the one-to-one correspondence of active states processed and nodes in the graph. The second conclusion follows by Theorem 3.2.1. \square

4. FULLY SPECULATIVE MACHINE MODELS

We now need to simulate the $\text{P-CEK}_{\text{PSLf}}^q$ on each of our machine models. First, Section 4.1 shows how to implement the active states multistack and the sets of suspended states efficiently in parallel, using the fetch-and-add operation. Then, Section 4.2 shows how to implement the rest of the machine efficiently in each of the models.

4.1 Representation of the Active States Multistack and Sets of Suspended States

The multistack (stack, for short) of active states requires three operations:

- creating a new stack at the beginning of an evaluation,
- pushing states onto the stack in parallel, and
- popping states from the stack in parallel.

The sets of suspended threads similarly require three operations:

- creating a new set when creating a new thread,
- adding threads onto multiple sets in parallel, and
- removing all threads from multiple sets in parallel.

We do not have a bound on the maximum size of the stack or sets, so their representations must be able to grow.

For both the stack and the sets, we use an array-based representation for its constant-time lookup and update per element. To grow the data structure, we create a new larger array when necessary, and copy the old elements into the new array. The key to efficiency is to copy infrequently, so that copying does not dominate the cost of using the data structure. The standard technique for this is to double the size of the array each time it grows. The copying is sufficiently infrequent that its cost is amortized to a constant time per step.

The implementation uses an operation called fetch-and-add that allows efficient implementation of the queuing operations on multiple queues at once. Our cost bounds are parameterized by the time cost of this operation, $TF(p)$, as Figure 25

Machine	$TF(p)$ Time for fetch-and-add
Hypercube	$O(\log p)$
EREW PRAM	$O((\log p)^{3/2} / \sqrt{\log \log p})$
CREW PRAM	$O(\log p \log \log p)$
CRCW PRAM	$O(\log p / \log \log p)$

Fig. 25. Time bounds $TF(p)$ for implementing fetch-and-add on randomized machines with p processors.

shows. These bounds hold with high probability [Ranade 1989; Matias and Vishkin 1991; 1995; Gil and Matias 1994].

The stack of active states StA and the queues of suspended states are each implemented by *multithreaded dynamically growing arrays* (MDGAs). An MDGA of states is a pair (m, \vec{st}) of its length m and an array of states St such that

- the array is at least as large as the specified length, $m \leq |\vec{st}|$;
- the rear of the array stores the MDGA’s contents, so that it can grow at the front; data element i of StA is $st_{|\vec{st}|-m+i}$, for each $i \in \{0, \dots, m-1\}$; and
- the array elements are sorted as to age in the array, i.e., elements having been in the array longer are in the rear.

Sets are also implemented by MDGAs. Each operation returns a new pair of the length and a new or modified array.

To initially create an MDGA with one element, create the pair (m, \vec{st}) where $m = 1$, $|\vec{st}| \geq 1$ and where the state is in the last element of the array, i.e., $\vec{st}_{|\vec{st}|-1}$. A larger initial array would delay the need for creating a larger array as the stack grows. This clearly requires constant time and space.

We show that each step of a $P\text{-CEK}_{\text{PSLf}}^{p, TF(p)}$ machine can be implemented in $O(TF(p))$ amortized time, with high probability. Thus the cost bounds of the implementation are parameterized by the cost of fetch-and-add. The amortization comes from how we grow the active state stack. Since we have a bound on the number of steps required by the machine, this allows us to bound the total running time for these machines.

In the push operation, m MDGAs of arbitrary size may need to grow at the same time, and we must parallelize the allocation and copying of all of the relevant data in these arrays. Each processor $i \in \{0, \dots, p-1\}$ has states in an array St_i to add to some MDGA m_i . Clearly there are at most p MDGAs relevant to any given instance of this operation, so $m \leq p$. The state array of MDGA i is labeled St'_i , for $i \in \{0, \dots, m-1\}$. The operation is implemented as follows:

- (1) Compute (using a fetch-and-add) the number of states being added to each MDGA i , $k_i = \text{sum of } |St_j| \text{ such that } m_j = i$, and the total number of original and new states, $k' = \sum_{i=0}^{m-1} |St'_i|$ and $k = \sum_{j=0}^{p-1} |St_j|$. This requires $O(TF(p))$ time and $O(p)$ temporary space for the fetch-and-add operation, plus $O(TS(p))$ times for two add-reduces.
- (2) Increase the size of MDGA arrays, where necessary, as Figure 26 illustrates.
 - (a) Determine which MDGAs need larger arrays, and consider only these for the remainder of this step,

