A Provably Time-Efficient Parallel Implementation of Full Speculation

John Greiner and Guy E. Blelloch
Carnegie Mellon University
{jdg,blelloch}@cs.cmu.edu

Abstract

Speculative evaluation, including leniency and futures, is often used to produce high degrees of parallelism. Existing speculative implementations, however, may serialize computation because of their implementation of queues of suspended threads. We give a provably efficient parallel implementation of a speculative functional language on various machine models. The implementation includes proper parallelization of the necessary queuing operations on suspended threads. Our target machine models are a butterfly network, hypercube, and PRAM. To prove the efficiency of our implementation, we provide a cost model using a profiling semantics and relate the cost model to implementations on the parallel machine models.

1 Introduction

Futures, lenient languages, and several implementations of graph reduction for lazy languages all use speculative evaluation (*call-by-speculation* [15]) 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 function arguments. The evaluation of 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 [13, 14, 27] and MultiScheme [22], the programmer explicitly states what should be evaluated in parallel using the *future* annotation. In lenient languages, such as Id [25] and pH [26], by default all subexpressions can evaluate speculatively. With parallel implementations of lazy graph reduction [30, 18] speculative evaluation is used to overcome the inherent lack of parallelism of laziness [19, 36].

Although call-by-speculation is a powerful mechanism to achieve high degrees of parallelism, with current implementations it can be hard to understand the performance characteristics of a program without a reasonably deep understanding of the implementation. An important cause of this problem is the handling of blocked threads. Most implementations will suspend a blocked thread by placing it on a queue associated with the value it is waiting for, and when the value becomes ready the implementation will reactivate all the threads waiting on the queue [22, 17, 27, 6, 20, 10, 7]. The problem is that all implementations we know of sequentialize these queues, which in turn can fully sequentialize programs that appear to be highly parallel. This will happen when all threads access the same value and get suspended on a single queue. This sort of performance anomaly is, in fact, not unusual in implementations of parallel languages.

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

Figure 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 butterfly has $p \log_2 p$ switches, and the hypercube can communicate over all wires simultaneously (multiport version).

With the aim of avoiding such performance anomalies, we specify a provably time-efficient implementation of call-by-speculation which fully parallelizes the queues of suspended threads as well as the other aspects of the implementation. To make the cost of the source language explicit we use a *profiling semantics* similar to that of Roe [33]. This semantics specifies the cost of a computation in terms of the total *work* it performs and its parallel *depth* (the length of the critical path of sequential dependences). We call this cost-augmented language the *parallel speculative λ-calculus* (PSL). We then specify an implementation of the PSL on various machine models and prove a relationship between the work and depth given by the profiling semantics and the running time on these machines. 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 keep our implementation of call-by-speculation relatively simple to ease the proof of the simulations bounds. As such, and given that the bounds are based on big-O analysis, the implementation is not concerned with constants and therefore not particularly practical. For example, we use a high ratio of communication to computation, aggres-

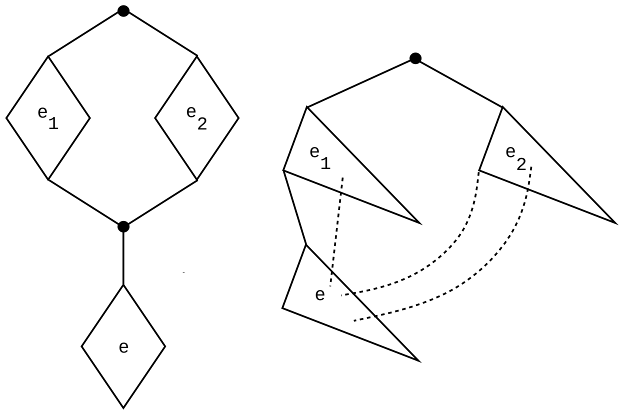

Figure 2: A comparison of the form of dependence graphs for an application $e_1\ e_2$, where e_1 evaluates to $\lambda x.e$. On the left is the graph for parallel call-by-value evaluation and on the right, for call-by-speculation evaluation.

sive sparking of threads, no local caching of values, and no compiler optimizations. In Section 6.1 we discuss how the implementation can be made more practical without effecting the bounds. In particular we believe our implementation of parallel queues is quite practical.

1.1 The model

The language we consider for the paper is the pure λ-calculus with some arithmetic primitives. In Section 2 we describe how most common language features can be added with only constant overheads. In the λ-calculus it is safe to evaluate the two expressions in a function application $e_1\ e_2$ in parallel. In previous work [2] we considered call-by-value parallelism in which e_1 and e_2 are evaluated in parallel to return v_1 and v_2. The processes then synchronize at which point v_1 is applied to v_2. This form of parallelism allows a program to evaluate all the arguments to a function in parallel but must hold off on executing the function body until all arguments are completed.

In call-by-speculation it is assumed that in a function application $e_1\ e_2$ not only can e_1 and e_2 evaluate in parallel, but if e_1 evaluates to $\lambda x.e$, then the body e can continue evaluating in parallel with e_2 (see Figure 2). This allows for a form of pipelined parallelism and can lead to asymptotic improvements in the depth of programs over parallel call-by-value evaluation. On the other hand, call-by-speculation is more difficult to implement since now every value can potentially be a synchronization point. Furthermore since an arbitrary number of threads can access any given value the synchronization might be among a large number of threads. In call-by-value evaluation only pairs of threads can fork and synchronize.

In this paper we distinguish between a fully speculative semantics, in which it is assumed that the body and argument are always evaluated, and a partially speculative semantics for which speculation is limited. We use the term *call-by-speculation* to refer to a fully speculative semantics. Such a semantics will execute the same work as the call-by-value semantics. In this paper our main results are based on call-by-speculation, but in section 6.2 we consider partially speculative implementations that can either kill inaccessible tasks or only selectively execute tasks, thus reducing the work over call-by-value semantics.

1.2 The Problem

We now look at an example of the problem with implementing call-by-speculation. Consider the following code,

```
let y = exp
in pmap (λx.x + y) data
```

which adds the result of expression *exp* to each element of *data*. We assume that `pmap` is some form of parallel map, and that *data* has n elements and is either an array (as with I-structures in Id) or a tree (as could be implemented in Multilisp). In this code if *exp* requires more time to compute than the time to fork the n threads for the parallel map, then all the n 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 by adding it to a queue associated with the variable y (some implementations will spin for a fixed amount of time and then suspend [5]).

In many cases such suspension works well, but the problem with the given code is that a large number of threads will try to suspend on a single variable almost simultaneously. Current implementations sequentialize this process by using a linked list for the queue [22, 17, 6, 24, 7], and therefore would sequentialize the code. We note that a call-by-value semantics would not have this problem since the value of `y` would be computed before executing the `pmap`. Reading the value would require a concurrent read to the location of y, which is assumed in our machine models. Even without direct hardware support for concurrent reads the code would be reasonably well-handled by a cache since only one thread per processor would actually read the value from the shared memory (future threads would find it in the cache).

To avoid this problem we need to be able both to enqueue (when a thread blocks) and dequeue (when the threads are reactivated) in parallel. A potential solution is to use a tree instead of a linked list to represent the queue. The problem with this is that although it would make it easy to dequeue in parallel, it is not clear how to implement the enqueue in parallel. Our solution is based on using a dynamically growing array for the 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 [11, 31]. To account for the cost of growing the array, we amortized it against the cost of originally inserting into the queue.

As well as handling the queues the implementation also has to handle the scheduling of the threads. This is somewhat more complicated than in the in the call-by-value implementation since completing a thread can reactivate an arbitrary number of suspended threads rather than just creating pairs of threads [2].