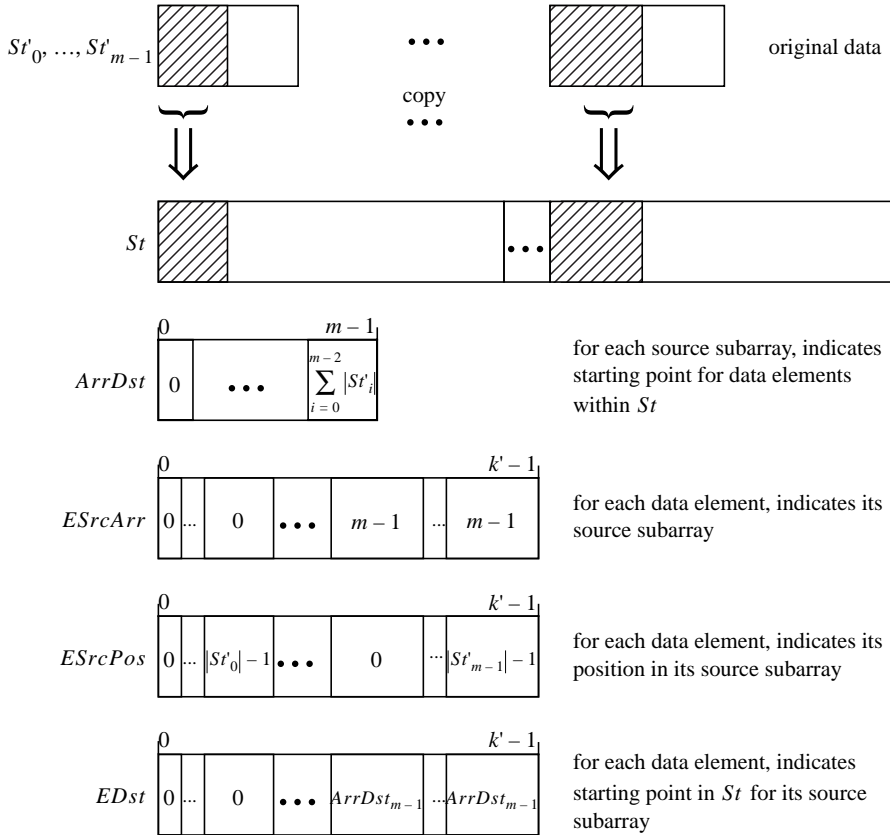


Fig. 26. Step 2 of a push operation on a multithreaded dynamically growing array (MDGA), allocating more space for the original data. There are m original subarrays and k' total original data elements.

- This requires constant time to check the new length of each of the $m \leq p$ MDGAs, and $O(m)$ temporary space to store these lengths.
- (b) Create a single array St such that each MDGA will use a subarray of it. As before, each MDGA allocates twice as much space as its new total number of states. The length of St is then the sum of the total space needed for each of the MDGAs and is computed with an add-reduce. This requires $O(TS(p))$ time, at most $O(k')$ control space, and $O(m)$ temporary space.
 - (c) Each MDGA computes (using an add-scan) the starting point within St for its array, and stores this in $ArrDst$. This requires $O(TS(p))$ time and $O(m)$ temporary space.
 - (d) For each location in array St that receives an old state, record the source of its states. For example, the source for one location might be the 0th element of St'_3 . Thus the sources are stored in an array $ESrcArr$ of MDGA numbers (here, 3) and an array $ESrcPos$ of indices within the corresponding

MDGA's states (here, 0). These are computed by a segmented distribute of \vec{m} and a segmented index, respectively.

This requires $O(k'/p + TS(p))$ time and $O(k')$ temporary space.

- (e) For each location in array St that receives an old state, record the destination of its states. The destinations are stored in an array $EDst$ of offsets into St marking where the new MDGA arrays start and in array $ESrcPos$. The former is computed by using $ESrcArr$ to index into $ArrDst$.

This requires $O(k'/p)$ time and $O(k')$ temporary space.

- (f) Copy the current contents of these arrays into St . Each processor copies a proportional share of the array, using $ESrcArr$ and $ESrcPos$ to index into the appropriate arrays St'_i . For example, the 7th data element is copied from element $ESrcPos_7$ of array $St'_{ESrcArr_7}$ into element $EDst_7 + ESrcPos_7$ of array St .

For each MDGA i , the time for copying each of its $|St'_i|$ elements is counted against the time for initially writing the elements that will be written into the array until the next time it grows. There are at least $|St'_i|$ such elements, since the array doubles in size each time it grows. If the array does not grow again, the cost of this copy operation is counted instead against the initial writing of these elements. Thus, the time for copying data is at most twice that of initially writing data. This requires $O(k'/p)$ amortized time and $O(k')$ space.

From now on, ignore the old arrays for these MDGAs and use the new ones.

Thus this step requires $O(k'/p + TS(p))$ amortized time, $O(k')$ space, and $O(k')$ temporary space.

- (3) Move the new states into the MDGA arrays such that the load is evenly distributed among the processors, as Figure 27 illustrates.

- (a) For each location in the arrays St'_i that receives a new state, record the source of its state. The sources are stored in an array $ESrcArr$ of processor numbers and an array $ESrcPos$ of indices within the corresponding processor's states. These arrays are computed by a segmented distribute of the processor numbers and a segmented index of the lengths of St_i , respectively.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

- (b) For each location in the arrays St'_i that receives a new state, record this destination. The destinations are stored in an array $EDstArr$ of MDGA numbers and an array $EDstPos$ of indices within the corresponding MDGA's states. The former is computed by using $ESrcArr$ to index into \vec{m} , and the latter by segment distributing the MDGA lengths and adding them to $ESrcPos$.

This requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

- (c) Copy the states into arrays St'_i . Each processor copies a proportional share of the data, using the sources and destinations just computed to control the indirect reads and writes, respectively.

This requires constant time per element, or $O(k/p)$ total time and no space.

Thus this step requires $O(k/p + TS(p))$ time and $O(k)$ temporary space.