1.3 Structure of the Implementation

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. We define an intermediary model using an abstract machine called the Fully Speculative Abstract Machine (FSAM). It performs a series of transitions on sets of states, where a state includes an environment, a λ-calculus expression, a continuation, and a queue of suspended threads. We prove that a derivation in the PSL model can be simulated in the FSAM model with only constant overheads in work and depth.

The second half of the PSL simulation is a mapping from the FSAM onto the target machines. At each step of the FSAM, the *active* states are mapped to processors in a load-balanced manner, and the FSAM state transition simulated locally on these states. The primary data structures are queues, with parallel enqueuing and dequeuing, and environments, represented as balanced binary trees.

2 The PSL Model

We define our cost model by adding cost measures to a standard operational semantics of the λ-calculus and call it the Parallel Speculative λ-calculus (PSL). It defines the work, *minimum depth*, and *maximum depth* of an evaluation, which we use in our simulation bounds of Sections 3 and 4. The minimum depth represents the depth at which a handle on the value becomes available, and the maximum depth represents the depth at which all work in the computation is complete. We use the λ-calculus as it represents a minimal language and allows a simple semantics. While it does not directly include many common features of languages, such as data structures and conditionals, we discuss in Sections 2.2 and 2.3 how these can be encoded with constant overheads. It also does not include recursion, and we discuss in Section 2.4 the choice between encoding and explicitly including it in the language and how this affects the costs of a computation.

To gain intuition about the cost measures of a computation we find it helpful to view the computation as a dependence graph (see Figure 3). Each node is a state representing constant work and depth, and each edge in the graph represents either a control or data dependence. The total number of nodes is the work and the shortest path from a node to the root of the graph is its depth. The evaluation of every expression has three important nodes, a source node, and minimum and maximum sink nodes, corresponding to the minimum and maximum depths.

Constants, variables, and λ-expressions require constant time and depth to evaluate, and so correspond to single edges. Graphs of applications have three subgraphs, one for the function, one for the argument, and one for the evaluation of the function body on the argument. There is a dependence from the minimum sink of the function subgraph to the source of the function body subgraph. There can also be data dependence edges from either the function or argument subgraphs to the function body subgraph, which represent the need of the body for a value calculated by those subgraphs.

The standard syntax of the λ-calculus with *constants* is

$$e \quad ::= \quad c \mid x \mid \lambda x.e \mid e_1 \; e_2$$

where x ranges over a countably infinite set of *variables*, and c ranges over a countable set of constants. Without loss of generality, we assume that all constants have an arity of zero or one.

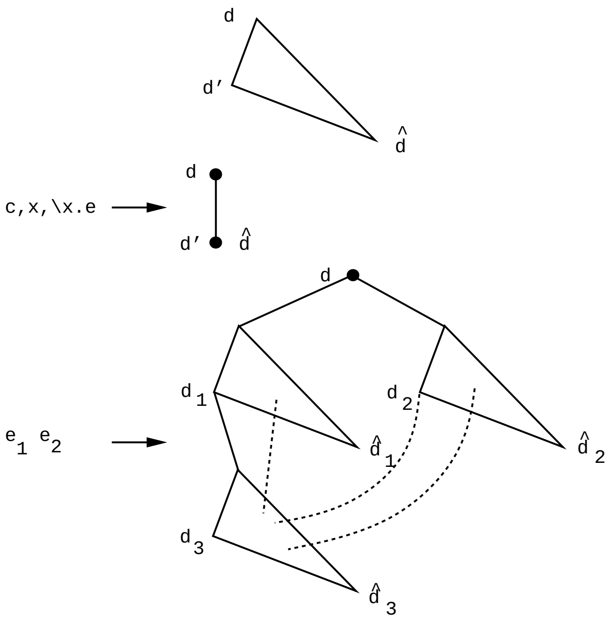

Figure 3: Speculative dependence graphs. At the top is the general form, where the source node is at depth d, the minimum sink node at d', and maximum sink node at $\hat{d}$. For constants, variables, and abstractions, the sink nodes coincide. For general applications, $d' = d_3$ and $\hat{d} = \max(\hat{d}_1, \hat{d}_2, \hat{d}_3)$.

The operational semantics of the language defines the value resulting from the evaluation of a program. Values are either constants or closures:

$$v \quad ::= \quad c \mid cl(E, x, e)$$

An *environment* E is a finite mapping from variables to values and can be the empty mapping $[\,]$ or an environment extended with a new binding $E[x \mapsto v]$.

We augment a standard call-by-value semantics, written $E \vdash e \stackrel{\lambda}{\Longrightarrow} v$, that defines the evaluation of an expression in an environment to a value. We add intensional information which describes the call-by-speculation of our model, defining when the result value of each subexpression is available [33]. To define the minimum depth of a variable, we must tag values in the environment with the depths at which they become available. Costs, tagged values, and tagged environments, respectively, are defined as follows.

$$\begin{array}{rcl} w, d, \hat{d} & ::= & 0 \mid 1 \mid \ldots \mid \infty \\ \dot{v} & ::= & c \mid cl(\dot{E}, x, e) \\ \dot{E} & : & \mathit{Variables} \rightarrow (\mathit{TagValues} \times \mathit{Costs}) \end{array}$$

We assume that the untagged value denoted v and environment denoted E are those obtained by omitting all costs in $\dot{v}$ and $\dot{E}$.

The profiling semantics is given by the relation

$$\dot{E}; d \vdash e \stackrel{\lambda}{\Longrightarrow} \dot{v}; w, d', \hat{d}$$

$$\dot{E}; d \vdash c \overset{\lambda}{\Longrightarrow} c; 1, d+1, d+1 \qquad \text{(CONST)}$$

$$\frac{\dot{E}(x) = \dot{v}; d'}{\dot{E}; d \vdash x \overset{\lambda}{\Longrightarrow} \dot{v}; 1, \max(d, d')+1, \max(d, d')+1} \qquad \text{(VAR)}$$

$$\dot{E}; d \vdash \lambda x.e \overset{\lambda}{\Longrightarrow} cl(\dot{E}, x, e); 1, d+1, d+1 \qquad \text{(ABS)}$$

$$\frac{\begin{array}{c}\dot{E}; d+1 \vdash e_1 \overset{\lambda}{\Longrightarrow} cl(\dot{E}', x, e); w_1, d_1, \hat{d}_1 \\ \dot{E}; d+1 \vdash e_2 \overset{\lambda}{\Longrightarrow} \dot{v}; w_2, d_2, \hat{d}_2 \\ \dot{E}'[x \mapsto \dot{v}; d_2]; d_1 \vdash e \overset{\lambda}{\Longrightarrow} \dot{v}'; w_3, d_3, \hat{d}_3\end{array}}{\begin{array}{l}\dot{E}; d \vdash e_1\ e_2 \overset{\lambda}{\Longrightarrow} \dot{v}'; w_1 + w_2 + w_3 + 1, \\ \qquad d_3, \max(\hat{d}_1, \hat{d}_2, \hat{d}_3)\end{array}} \qquad \text{(APP)}$$

$$\frac{\begin{array}{c}\dot{E}; d+1 \vdash e_1 \overset{\lambda}{\Longrightarrow} c; w_1, d_1, \hat{d}_1 \\ \dot{E}; d+1 \vdash e_2 \overset{\lambda}{\Longrightarrow} \dot{v}; w_2, d_2, \hat{d}_2\end{array}}{\begin{array}{l}\dot{E}; d \vdash e_1\ e_2 \overset{\lambda}{\Longrightarrow} \delta(c, \dot{v}); w_1 + w_2 + 2, \\ \qquad \max(d_1, d_2)+1, \\ \qquad \max(\hat{d}_1, \hat{d}_2)+1\end{array}} \qquad \text{(APPC)}$$

Figure 4: Profiling semantics of the PSL model. The dotted environments and values reflect the fact that the model tags values in environments with cost information. Compare APP to the corresponding DAG diagram.

defined in Figure 4. Since it augments the operational semantics, it holds only if $E \vdash e \overset{\lambda}{\Longrightarrow} v$. Evaluation begins at depth d, a handle on $\dot{v}$ is available at depth d', and e finishes evaluation at depth $\hat{d}$. We are primarily interested in the costs of a program e when evaluated in the empty environment starting at zero depth, *i.e.*, $[]; 0 \vdash e \overset{\lambda}{\Longrightarrow} \dot{v}; w, d', \hat{d}$. Since each inference rule of the PSL semantics introduces only constant overheads, the judgments in a PSL evaluation derivation correspond to the nodes of the dependence graph. These constant overheads also anticipate the overheads of the abstract machine of Section 3.