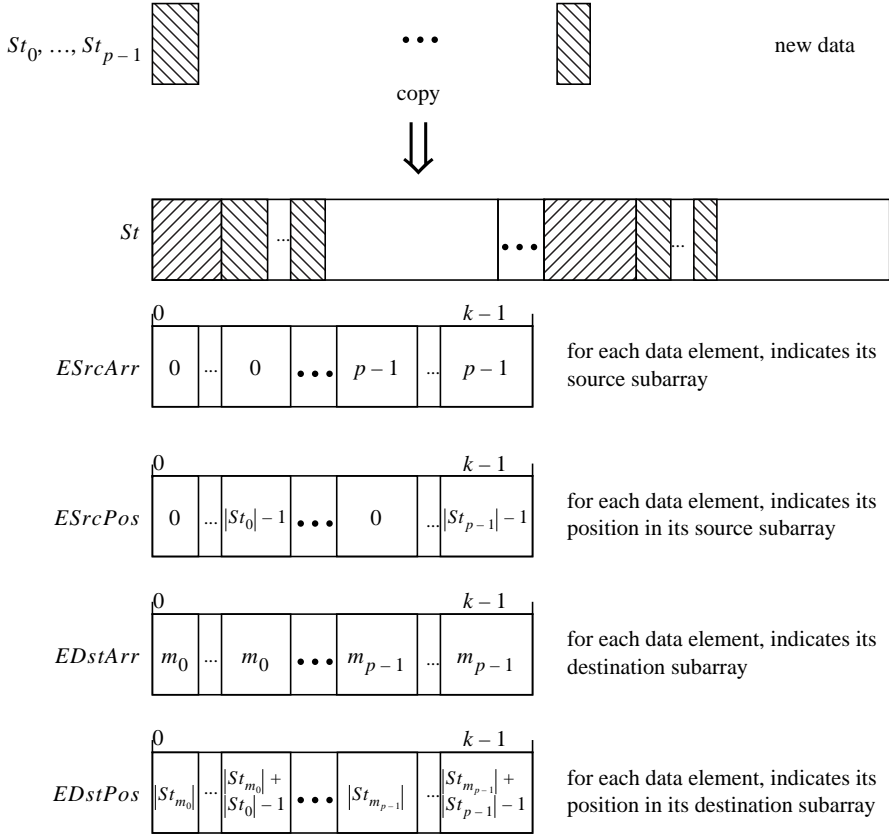


Fig. 27. Step 3 of a push operation on a multithreaded dynamically growing array (MDGA), adding new data to the array St . There are p processors and k total new data elements.

So in total, this requires $O((k' + k)/p + TF(p))$ amortized time and $O(k' + k)$ space for the data, plus $O(p)$ temporary space. This temporary space can be reused in each step.

We remove elements from the data structure when selecting (at most) q states for each step. To pop k states, each processor indexes into the array and grabs the appropriate k/p states. We use a scan operation to assign states to processors and ensure that they are assigned to processors in order (i.e., lower-numbered processors get lower-numbered states). Finally, the stack length is decremented by k . This requires $O(1)$ time and $O(k)$ space.

4.2 Machine Models

Using the basic data structures just described, we now simulate the $P\text{-CEK}_{\text{PSL}_f}^q$ on our machine models. First we examine the time required for each step of the $P\text{-CEK}_{\text{PSL}_f}^q$, then total this for all steps.

THEOREM 4.2.1. *Each $P\text{-CEK}_{\text{PSL}_f}^q$ step can be simulated within $O(q/p + TF(p))$ amortized time on the p processor hypercube and PRAM machine models, with high*

probability.

PROOF. Each processor is responsible for up to q/p elements, i.e., processor i is responsible for the elements $i\lceil q'/p\rceil, \dots, (i+1)\lceil q'/p\rceil - 1$, for $q' = \min(q, |StA|)$. We assume each processor knows its own processor number, so it can calculate a pointer to its section of the array.

The simulation of a step consists of the following:

- (1) locally evaluating the states ($\xrightarrow{\text{PSL}}_{comp}$), and synchronize all processors;
- (2) saving the result value, reactivating the queued states of all finishing states ($\xrightarrow{\text{PSL}}_{block}$), and synchronizing all processors;
- (3) suspending all states requesting to do so ($\xrightarrow{\text{PSL}}_{react}$); and
- (4) creating a new active state stack for the next step.

We now show each of these is executed in the given bounds.

We assume that environment access requires constant time, since the number of variables, and thus the maximum size of any environment, is fixed once given a program. Without this assumption, our time bounds need to be generalized to account for the time for environment accesses and updates, e.g., by representing environments as balanced binary trees, this adds a logarithmic factor in the number of distinct variables in the program [Blelloch and Greiner 1995]. We can then assume that the variables are renamed (e.g., using deBruijn indices) so as to minimize the number of variables.

Locally evaluating the states requires the time it takes to process $k = \lceil q'/p \rceil$ states. The implementation of $\xrightarrow{\text{PSL}}_{comp}$ is straightforward and requires constant time for the various operations. Thus the total time for locally evaluating the states on the machine models of interest is $O(k + TF(p))$, where $TF(p)$ provides an upper bound on any memory latency or space allocations.

In the second substep, each processor has up to k states to finish. Each processor writes its states' results, then returns pointers to their suspended sets. This requires $O(k + TF(p))$ time, including memory latency.

In the third substep, each processor has up to $2k$ states that suspend. The states suspend with a single push operation, requiring $O(k + TF(p))$ amortized time.

The new active state stack is the appending of the newly created and reactivated states to the unselected original active states. There are at most $2q$ new states. On average, there are at most $2q$ reactivated states, since each state is blocked and later reactivated at most once. Thus we amortize, over all steps, the number of states being added to the active states stack. We can push these elements onto the stack in $O(q/p + TF(p))$ amortized time. \square

To account for memory latency in the hypercube, and for the latency in the fetch-and-add operation for all three machines, we process $p \cdot TF(p)$ states on each step instead of just p , i.e., we use a $\text{P-CEK}_{\text{PAL}}^{p \cdot TF(p)}$ machine.

COROLLARY 4.2.1. *Each step of the $\text{P-CEK}_{\text{PSL}}^{p \cdot TF(p)}$ machine can be simulated within $O(TF(p))$ amortized time on the p -processor hypercube and PRAM machine models, with high probability.*

COROLLARY 4.2.2. *If e evaluates in the profiling semantics*

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSL}} v, \sigma; g,$$

then its evaluation in the abstract machine,

$$\cdot, \cdot \vdash e \xrightarrow{\text{PSLf}, p, TF(p)} v, \sigma'; Q, \psi,$$

can be simulated within $O(W(g)/p + D(g)TF(p))$ amortized time on the p -processor hypercube and PRAM machine models, with high probability.

5. PARTIALLY SPECULATIVE IMPLEMENTATIONS

A partially speculative implementation can abort and discard any irrelevant computation. We consider two definitions of relevance, varying in what program's result is treated.

- (1) If the result of the computation is only the semantic value obtained, i.e., a constant or location, the definition is as follows:

A node n of a computation is *relevant* if there is a path from n to the minimum sink nt of the overall graph, i.e., the final value depends on n .

- (2) If the result of the computation is either a constant or the entire data structure referenced from a location, the definition is as follows:

A node n of a computation is *relevant* if there is a path from n to the minimum sink nt of the overall graph, i.e., the final value depends on n , or if n is reachable from the value computed in nt , i.e., it is part of the final result data structure.

Note that both of these definitions are stronger than saying that an application's argument is relevant if its value is used in its function body, since that function body (or that part of it) might not be relevant.

The simplest appropriate modification to the fully speculative implementation is to end on the first iteration that the main thread is done, i.e., when l_{res} has a value. But since the fully speculative implementation does not maintain the states in any particular order, we might be unlucky and schedule all irrelevant computation before the relevant computation. Thus it might make sense to prioritize computations to minimize the amount of irrelevant computation. This might also allow us to detect and discard irrelevant threads during evaluation.

Sections 5.1 and 5.2 discuss some strategies and implementations for prioritizing and aborting threads. Then Section 5.3 discusses the benefits of partial speculation.

5.1 Prioritizing Threads

On each step, we select the (up to) q active states with the highest priority. The goal of prioritizing threads is to minimize the number of irrelevant states used during a computation, so as to reduce the cost of the computation. Since we do not know whether a thread is relevant or not until the computation is done, any priority scheme is either very restricted or based on heuristics. Furthermore, more involved prioritization methods introduce more complicated data structures to store the active states. The cost of prioritizing threads has the potential for overwhelming the cost of evaluation.

The most basic priority scheme distinguishes *necessary* threads, those known to be relevant, from *speculative* threads, those not yet known to be relevant or irrelevant. Necessary threads are given higher priority than speculative threads. The initial thread is immediately necessary, as is any thread spawned from a necessary thread by APPC, or any thread that a necessary thread blocks on. To implement this scheme, we can use two active state stacks, one for each priority, with only a constant factor of overhead (e.g., each newly created or reactivated state checks to which of the two stacks it should be added). This priority scheme was proposed by Baker and Hewitt [1977].

More general priority schemes can be based on the distinguishing degrees of “speculativeness”:

- Threads created by APPC have the same priority as the original parent thread, because they are relevant (or irrelevant) if and only if the original thread is.
- Threads created by APP are more speculative than the original thread, because the original parent thread must be used to communicate this one’s name for it to be used.

The prioritization used by Partridge [Partridge and Dekker 1989; Partridge 1991] is an example of this. If we assume that each speculative child has equal probability to be relevant, then the active states should be kept as a tree, where each node represents active states of equal priority, and each edge represents a speculative child relationship. Selecting the highest-priority threads is removing them from the top of the tree, which seems unlikely to be efficient. But adding new threads can be easily done by adding them in the appropriate places of the tree. Adding reactivated threads is also easy if we remember where they would have been placed when they blocked. Blocking also requires updating the priority of the thread blocked on to the higher of its current priority and the blocking thread’s priority, but comparing two priorities is also unlikely to be faster than logarithmic in the depth of the priority tree.

Further generalizing the scheme to allow arbitrary probabilities of relevance would likely be even more inefficient. Maintaining the accurate thread ordering can dominate the cost of computation, since it can involve touching many additional threads per step. As a simple example, consider storing the threads in order of relevance in an array. Inserting threads involves a sorted merge operation, requiring work linear in the number of currently active threads.

5.2 Aborting Threads

To distinguish *unnecessary* threads, those known to be irrelevant, requires a form of garbage collection on threads. For example, consider the evaluation of an application $e_1 e_2$ resulting in a closure. Even if the function body does not use the argument e_2 , the closure can contain a reference to the argument which is then used by the enclosing context. Only following all the relevant pointers can tell us which threads are no longer accessible, and thus unnecessary.

Several methods of garbage collection of processes has been previously described. Baker and Hewitt [1977] and Hudak and Keller [1982] used a mark-and-sweep approach, which is not asymptotically efficient, since it traverses pointers too many times. Grit and Page [1981] and Partridge [Partridge and Dekker 1989; Partridge

1991] used a reference-counting approach which can be efficient if we do not spend too much effort garbage collecting on each step. We discuss this option in more detail.

For each thread we maintain a count of the references to this thread from environments. The count is one when the thread is created. Counts are incremented when environments are extended, creating a new copy of the environment. Of course, if environments are partially shared to minimize space, the counts should appropriately reflect the sharing. And the appropriate counts are decremented when environments are restricted or threads finish.

When a state is selected at the beginning of a step, we check the count of its thread. If its count is zero, we abort this thread, including decrementing all counts of threads reachable from its environment. But these environments contain values, which in turn contain environments, etc., and we cannot efficiently traverse the entire environment and decrement all these counts at once. So, we use a queue (implemented with an MDGA, but with queue operations) of environment bindings, decrementing $k \cdot q$ reference counts each step, for some constant k .

We can augment this scheme further by observing that all threads spawned by an irrelevant thread (i.e., its *children*) are themselves irrelevant. When a thread is aborted, we also abort its children. To implement this, each thread also keeps a set of pointers to its children. When aborting a thread, the machine sets the counts of its descendants to zero and aborts them. While a given thread can have many more than q descendants, the machine can amortize the cost of aborting them over all the steps, accounting this cost against the cost of creating the threads. Alternatively, we can use a queue of threads to abort, and abort $k \cdot q$ of these on each step, adding the children of any aborted thread onto the queue. This queue of threads could be implemented by an MDGA, although Grit and Page [1981] used a less efficient binary tree data structure for this.