The minimum and maximum depths of a computation are generally not equal for two reasons. First, if the result value is a closure, this closure is returned without waiting for values in its environments to finish evaluation. For example, using any standard λ-calculus encoding of lists, a *cons* operation returns a cons-cell while its components are possibly still evaluating. The minimum depth tracks when the cell is returned, whereas the maximum depth tracks when its components are also done. Second, evaluation need not wait for other computations which are irrelevant to this one. For example, evaluation $(\lambda x.0)\ e$ returns a value after constant depth, regardless of e. The minimum depth of the whole computation is independent of that of e, but the maximum depth is not. In particular, the minimum depth of this application is finite even if its work and maximum depth are not (*i.e.*, if e does not terminate).

Now consider each inference rule in Figure 4. We assume that evaluating constants and abstractions requires unit work and depth. We also assume that looking up a variable in an environment requires unit work and depth, but the lookup cannot happen until the minimum depth at which the root of the value is available. (We will account for the costs of environment lookups later.) An application $e_1\ e_2$ leads to speculative evaluation in that the function and argument start evaluating at the same time, even if the argument is never used. These subcomputations start on the next step since they correspond to child nodes in the computation graph. If e_1 evaluates to a closure, then the evaluation of the function body begins once e_1 is evaluated, even if e_2 has not finished evaluating. The argument's minimum depth is only included in that of the application if the argument is used, whereas its maximum depth is always included in that of the application. So, its minimum depth is stored in the relevant environment and accounted for in VAR rule as needed. Since we eventually evaluate the argument regardless of whether it is needed, the total work is the sum of all of its subcomputations, plus constant overhead. If e_1 evaluates to a constant function, then we use the function δ to apply it.

$$\begin{array}{lcl@{\qquad}lcl}
\delta(\mathbf{add}, i) & = & \mathbf{add}_i & \delta(\mathbf{add}_i, i') & = & i + i' \\
\delta(\mathbf{mul}, i) & = & \mathbf{mul}_i & \delta(\mathbf{mul}_i, i') & = & i \times i' \\
\delta(\mathbf{div}, i) & = & \mathbf{div}_i & \delta(\mathbf{div}_i, i') & = & \lfloor i/i' \rfloor \\
\delta(\mathbf{neg}, i) & = & -i & \delta(<, i) & = & <_i \\
\delta(<_i, i') & = & \multicolumn{4}{l}{\textit{if } i < i' \textit{ then } cl([], x, \lambda x'.x)} \\
 & & \multicolumn{4}{l}{\qquad\qquad \textit{else } cl([], x, \lambda x'.x')}
\end{array}$$

Figure 5: Example arithmetic constants.

2.1 Arithmetic constants

So far, we have considered no specific constants in the model. However, for the sake of practicality, we would use a set of arithmetic constants such as

$$\begin{array}{rcl} c & ::= & i \mid \mathbf{add} \mid \mathbf{mul} \mid \mathbf{neg} \mid \mathbf{div} \mid < \mid \\ & & \mathbf{add}_i \mid \mathbf{mul}_i \mid \mathbf{div}_i \mid <_i \end{array}$$

where i ranges over the integers. The primitive functions are addition, multiplication, division, negation, and the less-than comparison, and for syntactic simplicity, all primitive functions are curried, as defined in Figure 5. Since we neither include pairs nor binary application in the the language (for brevity), we use constants that represent partially applied constants, *i.e.*, a binary function applied to its first argument. Also, the result of a comparison is an encoding of "true" or "false", since we have not included booleans.

2.2 Data Structures

Using standard data structure encodings into the λ-calculus ensures that these data structures are also speculative. For example, an encoding of the list constructor *cons* is

$$\begin{array}{rcl} \textit{cons} & \equiv & \lambda x_1.\lambda x_2.\lambda x.x\ x_1\ x_2 \\ \textit{car} & \equiv & \lambda x.x\ (\lambda x_1.\lambda x_2.x_1) \\ \textit{cdr} & \equiv & \lambda x.x\ (\lambda x_1.\lambda x_2.x_2) \end{array}$$

Then expression *cons* $e_1\ e_2$ evaluates e_1 and e_2 speculatively and returns a cons-cell in constant work and depth. This encoding is equivalent to including the inference rules

$$\frac{\begin{array}{c}\dot{E}; d+2 \vdash e_1 \stackrel{\lambda}{\Longrightarrow} v_1; w_1, d_1, \hat{d}_1 \\ \dot{E}; d+1 \vdash e_2 \stackrel{\lambda}{\Longrightarrow} v_2; w_2, d_2, \hat{d}_2\end{array}}{\begin{array}{l}\dot{E}; d \vdash cons\ e_1\ e_2 \stackrel{\lambda}{\Longrightarrow} \\ \langle v_1; d_1, v_2; d_2\rangle; \\ w_1 + w_2 + 5, d+5, \max(d+5, \hat{d}_1, \hat{d}_2)\end{array}} \quad \text{(CONS)}$$

$$\frac{\dot{E}; d+1 \vdash e \stackrel{\lambda}{\Longrightarrow} \langle v_1; d_1, v_2; d_2\rangle; w, d', \hat{d}}{\begin{array}{l}\dot{E}; d \vdash car\ e \stackrel{\lambda}{\Longrightarrow} \\ v_1; w+12, \\ \quad \max(d+9, d_1+2, d'+6), \\ \quad \max(d+9, d_1+2, d_2+1, d'+6, \hat{d})\end{array}} \quad \text{(CAR)}$$

plus a similar CDR rule, where we use the following encoding for a cons cell:

$$\langle v_1; d_1, v_2; d_2\rangle \equiv cl([x_1 \mapsto v_1; d_1, x_2 \mapsto v_1; d_2], x, x\ x_1\ x_2)$$

2.3 Other language constructs

The standard λ-calculus encodings of many other language constructs behave as desired under speculative evaluation. Here we briefly describe conditionals and local bindings.

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

$$\begin{array}{rcl}\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \equiv & e_1\ \lambda x.e_2\ \lambda x.e_3 \\ \textbf{true} & \equiv & \lambda x_1.\lambda x_2.x_1 \\ \textbf{false} & \equiv & \lambda x_1.\lambda x_2.x_2\end{array}$$

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 [27].

Similarly, the standard definition of local binding:

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

produces speculative evaluation of both expressions as in some languages [33]. A serialized equivalent can be encoded using a continuation passing style transform, $CPS[e_1]\ \lambda x.e_2$. Or we could add a special expression and inference rule such as

$$\frac{\begin{array}{c}\dot{E}; d+1 \vdash e_1 \stackrel{\lambda}{\Longrightarrow} v_1; w_1, d_1, \hat{d}_1 \\ \dot{E}[x \mapsto v_1]; d_1+1 \vdash e_2 \stackrel{\lambda}{\Longrightarrow} v_2; w_2, d_2, \hat{d}_2\end{array}}{\begin{array}{l}\dot{E}; d \vdash \textbf{slet } x = e_1 \textbf{ in } x_2 \stackrel{\lambda}{\Longrightarrow} \\ x_2; w_1+w_2+1, d_2, \max(\hat{d}_1, \hat{d}_2)\end{array}} \quad \text{(SLET)}$$

Note how the body does not begin evaluation until the bound expression's value is available (by the use of d_1).