Reference counting is asymptotically efficient, since we only need to change counts for threads that the machine is touching anyway. Thus it involves only a constant factor overhead.

The standard problem with reference counting in garbage collection is accounting for recursive data. Here it works with recursive functions because a recursive closure $\mathbf{cl}(\rho, x, y, e)$ would not be represented recursively. The key to that representation is explicitly naming the closure x and unrolling the recursion only when necessary. The same technique could be applied to circular data structures, e.g., the semantics suggested in the Appendix could be altered so that it created a named pair value, similar to the named closure, that is unrolled when applied to the selectors **fst** or **snd**.

5.3 Cost Benefits of Partial Speculation

When a priority scheme schedules computation well, for some computations it can greatly reduce the number of states processed or steps an evaluation. Clearly any partially speculative implementation should quickly detect that the potentially large computation e in the expression $(\lambda x.1) e$ can be aborted.

Consider the subgraph g_r consisting of only the relevant nodes of a computation graph g . Note that a serial call-by-need implementation requires $W(g_r)$ time to evaluate the computation. Unlike the PSLf, any partially speculative abstract ma-

chine correctly prioritizing necessary computation processes at most $q \cdot W(g_r)$ states and $W(g_r)$ steps, since at least one relevant node is traversed on each step. *This implies that it terminates if the call-by-need implementation does.* But we conjecture that it is possible to construct, for any priority scheme, example computation graphs that the priority scheme does not significantly parallelize even if its relevant subgraph has significant parallelism.

Garbage collecting threads obviously benefits the space cost of evaluation. But it is unclear whether it can improve the worst-case asymptotic space bounds of an evaluation strategy. Consider an irrelevant subcomputation that generates many threads before the main thread of this subcomputation is known to be irrelevant. As the machine aborts these threads, their descendants, since they are not yet known to be irrelevant, can generate new irrelevant threads. The aborting of threads will eventually catch up with the spawning of new irrelevant threads if the machine does both of the following:

- It aborts more threads per step than it creates, e.g., it aborts up to $k \cdot q$ threads per step, for some constant $k > 1$.
- It aborts on at least two levels of the graph each step, e.g., it aborts some threads, which adds those threads' children to the abort queue and then aborts some of these children.

Without the latter condition, spawning could always be a level ahead of aborting, as in Grit and Page's description.

6. RELATED WORK

The PSL model is “speculative” in two senses. First, it is speculatively parallel relative to a call-by-value model (e.g., the PAL model [Blelloch and Greiner 1995; Greiner 1997]), as it allows a function body and argument to be evaluated in parallel when possible, which is consistent with Hudak and Anderson's call-by-speculation [Hudak and Anderson 1987]. Second, it is speculative relative to a call-by-need evaluation, as it at least starts the evaluation of an argument even if it is irrelevant. This contrasts with some descriptions of speculativeness [Osborne 1989; Flanagan and Felleisen 1995; Moreau 1994] that are speculative relative to a call-by-value evaluation. By definition of those descriptions, the parallel execution of a program must be extensionally equivalent to the serial execution, even in the presence of control escapes or side-effects, e.g., when evaluating **let** $x = e_1$ **in** e_2 , if e_1 has an error or escape, then any escapes or side-effects of e_2 's evaluation must be ignored. Thus, e_1 is considered *mandatory*, while e_2 is *speculative*. Similarly, any side-effects in e_1 must occur before any conflicting side-effects in e_2 . Computation of e_1 and e_2 may still be parallelized within these constraints. Our results are still applicable to this alternate view of speculation by simply reversing which evaluations are considered mandatory or speculative.

In Multilisp, any expression can be designated as a future that spawns a thread which may be executed in parallel. If its value is needed and not yet computed, the thread requesting the future's value blocks until the value is available. A future can be explicitly *touched* to force its evaluation and its synchronization with the touching thread. It can also be explicitly aborted—if its value is relevant, this leads to an error. Speculative evaluation is equivalent to designating all expressions as

futures and disallowing touching. Full speculation also disallows aborting futures, whereas partial speculation allows aborting them, but in a safe manner not in the programmer’s control.

Full speculation and *leniency* are essentially the same thing, although the term “leniency” originally implied a specific lack of evaluation ordering [Traub 1988]. Id and pH evaluate all subexpressions fully because they may contain side-effects, although a compiler might optimize cases when this is not necessary.

Graph reduction is one technique for implementing lazy (call-by-need) functional languages. But since lazy evaluation entails an inherent lack of parallelism [Kenaway 1994; Tremblay and Gao 1995], parallel versions of these languages have incorporated partial speculation, compromising on the laziness of the language.

By using computation graphs as our costs, we have been able to simplify the semantics as compared to those by Roe [1990; 1991] and by Greiner and Blleloch [1996]. Similar to here, they included depths in an environment to describe when values had been computed. But they also required the context of a judgment to contain a depth at which the evaluation begins, like threading clocks through the evaluation. Here we accomplish this result by building computation graphs which contain the same information in the connections.

Roe’s semantics suffers from an additional complication resulting from his use of some serial expressions. Each expression results in two depths (or Roe’s “times”): when the value becomes “available” (i.e., the depth of the minimum sink) and when the evaluation has finished traversing the expression. This latter is just the depth of the minimum sink in our model, but to explain the difference in his, consider a pairing expression, (e_1, e_2) . The most natural rule for it in the PSL would express that the two subexpressions start evaluating at the same depth. But using a typical encoding, similar to that for *cons* $e_1 e_2$ in the Appendix, the second subexpression starts at a constant lower depth than the first. This happens in Roe’s model, despite there being a rule specifically for pairing expressions. Described operationally, first evaluation of the first component is started, then the second, and then the pair is created where the component values eventually reside.

Another difference from Roe’s semantics is that he tags every value with the depth at which it becomes available. This is a result of his inclusion of explicit data structures (cons-cells). These tags are subsumed here by

- tagging values in environments with their computation graphs (recall that the encoding of a data structure is a closure, which contains an environment), together with
- the evaluation judgment resulting in a value and its graph.

Flanagan and Felleisen [1995] and Moreau [1994; 1995; 1996] also provided semantics for speculative languages that were augmented with costs. Both used small-step contextual-style operational semantics and included continuations or escapes in the language. Moreau also included side-effects. Each described two measures of the work cost of evaluation: the total and the *mandatory*, i.e., nonspeculative, work. Note that in an expression **let** $x = e_1$ **in** e_2 , they considered e_2 , not e_1 , to be speculative, since e_1 could abort, so that the result of e_2 would not be needed. Neither measured depth, or any related cost, even though both described parallel computation.

Moreau [1995; 1996] uses two similar, but more abstract, machines for speculative computation. The more detailed of the two is similar in form, as it is also based on the CEKS serial machine and consists of a series of steps transforming a collection of states and a store, such that each state contains a control string, environment, and continuation. It has three primary differences from ours:

- it includes side-effects and continuations;
- it does not explain how to schedule threads; and
- it uses one giant set of suspended threads, rather than a queue per thread.

Nikhil introduces the P-RISC [Nikhil 1990] abstract machine for implementing Id, a speculative language. The machine, however, is not meant as a formal model and does not fully define the interprocess communication and the selection of tasks to evaluate. It is a more pragmatic concurrent model designed to reduce communication costs, rather than a synchronous one designed to formally analyze runtime across a whole computation. Aside from the basic idea of having queues of blocked threads, the P-RISC and P-CEK_{PSLf}^q have little in common.

7. DISCUSSION

We have specified a fully speculative implementation of the λ -calculus (and related languages) and proved asymptotic time bounds for several machine models. Our time bounds are good in the sense that they are work-efficient and within a logarithmic factor optimal in terms of time. To obtain these bounds, we introduce fully parallel operations on queues of suspended threads.

We believe that the approach we use is an important step in trying to make it possible for users to better understand the performance of high-level parallel languages without requiring them to understand details of the implementation. As discussed in the introduction, however, the “implementation” as described was optimized to make the proofs easy for the asymptotic case and therefore has many constant-factor overheads that are likely to make it impractical. The introduction mentioned how many of these overheads might be reduced; however, this article certainly does not prove that these lead to fast implementations. Another important step therefore would be an actual implementation with experimental results comparing predicted and actual times.

The standard λ -calculus encodings of many language constructs including data structures (e.g., lists), conditionals, local bindings, and recursion behave as desired under speculative evaluation. Furthermore these encodings only involve constant overhead in work and depth [Greiner 1997] over a direct definition. Here we briefly describe some of these constructs.

Lists. Using the standard encodings of lists into the λ -calculus ensures that they are speculative, as desired. For example, an encoding of the list constructor *cons* is

$$\begin{aligned} \text{cons} &\equiv \lambda x_1. \lambda x_2. \lambda x. x \ x_1 \ x_2 \\ \text{car} &\equiv \lambda x. x \ (\lambda x_1. \lambda x_2. x_1) \\ \text{cdr} &\equiv \lambda x. x \ (\lambda x_1. \lambda x_2. x_2). \end{aligned}$$

$\frac{\rho, \sigma \vdash e_1 \xrightarrow{\text{PSL}} v_1, \sigma_1; g_1 \quad \rho[x \mapsto v_1; g_1], \sigma_1 \vdash e_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; g_2}{\rho, \sigma \vdash \mathbf{slet} \ x = e_1 \ \mathbf{in} \ x_2 \xrightarrow{\text{PSL}} v_2, \sigma_2; \mathbf{1} \oplus g_1 \oplus g_2} \quad (\text{SLET})$
--

Fig. 28. Potential PSL rule for a serial binding expression. This assumes that the definition of expressions is suitably extended.

Then expression *cons* $e_1 \ e_2$ evaluates e_1 and e_2 speculatively and returns a cons-cell in constant work and depth.

Conditionals. Using the standard call-by-value encodings of conditionals and booleans leads to nonspeculative evaluation of conditional branches—only the appropriate branch is evaluated. Here we have

$$\begin{aligned} \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 &\equiv e_1 \ (\lambda x.e_2) \ (\lambda x.e_3) \\ \mathbf{true} &\equiv \lambda x_1.\lambda x_2.x_1 \ 0 \\ \mathbf{false} &\equiv \lambda x_1.\lambda x_2.x_2 \ 0 \end{aligned}$$

where x is a fresh variable. During the execution of a conditional, both of the abstractions encoding the branches are evaluated speculatively. But since they are abstractions, they terminate in one step. Only the appropriate branch, i.e., the body of the corresponding abstraction, is evaluated once the test has been evaluated. An encoding which does not wrap e_2 and e_3 in abstractions would lead to speculative evaluation of both branches, an option offered in some languages [Osborne 1989].

Let. The standard definition

$$\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \equiv (\lambda x.e_2) \ e_1$$

results in the two subexpressions e_1 and e_2 being evaluated in parallel.