2.4 Recursion

There are two ways we can define recursion, either by encoding it within the basic λ-calculus, or by adding an additional construct. This distinction is asymptotically important for the definition of costs of recursive data structures. Providing an explicitly recursive language construct can be favorable in terms of program costs.

Consider the following recursive definition:

$$\textbf{let } x = cons\ e_1\ x \textbf{ in } e_2$$

With call-by-speculation, it is natural for this to create a circular list, rather than one infinitely long, since its definition returns a cons-cell (in constant work and depth) and binds it to x while the cell's components are evaluating, delaying where necessary until x's value is available. We would prefer to define that such recursive definitions create circular data structures in constant work and depth, rather than creating infinitely long data structures in infinite work and maximum depth. (Note that even with an infinite data structure, each component is available in finite minimum depth.)

Encoding recursion with the call-by-value least fixed-point combinator Y results in infinite data structures. Existing lenient languages allow circular data structures, and thus require an explicitly recursive construct. Without explicit recursion, the only way to terminate with such data structures it to rewrite the program to delay and force the structures' components.

Adding an explicitly recursive operator to the semantics would require the use of stores. Such a change would be straightforward in the PSL, but would add to rule verbosity. In the underlying FSAM model (Section 3), stores are needed anyway, and adding recursion would be simple.

3 The Fully Speculative Abstract Machine

We now examine how to implement the PSL on other machine models. As an intermediate step, we introduce the Fully Speculative Abstract Machine (FSAM), based loosely on the P-ECD machine [2]. It executes a sequence of steps in parallel over a sets of states. Each thread of computation is represented by a series of states over time. We prove the following relations between the costs in the PSL and FSAM models:

- The total number of states processed is $O(w)$.
- The number of steps until the original thread of the program finishes is $O(d)$.
- The number of steps until all the computation finishes is $O(\hat{d})$.

A state s is a tuple $(\ddot{E}, e, \kappa, r)$ consisting of

- an environment $\ddot{E}$ mapping each variable to the result of the state computing that value,
- an expression e to be evaluated,
- a *continuation* κ listing the result locations for the arguments of e,
- *result locations* r, a pair of *locations* containing
  - the state's value, or NoValue if the value has not yet been computed, and
  - a queue of all the (inactive) states suspended on this one.

(The projection functions π_1 and π_2 are used to obtain the individual locations from a pair.)

We assume there is a countably infinite set of locations. We will thread a *store* through the computation to map each location to a value or a queue of states.

The semantic domains used for the FSAM are slightly different than those for the PSL. For convenience, we add an additional form of expression to represent the equivalent of a function body for a constant function application. The expression @ $c\ x$ represents c applied to the value to which x is bound. The FSAM uses another slightly different form of values and environments:

$$\begin{array}{rcl} \ddot{v} & ::= & c \mid cl(\ddot{E}, x, e) \\ \ddot{E} & : & \mathit{Variables} \to \mathit{ResultLocs} \end{array}$$

As before, we assume that v and E are the untagged equivalents of $\ddot{v}$ and $\ddot{E}$.

A single step of the machine is written $A, \sigma \stackrel{F}{\Longrightarrow} A', \sigma'$, where A is the current set of active states, and σ is the current store. The machine starts with one state that represents the entire program and a store containing that state's result locations. When there are no remaining active states, the machine finishes and the final value is stored in the result locations of the original state. In addition, each step may update the result locations of some states. Over time, the number of active states may decrease as threads finish or suspend, or increase as threads are created or reactivated.

We ensure that no currently active states have the same result locations. The series, over time, of states which do share the same result locations represents a thread of computation. Thus we use the pair of result locations as this thread's identifier. Also, while we generally describe the FSAM as manipulating states, we can equivalently describe it in terms of threads. While evaluating an application, the current thread evaluates the function and then the function body, while a new thread evaluates the argument.

The basic ideas of each step are as follows. If a state's expression is a constant, we immediately have its value (the constant itself). So we check its continuation to see if it needs to be applied to an argument. If so, we start the evaluation of the application; otherwise, we store the constant as the state's result value and reactivate anything blocked on this. If the expression is an abstraction, we build the appropriate closure, and similarly finish or apply, depending on the continuation. If it is a variable, we look up its value. If the value is available, we finish or apply as appropriate; otherwise we suspend this thread. The same state will be reactivated when the value is available. If it is an application, the current thread will evaluate the function, and we fork a new thread to evaluate the argument. And if it is a constant function body, we try to lookup the variable, apply δ if the variable's value is available, and then finish or apply.

On each step, each active state uses three substeps as defined in Figure 6. The first substep, $\stackrel{F}{\Longrightarrow}_1$, is the core of the evaluation of each active state. It performs the case analysis on a state expression just outlined, using the *fin_app* routine to check if a state finishes with a value or applies it to an argument. For each active state, it results in one of

- $\mathsf{States}(A, \sigma)$, with a set of new states to be active on the next step and new store bindings,
- $\mathsf{Susp}(s, l)$, indicating that the state needs to be added to the queue at location l, or
- $\mathsf{Fin}(\ddot{v}, r)$, indicating that the thread has finished with the given value and result locations.

The second substep, $\stackrel{F}{\Longrightarrow}_2$, places all of the suspending states on the appropriate queues. The third, $\stackrel{F}{\Longrightarrow}_3$, saves the final value and reactivates the blocked threads of all the finishing states. The new and the reactivated states are taken as the result of the step $\stackrel{F}{\Longrightarrow}$ as a whole and are the active states of the next step.

Synchronization between the last two substeps is necessary for correctness. Without it, a state could be added to a queue of suspended states after that queue is reactivated, leaving the state suspended forever. Additionally, the synchronization is also useful for cost predictability, as otherwise an arbitrarily long series of values could become available and used in a single step. For example, for any k, and all i in $\{1, \ldots, k\}$, if

$$s_i = ([x_i \mapsto r_{i+1}], @\ c_i\ x_i, [\,], r_i)$$

the states $s_1, \ldots, s_k$ could evaluate in a single step.

Definition 1 *The* FSAM evaluates e to $\ddot{v}$ with w work, d minimum steps, and $\hat{d}$ maximum steps, *or* $e \stackrel{F}{\Longrightarrow}^* \ddot{v}; w, d, \hat{d}$, *if $\hat{d}$ is the minimum step such that*

$$A_0, \sigma_0\ (\stackrel{F}{\Longrightarrow} \circ \cdots \circ \stackrel{F}{\Longrightarrow})\ A_{\hat{d}}, \sigma_{\hat{d}}$$

where

- *it starts with one active state to evaluate e:*

$$\begin{array}{rcl} A_0 & = & \{([\,], e, [\,], r)\} \\ \sigma_0(\pi_1 r) & = & \mathsf{NoValue} \\ \sigma_0(\pi_2 r) & = & \mathit{emptyq} \end{array}$$

- *it finishes with no active states:*

$$A_{\hat{d}} = \{\}$$

- *the initial thread's value is ready at step d:* *for all $i \geq d$, $\sigma_i(\pi_1 r) = \ddot{v}$, and*
- *the number of active states processed is the amount of work:* $\sum_{i=0}^{\hat{d}-1} |A_i| = w$, *where $|A|$ is the size of A.*

Thus, for a given step, each active state requires constant work.