Serial Let. Since we include no basic serialization construct in the core of PSL, providing a serializing binding construct, for example, is more difficult. But it can be encoded using a *continuation passing style* (CPS) transformation (e.g., Plotkin [1974]):

$$\mathbf{slet} \ x = e_1 \ \mathbf{in} \ e_2 \equiv \text{CPS}[e_1] \ (\lambda x.e_2)$$

Traditionally used for serial computation, CPS makes the standard serial path of evaluation control explicit. The transformation to CPS introduces additional dependences, so that no significant computation can be performed in parallel under speculative evaluation. Alternatively, we could simply add a special expression with a serial semantics, as in Figure 28.

Recursion. If we consider only recursive functions, the standard translation of **letrec** using the call-by-value *Y-combinator* introduces only constant work and depth overhead to each unrolling of a recursive function.

However, it might make sense to create a subclass of recursive data structures—those that are circular. For example, consider the following recursive definition in an extended speculative language:

$$\mathbf{letrec} \ x = (e_1, x) \ \mathbf{in} \ e_2$$

$\begin{array}{l} \rho', \sigma \vdash e_1 \xrightarrow{\text{PSL}} v_1, \sigma_1; g_1 \\ \rho', \sigma_1[l \mapsto v'; g_1] \vdash e_2 \xrightarrow{\text{PSL}} v, \sigma_2; g_2 \end{array}$ <hr style="width: 50%; margin: 0 auto;"/> $\rho, \sigma \vdash \mathbf{letrec} \ x = \mathbf{cons} \ e_1 \ \mathbf{in} \ e_2 \xrightarrow{\text{PSL}} v, \sigma_2; \mathbf{1} \oplus g_2$ <p style="text-align: center; margin-top: 5px;">where $l \notin \sigma, \rho' = \rho[x \mapsto l; g_1]$</p>	(LETREC-pair)
---	---------------

Fig. 29. Potential PSL rule for creating circular pairs. This assumes that the definition of expressions is suitably extended.

The semantics rule of Figure 29 would result in a circular pair. In this example, the definition returns a location for the pair value in constant work and depth and binds it to x while the pair's components are still evaluating. The pair's second component returns having the value of the pair itself, circularly.

This form of a **letrec** expression can also be encoded in the basic λ -calculus, but its evaluation results in an infinitely long chain of pairs being created. Each pair in this chain is created by a separate thread in finite work and depth, but the overall computation never stops creating new threads for the rest of the chain. In full speculation without explicit recursion, the only way to terminate with circular data structures is to rewrite the program to delay and force the structures' components. In partial speculation without explicit recursion, any irrelevant threads should be eventually aborted, assuming the particular variant of partial speculation allows thread aborting to catch up with thread spawning, as discussed in Section 5.2.

REFERENCES

- BAKER, JR., H. G. AND HEWITT, C. 1977. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages*. SIGPLAN Notices, vol. 12. 55–59.
- BELLELOCH, G., GIBBONS, P., AND MATIAS, Y. 1995. Provably efficient scheduling for languages with fine-grained parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*. 1–12.
- BELLELOCH, G. AND GREINER, J. 1995. Parallelism in sequential functional languages. In *Proceedings 7th International Conference on Functional Programming Languages and Computer Architecture*. 226–237.
- BELLELOCH, G. AND REID-MILLER, M. 1997. Pipelining with futures. In *Proceedings 9th ACM Symposium on Parallel Algorithms and Architectures*. 249–259.
- BELLELOCH, G. E. 1992. NESL: A nested data-parallel language. Tech. Rep. CMU-CS-92-103, Carnegie Mellon University. Jan.
- BELLELOCH, G. E. AND GREINER, J. 1996. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*. 213–225.
- BLUMOFFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. Santa Barbara, CA, 207–216.
- BLUMOFFE, R. D. AND LEISERSON, C. E. 1993. Space-efficient scheduling of multithreaded computations. In *Proceedings 25th ACM Symposium on Theory of Computing*. 362–371.
- CALLAHAN, D. AND SMITH, B. 1990. A future-based parallel language for a general-purpose highly-parallel computer. In *Languages and Compilers for Parallel Computing*, D. Galernter, A. Nicolau, and D. Padua, Eds. Research Monographs in Parallel and Distributed Computing. MIT Press, Chapter 6, 95–113.
- CHANDRA, R., GUPTA, A., AND HENNESSY, J. L. 1990. COOL: a language for parallel programming. In *Languages and Compilers for Parallel Computing*, D. Galernter, A. Nicolau, and D. Padua,