Equivalence of the PSL Model and the FSAM

We prove that the PSL and FSAM models compute the same value with asymptotically equivalent costs. To achieve this, we must show that the accounting of work and depth in the PSL model accurately counts the steps involved in the corresponding FSAM evaluation. However, whereas the FSAM explicitly deals with blocking and thus reflects its costs, the PSL model does not. Since each thread can block at most once, the FSAM can spend at most half of its work blocking. Thus it affects the complexity of the simulation only by a constant factor. But to simplify the proof of equivalence, we modify the PSL model to explicitly account for the

$$
\begin{array}{llll}
\overbrace{(\ddot{E},\; c,\; \kappa,\; r)}^{s},\ \sigma & \stackrel{F}{\Longrightarrow}_1 & \mathit{fin\_app}\ c\ \kappa\ r & \text{constant} \\
(\ddot{E},\; x,\; \kappa,\; r),\ \sigma & \stackrel{F}{\Longrightarrow}_1 & \mathit{case}\ \sigma(\pi_1(\ddot{E}(x)))\ \mathit{of} & \text{variable} \\
 & & \quad \mathsf{NoValue} \Rightarrow \mathsf{Susp}(s, \pi_2(\ddot{E}(x))) & \\
 & & \quad \ddot{v} \qquad\quad\ \Rightarrow \mathit{fin\_app}\ \ddot{v}\ \kappa\ r & \\
(\ddot{E},\; \lambda x.e,\; \kappa,\; r),\ \sigma & \stackrel{F}{\Longrightarrow}_1 & \mathit{fin\_app}\ \mathit{cl}(\ddot{E}, x, e)\ \kappa\ r & \text{abstraction} \\
(\ddot{E},\; e_1\ e_2,\; \kappa,\; r),\ \sigma & \stackrel{F}{\Longrightarrow}_1 & \mathit{let}\ r' = (\mathit{fresh}\ l, \mathit{fresh}\ l') & \text{application} \\
 & & \mathit{in}\ \mathsf{States}(\{(\ddot{E}, e_1, r' :: \kappa, r), (\ddot{E}, e_2, [\,], r')\}, & \\
 & & \qquad [l \mapsto \mathsf{NoValue}, l' \mapsto \mathit{emptyq}]) & \\
(\ddot{E},\; @\ c\ x,\; \kappa,\; r),\ \sigma & \stackrel{F}{\Longrightarrow}_1 & \mathit{case}\ \sigma(\pi_1(\ddot{E}(x)))\ \mathit{of} & \text{constant application} \\
 & & \quad \mathsf{NoValue} \Rightarrow \mathsf{Susp}(s, \pi_2(\ddot{E}(x))) & \\
 & & \quad \ddot{v} \qquad\quad\ \Rightarrow \mathit{fin\_app}\ \delta(c, \ddot{v})\ \kappa\ r &
\end{array}
$$

$$
\begin{array}{lllll}
\text{where} & \mathit{fin\_app}\ \ddot{v} & [\,] & r & = \mathsf{Fin}(\ddot{v}, r) \\
 & \mathit{fin\_app}\ \mathit{cl}(\ddot{E}', x, e) & (r' :: \kappa) & r & = \mathsf{States}(\{(\ddot{E}'[x \mapsto r'], e, \kappa, r)\}, [\,]) \\
 & \mathit{fin\_app}\ c & (r' :: \kappa) & r & = \mathsf{States}(\{([x \mapsto r'], @\ c\ x, \kappa, r)\}, [\,])
\end{array}
$$

$$
\{\mathsf{Susp}(s_1, l_1), \ldots, \mathsf{Susp}(s_n, l_n)\}, \sigma \quad \stackrel{F}{\Longrightarrow}_2 \quad \begin{array}{l} \sigma[l'_1 \mapsto \mathit{enqueue}\ \{s_i | l_i = l'_1\}\ \sigma(l'_1), \ldots, \\ \quad l'_m \mapsto \mathit{enqueue}\ \{s_i | l_i = l'_m\}\ \sigma(l'_m)] \end{array}
$$

where $l'_1, \ldots, l'_m$ are the distinct locations in $l_1, \ldots, l_n$

$$
\{\mathsf{Fin}(\ddot{v}_1, r_1), \ldots, \mathsf{Fin}(\ddot{v}_n, r_n)\}, \sigma \quad \stackrel{F}{\Longrightarrow}_3 \quad \left(\bigcup_{i=1}^{n} \mathit{dequeue\_all}\ (\pi_2 r_i)\right), \sigma[\pi_1 r_1 \mapsto \ddot{v}_1, \ldots, \pi_1 r_n \mapsto \ddot{v}_n]
$$

$$
\{s_1, \ldots, s_n\}, \sigma \quad \stackrel{F}{\Longrightarrow} \quad A \cup \left(\bigcup_{i=1}^{n} A_i\right), \sigma'' \cup \left(\bigcup_{i=1}^{n} \sigma_i\right) \ \text{if} \quad \begin{array}{rcl} s_i, \sigma & \stackrel{F}{\Longrightarrow}_1 & X_i \\ \{X_i | X_i = \mathsf{Susp}(\cdot,\cdot)\}, \sigma & \stackrel{F}{\Longrightarrow}_2 & \sigma' \\ \{X_i | X_i = \mathsf{Fin}(\cdot,\cdot)\}, \sigma' & \stackrel{F}{\Longrightarrow}_3 & A, \sigma'' \end{array}
$$

where $\{\mathsf{States}(A_1, \sigma_1), \ldots, \mathsf{States}(A_n, \sigma_n)\} = \{X_i | X_i = \mathsf{States}(\cdot,\cdot)\}$

Figure 6: Fully speculative substeps. Each step of evaluation $\stackrel{F}{\Longrightarrow}$ consists of using the three substeps $\stackrel{F}{\Longrightarrow}_1$, $\stackrel{F}{\Longrightarrow}_2$, and $\stackrel{F}{\Longrightarrow}_3$, synchronizing between them.

$$\frac{\dot{E}(x) = \dot{v}; d'}{\dot{E}; d \vdash x \stackrel{\lambda'}{\Longrightarrow} \dot{v}; o+1, \max(d, d')+o+1, \max(d, d')+o+1} \quad \text{(VAR')}$$

where $o = one(d' > d)$

$$\frac{\dot{E}; d+1 \vdash e_1 \stackrel{\lambda'}{\Longrightarrow} c; w_1, d_1, \hat{d}_1 \qquad \dot{E}; d+1 \vdash e_2 \stackrel{\lambda'}{\Longrightarrow} \dot{v}; w_2, d_2, \hat{d}_2}{\dot{E}; d \vdash e_1\ e_2 \stackrel{\lambda'}{\Longrightarrow} \delta(c, \dot{v}); w_1+w_2+o+2, \max(d_1, d_2)+o+1, \max(\hat{d}_1, \hat{d}_2)+o+1} \quad \text{(APPC')}$$

where $o = one(d_2 > d_1)$

where $one(b) = 1$ if b, 0 otherwise

Figure 7: Modified profiling semantics of the blocking-PSL model accounts for the higher constant costs when threads block. The CONST', ABS', and APP' rules are like the corresponding rules of the PSL model.

costs of blocking, so that its costs are *exactly* equivalent to those of the FSAM. This modified semantics, the blocking-PSL model (Figure 7), differs from the PSL model only in that the constant overheads in VAR' and APPC' account for blocking. The exact cost equivalence of the blocking-PSL and FSAM (Corollary 1) leads directly to the asymptotic equivalence of the PSL and the FSAM (Corollary 2). Each of these follow from Theorem 1 which generalizes the equivalence to individual subcomputations of the blocking-PSL and FSAM models. As part of this, we formalize the notion of subcomputations within the FSAM by defining the children and descendents of states (Definition 2). A state's descendents are all those required for the computation begun by the state.

Definition 2 *If $\{s\}, \sigma \stackrel{F}{\Longrightarrow} A, \sigma'$ is one step of a FSAM evaluation, then each of the states in A is a* child *of s. From this relation we also define the* descendents *of a state in the obvious way (such that a state is a descendent of itself).*

Theorem 1 *If*

1. $[x_1 \mapsto \dot{v}_1; d_1, \ldots, x_n \mapsto \dot{v}_n; d_n]; d \vdash e \stackrel{\lambda'}{\Longrightarrow} \dot{v}; w, d', \hat{d}$,
2. $A_d, \sigma_d \ (\stackrel{F}{\Longrightarrow} \circ \cdots \circ \stackrel{F}{\Longrightarrow})\ A_{\hat{d}}, \sigma_{\hat{d}}$, *where*
   - *at step d, there is an active state to start the evaluation of e that has an environment corresponding to that in the blocking-PSL model:*
     $$\begin{aligned} A_d &= A \cup \{s\} \\ s &= ([x_1 \mapsto r_1, \ldots, x_n \mapsto r_n], e, \kappa, r) \end{aligned}$$
   - *the threads computing the environments' contents will obtain values at the indicated depths:*
     for all $r_i \neq r$ and $j > d_i$, $\sigma_j(\pi_1 r_i) = \ddot{v}_i$,
     (The restriction that $r_i \neq r$ ensures that we do not assume part of the conclusion.)

then

1. *the FSAM finishes evaluating e and either stores its value or starts its application by step d':*
   at step $d'-1$, fin_app is called on r, i.e.,
   - *if $\kappa = []$, then $\sigma_{d'-1}(\pi_2\ r) \subseteq A_{d'}$, and for $i \geq d'$, $\sigma_i(\pi_1 r) = \ddot{v}$*
   - *if $\kappa = r' :: \kappa'$ and $\dot{v} = c$, then*
     $$([x \mapsto r'], @\ c\ x, \kappa', r) \in A_{d'}$$
   - *if $\kappa = r' :: \kappa'$ and $\dot{v} = cl(\dot{E}', x, e')$, then*
     $$(\ddot{E}'[x \mapsto r'], e', \kappa', r) \in A_{d'}$$
2. *the computation for this expression requires the indicated work and maximum depth:*
   the sum over all $\hat{d}$ steps of the number of active states that are descendents of s is w, and for all $i > \hat{d}$, A'_i contains no descendents of s.

Proof:

cases CONST', $e = c$, and ABS', $e = \lambda x.e'$: By the appropriate inference rule, $w = 1$ and $d' = \hat{d} = d + 1$. The FSAM fully evaluates e to the appropriate value in one step of unit work, calling *fin_app* on step d. No descendents of s are created, and the conclusion holds.

case VAR', $e = x_i$: There are two subcases, depending on whether s blocks on this lookup:

- If $d_i > d$, then $w = 2$ and $d' = \hat{d} = d_i + 2$. In the FSAM, the variable's value is not available in the current step, and this thread spends this step suspending. It is reactivated at the end of step d_i and active in step $d_i + 1$ to fetch the new value. Thus, it requires two units of work and is done at step $d_i + 2$.
- Otherwise, if $d_i \leq d$, then $w = 1$ and $d' = \hat{d} = d_i + 1$, and the FSAM requires one step to fetch the value.

In either case, *fin_app* is called on r at step $d_i + 1$, and s has no descendents, so the conclusion holds.

case APP', $e = e_1\ e_2$: The FSAM uses unit work at step d to fork new states s_1 and s_2. By induction on e_1 and e_2, we know that for $i \in \{1, 2\}$, *fin_app* is called on r at step $d_1 - 1$, and on r_2 at step $d_2 - 1$. Also, the total work and maximum depth of these evaluations are w_i and $\hat{d}_i$. In particular, we have $(\ddot{E}'[x \mapsto r_2], e', \kappa, r) \in A_{d'_1}$, and for $i \geq d'_2$, $\sigma_i(\pi_1 r_2) = \ddot{v}_2$

Now by induction on e', *fin_app* is called on r at step $d_3 - 1$, and the total work and maximum depth of this evaluation are w_3 and $\hat{d}_3$. Thus the total work and maximum depth of the entire evaluation are $w = w_1 + w_2 + w_3 + 1$ and $\hat{d} = \max(\hat{d}_1, \hat{d}_2, \hat{d}_3)$, so the conclusion holds.

case APPC', $e = e_1\ e_2$: This case is similar to a combination of the VAR' and APP' cases.

□

Machine Model	$T_{fetchadd}(p)$
Butterfly (rand.)	$O(\log p)$
Hypercube (rand.)	$O(\log p)$
CRCW PRAM (rand.)	$O(\log p/\log\log p)$

Figure 8: Time bounds for implementing fetch-and-add on various machine models with p processors. As these models are all randomized, the bounds hold with high probability.

Corollary 1 *If* $[\,]; 0 \vdash e \overset{\lambda'}{\Longrightarrow} \dot{v}; w, d, \hat{d}$, *then* $e \overset{F}{\Longrightarrow}^{*} \ddot{v}; w, d, \hat{d}$.

Proof: This follows from Theorem 1 by using $n = 0$, $d = 0$, $\kappa = [\,]$, $A = \{\}$, and $\sigma_0(\pi_2 r) = emptyq$. □

Corollary 2 *If* $[\,]; 0 \vdash e \overset{\lambda}{\Longrightarrow} v; w, d, \hat{d}$, *then* $e \overset{F}{\Longrightarrow}^{*} \ddot{v}; w', d', \hat{d}'$ *such that* $2w \geq w'$, $2d \geq d'$, *and* $2\hat{d} \geq \hat{d}'$.

Proof: The PSL and blocking-PSL models differ only in their constant overheads, with the latter having constants at most twice of those in the former. □

4 Implementing the FSAM on Machine Models

We introduced the FSAM only as an intermediary in the mapping of the PSL model onto more realistic machines. So now we complete this mapping and show how to implement the FSAM on such machines, in particular, a butterfly network, hypercube, and PRAM. The implementation closely follows the previous description of the FSAM, but it depends on several underlying tools. Since speculative evaluation centers on the use of queues, the operations on queues are of particular importance.

Our simulation bounds are parameterized by the asymptotic time $T_{fetchadd}(p)$ required to implement a *fetch-and-add* operation [11] (also called a multiprefix [31]) on p processors. In a fetch-and-add operation, each processor has an address and an integer value i. In parallel all processors can atomically fetch the value from the address while incrementing the value by i. This can be implemented in a butterfly or hypercube network by combining requests as they go through the network [31], and on a PRAM by various other techniques [21, 9]. The bounds for $T_{fetchadd}(p)$ for these machine models are given in Figure 8. These bounds assume the butterfly has $p \log_2 p$ switches which can do the combining, and the hypercube can communicate and combine over all wires simultaneously (multiport version). For all these machines if each processor is making m requests, these requests will take a total of $O(m + T_{fetchadd}(p))$ time. The fetch-and-add operation is used in three places in our implementation: memory allocation, the enqueue operation, and allocating tasks to processors.

To handle memory allocation throughout the implementation we assume one global memory space and keep a counter pointed to the next available memory location. Whenever a processor needs memory it can fetch-and-add from this counter using the size of the block it needs. From this, a processor receives the start address of its requested space. In our implementation we assume an unbounded memory and therefore do not consider garbage collection, although there is nothing that precludes its use.