- Eds. Research Monographs in Parallel and Distributed Computing. MIT Press, Chapter 8, 126–148.
- FEELEY, M. 1993. An efficient and general implementation of futures on large scale shared-memory multiprocessors. Ph.D. thesis, Brandeis University.
- FELLEISEN, M. AND FRIEDMAN, D. P. 1987. A calculus for assignments in higher-order languages. In *Proceedings 13th ACM Symposium on Principles of Programming Languages*. 314–325.
- FLANAGAN, C. AND FELLEISEN, M. 1995. The semantics of future and its use in program optimization. In *Proceedings 22nd ACM Symposium on Principles of Programming Languages*. 209–220.
- GIL, J. AND MATIAS, Y. 1994. Fast and efficient simulations among CRCW PRAMs. *Journal of Parallel and Distributed Computing* 23, 2 (Nov.), 135–148.
- GOLDMAN, R. AND GABRIEL, R. P. 1988. Qlisp: Experience and new directions. In *Proceedings ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*. 111–123.
- GOTTLIEB, A., LUBACHEVSKY, B. D., AND RUDOLPH, L. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst.* 5, 2 (Apr.).
- GREINER, J. 1997. Semantics-based parallel cost models and their use in provably efficient implementations. Ph.D. thesis, Carnegie Mellon University.
- GREINER, J. AND BLELLOCH, G. E. 1996. A provably time-efficient parallel implementation of full speculation. In *Proceedings 23rd ACM Symposium on Principles of Programming Languages*. 309–321.
- GRIT, D. H. AND PAGE, R. L. 1981. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Trans. Program. Lang. Syst.* 3, 1 (Jan.), 49–59.
- HALSTEAD, JR., R. H. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct.), 501–538.
- HALSTEAD, JR., R. H. 1989. New ideas in Parallel Lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, T. Ito and R. H. Halstead, Jr., Eds. Number 441 in Lecture Notes in Computer Science. Springer-Verlag, 2–51.
- HUDAK, P. AND ANDERSON, S. 1987. Pomset interpretations of parallel functional programs. In *Proceedings 3rd International Conference on Functional Programming Languages and Computer Architecture*. Number 274 in Lecture Notes in Computer Science. Springer-Verlag, 234–256.
- HUDAK, P. AND KELLER, R. M. 1982. Garbage collection and task deletion in distributed applicative processing systems. In *Proceedings ACM Conference on LISP and Functional Programming*. 168–178.
- HUDAK, P. AND MOHR, E. 1988. Graphinators and the duality of SIMD and MIMD. In *Proceedings ACM Conference on LISP and Functional Programming*. 224–234.
- ITO, T. AND MATSUI, M. 1989. A parallel lisp language PaiLisp and its kernel specification. In *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, T. Ito and R. H. Halstead, Jr., Eds. Number 441 in Lecture Notes in Computer Science. Springer-Verlag, 58–100.
- JOY, M. AND AXFORD, T. 1992. Parallel combinator reduction: Some performance bounds. Tech. Rep. RR210, University of Warwick.
- KENNAWAY, R. 1994. A conflict between call-by-need computation and parallelism (extended abstract). In *Proceedings Conditional Term Rewriting Systems-94*. 247–261.
- KRANZ, D. A., HALSTEAD, JR., R. H., AND MOHR, E. 1989. Mul-T: A high-performance parallel lisp. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*. 81–90.
- LANDIN, P. J. 1964. The mechanical evaluation of expressions. *The Computer Journal* 6, 308–320.
- MATIAS, Y. AND VISHKIN, U. 1991. On parallel hashing and integer sorting. *Journal of Algorithms* 12, 4 (Dec.), 573–606.
- MATIAS, Y. AND VISHKIN, U. 1995. A note on reducing parallel model simulations to integer sorting. In *Proceedings 9th International Parallel Processing Symposium*. 208–212.
- ACM Transactions on Programming Languages and Systems, Vol. 21, No. 2, March 1999.

- MILLER, J. S. 1987. MultiScheme: A parallel processing system based on MIT Scheme. Ph.D. thesis, Massachusetts Institute of Technology.
- MOREAU, L. 1994. The PCKS-machine. an abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming*. Number 788 in Lecture Notes in Computer Science. Springer-Verlag, 424–438.
- MOREAU, L. 1995. The semantics of Scheme with future. Tech. Rep. M95/7, Department of Electronics and Computer Science, University of Southampton.
- MOREAU, L. 1996. The semantics of Scheme with future. In *Proceedings 1st ACM SIGPLAN International Conference on Functional Programming*. 146–156.
- NIKHIL, R. S. 1990. The parallel programming language Id and its compilation for parallel machines. Tech. Rep. Computation Structures Group Memo 313, Massachusetts Institute of Technology. July.
- NIKHIL, R. S. 1991. Id version 90.1 reference manual. Tech. Rep. Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology. July.
- NIKHIL, R. S., ARVIND, HICKS, J., ADITYA, S., AUGUSTSSON, L., MAESSEN, J.-W., AND ZHOU, Y. 1995. pH language reference manual, version 1.0—preliminary. Tech. Rep. Computation Structures Group Memo 369, Laboratory for Computer Science, Massachusetts Institute of Technology. Jan.
- OSBORNE, R. B. 1989. Speculative computation in Multilisp. Ph.D. thesis, Massachusetts Institute of Technology.
- PARIGOT, M. 1988. Programming with proofs: A second order type theory. In *Proceedings 2nd European Symposium on Programming*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 300. Springer-Verlag, 145–159.
- PARTRIDGE, A. S. 1991. Speculative evaluation in parallel implementations of lazy functional languages. Ph.D. thesis, Department of Computer Science, University of Tasmania.
- PARTRIDGE, A. S. AND DEKKER, A. H. 1989. Speculative parallelism in a distributed graph reduction machine. In *Proceedings Hawaii International Conference on System Sciences*. Vol. 2. 771–779.
- PEYTON JONES, S. L. 1989. Parallel implementations of functional programming languages. *The Computer Journal* 32, 2, 175–186.
- PLOTKIN, G. D. 1974. Call-by-name, call-by-value and the λ -calculus. *Theor. Comput. Sci.* 1.
- RANADE, A. G. 1989. Fluent parallel computation. Ph.D. thesis, Yale University, New Haven, CT.
- ROE, P. 1990. Calculating lenient programs' performance. In *Proceedings Functional Programming, Glasgow 1990*, S. L. Peyton Jones, G. Hutton, and C. K. Holst, Eds. Workshops in computing. Springer-Verlag, 227–236.
- ROE, P. 1991. Parallel programming using functional languages. Ph.D. thesis, Department of Computing Science, University of Glasgow.
- ROSENDAHL, M. 1989. Automatic complexity analysis. In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag.
- SANDS, D. 1990. Calculi for time analysis of functional programs. Ph.D. thesis, University of London, Imperial College.
- TRAUB, K. R. 1988. Sequential implementation of lenient programming languages. Ph.D. thesis, Massachusetts Institute of Technology.
- TREMBLAY, G. AND GAO, G. R. 1995. The impact of laziness on parallelism and the limits of strictness analysis. In *Proceedings High Performance Functional Computing*, A. P. W. Bohm and J. T. Feo, Eds. 119–133.

Received August 1997; revised May 1998; accepted June 1998