We first discuss how to implement the queues of suspended states. We show that operations on these queues, particularly enqueuing of n elements, requires $O(n)$ amortized work and $O(n/p + T_{fetchadd}(p))$ time. The other main FSAM data structure is an environment. Using balanced binary trees, we can bound the time for each environment access and update by v_e, the logarithm of the number of variables in the program e [2]. Given these, implementing each of the FSAM substeps is straightforward. But we must also show how states map onto machine processors and how these processors are load-balanced.

4.1 Queues

We must support three operations on these queues: creating an empty queue in the application case of $\overset{F}{\Longrightarrow}_1$ and the initialization of the FSAM, enqueuing elements in $\overset{F}{\Longrightarrow}_2$, and dequeuing all elements in $\overset{F}{\Longrightarrow}_3$. A queue q is implemented as a pair of an array q_a and a length q_l, where q_l is the length of the queue, and $q_l \leq |q_a|$. To create a queue we return a new array (of some constant size) and a length 0. To dequeue all elements we just return the queue's array pointer and length—data movement is left to the creation of the new active state array. As we enqueue data, we may need to allocate a new, larger array a, copying the old contents into the new array. However, we must be careful to bound the work spent on such copying.

In an enqueue, many processors each add one element onto one of many queues, where each queue may receive multiple elements. To implement the enqueue each processor fetch-and-adds 1 to the current length q_l of its destination queue, receiving the offset o within the queue for its element. As a side effect q_l is incremented appropriately. Now all processors that receive an offset within the bounds of the destination array ($o \leq |q_a|$), write their element into the position o of the array. Some of the queues, however, might have an overflow ($q_l > |q_a|$). For these queues we will grow their arrays.

To grow the arrays we (1) identify the queues with an overflow and allocate a new array of size $2q_l$ for these queues, (2) copy the contents from the old array, and (3) complete the enqueue operations that were postponed because of overflow. To identify the queues with overflow each processor that has received an offset $o = |q_a| + 1$, remains active and the other processors drop out. These active processors can allocate the space for the new queues by reading q_l, and allocating a space of twice this size from the global pool. The copying is more difficult since the work for copying needs to be balanced across the processors—some queues might be very large, in fact much larger than the current number of states. To properly balance the work for copying the queues we allocate a number of processors proportional to the size of the old queue ($|q_a|$) to each queue. Such allocation can be implemented with the fetch-and-add operation or with segmented operations [3]. Each processor then copies their specified portion of a queue or set of queues into the new arrays. The length q_l of each new queue is set to the number of elements that were copied ($|q_a|$ of the old array). To complete the enqueue operations the processors that received an out of bounds offset previously now enqueue again on the updated queues.

In the enqueue all the operations other than the copying can be implemented in work proportional to the number of enqueue requests. As discussed later, this work is load balanced across the processors. The actual copying can require more work since large queues might need to be copied. However, the cost of these copies can be amortized against the time it took to enqueue into the queues in the first place. In particular for each queue of size q_l, we will have spent at most $2q_l$ aggregated work copying its elements. This is because we are at least doubling its size on each expansion of the array so the cost of a copy is always at most half as much as the cost of the next copy. Since we are accounting for q_l work for the enqueues, the extra work for the copies is amortized against them. The work for copying is properly load balanced, so the running time is also amortized.

4.2 FSAM Implementation and Costs

Using the basic data structures just described, we now simulate the FSAM on our machine models. First we examine the time required for each step of the FSAM, then total this for all steps.

Lemma 1 *Each FSAM step starting with active states A and ending with active states A' can be implemented on a p processor machine within*

$$O(v_e((|A| + |A'|)/p + T_{fetchadd}(p)))$$

amortized time.

Proof: We start with the set of active states in an array A of length a. Each processor is responsible for up to $m = \lceil a/p \rceil$ elements (*i.e.*, processor i is responsible for the elements from ia/p to $(i+1)a/p - 1$). 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 evaluate the states ($\xRightarrow{F}_1$), and synchronize all processors.
2. Suspend all states requesting to do so ($\xRightarrow{F}_2$), and synchronize all processors.
3. Save the result value and reactivate the queued states of all finishing states ($\xRightarrow{F}_3$).
4. Create a new active state array for the next step.

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

Local evaluation of the states requires the time it takes to process m states. The implementation of $\xRightarrow{F}_1$ is straightforward and requires $O(v_e)$ time for environment access and constant time for all other operations. Thus the total time for local evaluation of the states on the machine models of interest is $O(v_e(m + T_{fetchadd}(p)))$, where $T_{fetchadd}(p)$ provides an upper bound on any memory latency or space allocations.

In the second substep, each of the active states may suspend itself. This is accomplished with a single enqueue operation. This requires a constant number of fetch-and-adds over at most a active states. As previously mentioned, such fetch-and-adds pipeline and require a total of $O(m + T_{fetchadd}(p))$ time. As discussed earlier the copying of the arrays for queues that overflow might require more time, but this is amortized against previous steps.

In the third substep, each processor has up to m states to finish. Each processor writes its states' results, then dequeues its states' suspended queues, *i.e.*, returns the queues' pointers and lengths. This requires $O(m + T_{fetchadd}(p))$ time, including memory latency.

The new active state array is the union of the a' states returned by the first and third substeps. We then create the array of size a' in $O((a+a')/p + T_{fetchadd}(p))$ time. This is implemented using a similar processor allocation technique as used to allocate processors for copying arrays in the enqueue operation. □

Theorem 2 *If* $[\,]; 0; e \vdash \dot{v} \xRightarrow{\lambda} w, d, \hat{d};$, *then a p processor machine can calculate $\dot{v}$ from e within*

$$O(v_e(w/p + \hat{d}T_{fetchadd}(p)))$$

time. Analogous results hold for the other models.

Proof: The proof uses Brent's scheduling principle [4]. We assume that step i of the FSAM processes a_i active states. We know from Corollary 2 that $\sum_{i=0}^{i<\hat{d}} a_i = w$. We also know from Lemma 1 that it takes $kv_e(a_i + a_{i+1})/p + T_{fetchadd}(p)$ time for step i, for some constant k. The total time is then

$$\begin{aligned} T &= \textstyle\sum_{i=0}^{i<\hat{d}} kv_e((a_i + a_{i+1})/p + T_{fetchadd}(p)) \\ &= kv_e \textstyle\sum_{i=0}^{i<\hat{d}} (a_i + a_{i+1})/p + T_{fetchadd}(p) \\ &= 2kv_e((\textstyle\sum_{i=0}^{i<\hat{d}} a_i/p) + \hat{d}T_{fetchadd}(p)) \\ &= 2kv_e(w/p + \hat{d}T_{fetchadd}(p)) \end{aligned}$$

□

5 Related Work

Several researchers have used cost-augmented semantics for automatic time analysis or definitional purposes, *e.g.*, [34, 35, 38, 39], the most similar being that of Roe for a lenient language [32, 33]. The following are the primary differences of Roe's model as compared to ours:

- Looking up a variable does not wait for the value as in VAR. He waits for the value only if it is needed, *e.g.*, to apply it.
- Because of the previous difference, two separate depths indicate when the value becomes available (our minimum depth) and provide a "clock" to serialize some computations. We have shown in Section 2.3 how we can serialize with the minimum depth.
- The maximum depth is not tracked.
- Lists are explicitly in the language, rather than encoded. Each cons-cell component is tagged with the depth at which it completes, whereas in our encoding those tags are in the environment of the closure representing the cons-cell. While lists can be encoded, a more pragmatic implementation would need to include them.

Flanagan and Felleisen also gave a cost-augmented operational semantics for a language with futures, defining the total work and the *mandatory* work of a computation [8]. However, none of these related the costs of the modelled language to those in machine models.

Nikhil introduce the P-RISC [24] 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 FSAM have little in common.

6 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 (for the butterfly and hypercube) of optimal in terms of time. To obtain these bounds, we introduce fully parallel operations on queues of suspended threads. An important contribution of the work is the use of a semantic model to define an abstract notion of costs and the techniques used to relate these to the running time, including the introduction of the FSAM.

6.1 Pragmatic Issues

The paper has concentrated on asymptotic behavior at the cost of ignoring "constant" overheads. In particular, while we account for communication costs, we ignore the fact that communication is typically significantly slower than computation. Here we briefly discuss some pragmatic issues about the implementation and how it could be modified to reduce the constants.

Our implementation aggressively creates many threads to maximize parallelism, and it frequently synchronizes all threads to guarantee load-balancing. Since most expressions are relatively simple, and the cost of thread management is high, creating a thread for each subexpression involves too much overhead [30]. Furthermore, the substeps $\stackrel{F}{\Longrightarrow}_i$ embody little computation between each load-balancing. One way to improve this would be to group sets of these substeps between each load-balancing. As long as they were grouped into clusters of constant size, this would not effect asymptotic time bounds, but could greatly reduce load-balancing costs. This is the same basic idea as work examining heuristics for building large sequential blocks of code, *e.g.*, [16, 30].

Another way to reduce communication is to cache in the environment the results of fetching values from other threads. This is simple since the value of a thread never changes once computed.

There are several approaches available to improve the space efficiency of the FSAM. One would be to introduce garbage collection of the various semantics objects. Properly including the time costs of garbage collection would require that we formally model the space taken by a computation and include space bounds. Another is to mutate the state of each thread, rather than creating a series of states over time. As mentioned in section 6.2, a partially speculative variant of the FSAM may be able to schedule states to minimize the number of the active states.

6.2 Partial speculation

Here we discuss two possible relation modifications to the FSAM. First, instead of fully evaluating all threads, any threads irrelevant to computing the final result may be discarded during evaluation [12]. Clearly, this can reduce the asymptotic work and maximum depth of some computations. Second, instead of using all the currently active states on each FSAM step, we could schedule only a subset, leaving the unscheduled ones to be active on the next step. In conjunction with the previous modification, the goal is to minimize the number of irrelevant states that are scheduled while still maintaining sufficient parallelism. In particular, we schedule at most $pT_{fetchadd}(p)$ threads per step. Also, if we can choose the scheduled states appropriately, we might also be able to minimize the maximum number of active states on any step, for space efficiency [1]. But it seems unlikely that call-by-speculation allows an efficient implementation of a depth-first p-traversal of the computation DAG.

Consider a modification of the FSAM model, called the Partially Speculative Abstract Machine (PSAM), which incorporates these changes. Before we can discard irrelevant threads, we must first detect them. For this we assume each thread has a reference count, and that a scheduled state marks its thread irrelevant if it has a zero reference count. It also adds its children to a pool of threads to be marked irrelevant. On each step, we mark some of those threads irrelevant and add their children to the pool. (We mark only as many as the number of scheduled active states on this step in order to maintain our per-step work bound.) We could also detect irrelevant threads statically with strictness analysis. This would detect irrelevant threads less accurately, but obviously earlier. While for some expressions, this may reduce work and maximum depth asymptotically, it is easy to construct expressions where most of each irrelevant thread is evaluated before it is detected to be irrelevant. What we have just described is a form of garbage collection designed specifically not to increase our work bound. Some languages simply incorporate this into their usual garbage collection mechanism [27, 23].

For the purpose of minimizing the work spent evaluating irrelevant threads, scheduling a subset of the active states is only beneficial if we can effectively prioritize states and threads. In order to have the same depth bounds as the FSAM, the PSAM must prioritize threads known to be relevant at the highest level. Note that at most one of those threads is active at a time. One approach is to prioritize the argument of an application lower than its function, since the argument may not be relevant even if the function is. While used in practice [27, 29, 28, 37], it is unclear whether this approach can reduce work asymptotically without increasing depth asymptotically. The problem is updating thread priorities efficiently. During evaluation, as we detect which threads are relevant, we adjust priorities. If changing one thread's priority changes its children's priorities, we must propagate this change to its descendents. On the one hand, we can only propagate some of these priorities within the work and depth bounds of a single PRAM step. On the other, if we do not update priorities fast enough, the priorities used for scheduling may be too inaccurate. Furthermore, if we have an unbounded number of priority levels, we cannot afford the work to sort all the active states' priorities to grab those of highest priority. Although it might be possible to maintain extra order on the tree of threads so that we can efficiently grab those of highest priority.

Acknowledgements

We would like to thank Bob Harper for providing helpful feedback. This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330 and in part by an NSF Young Investigator Award.

References

[1] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[2] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, June 1995.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[4] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[5] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-paralle computer. In David Galernter, Alexander Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, chapter 6, pages 95–113. MIT Press, 1990.

[6] Rohit Chandra, Anoop Gupta, and John L Hennessy. COOL: a language for parallel programming. In David Galernter, Alexander Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, chapter 8, pages 126–148. MIT Press, 1990.

[7] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, June 1993.

[8] Cormac Flanagan and Mattias Felleisen. The semantics of future and its use in program optimization. In *Proceedings 22nd ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[9] Joseph Gil and Yossi Matias. Fast and efficient simulations among CRCW PRAMs. *Journal of Parallel and Distributed Computing*, 23(2):135–148, November 1994.

[10] Ron Goldman and Richard P. Gabriel. Qlisp: Experience and new directions. In *Proceedings ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 111–123, July 1988.

[11] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

[12] Dale H. Grit and Rex L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems*, 3(1):49–59, January 1981.

[13] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[14] Robert H. Halstead, Jr. New ideas in parallel lisp: Language design, implementation, and programming tools. In T. Ito and R. H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, number 441 in Lecture Notes in Computer Science, pages 2–51. Springer-Verlag, June 1989.

[15] Paul Hudak and Steve Anderson. 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, pages 234–256. Springer-Verlag, September 1987.

[16] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *Proceedings ACM Conference on LISP and Functional Programming*, pages 79–90, July 1994.

[17] Takayasu Ito and Manabu Matsui. A parallel lisp language PaiLisp and its kernal specification. In T. Ito and R. H. Halstead, Jr., editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, number 441 in Lecture Notes in Computer Science, pages 58–100. Springer-Verlag, June 1989.

[18] Mike Joy and Tom Axford. Parallel combinator reduction: Some performance bounds. Technical Report RR210, University of Warwick, 1992.

[19] Richard Kennaway. A conflict between call-by-need computation and parallelism (extended abstract). In *Proceedings Conditional Term Rewriting Systems-94*, February 1994.

[20] David A. Kranz, Jr. Robert H. Halstead, and Eric Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[21] Yossi Matias and Uzi Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[22] James S. Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, Massachusetts Institute of Technology, September 1987.

[23] James S. Miller and Barbara S. Epstein. Garbage collection in MultiScheme (preliminary version). In T. Ito and Jr. R. H. Halstead, editors, *Parallel Lisp: Languages and Systems, US/Japan Workshop on Parallel Lisp*, number 441 in Lecture Notes in Computer Science, pages 138–160. Springer-Verlag, June 1989.

[24] Rishiyur S. Nikhil. The parallel programming language Id and its compilation for parallel machines. Technical Report Computation Structures Group Memo 313, Massachusetts Institute of Technology, July 1990.

[25] Rishiyur S. Nikhil. Id version 90.1 reference manual. Technical Report Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1991.

[26] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennar Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH language reference manual, version 1.0—preliminary. Technical Report Computation Structures Group Memo 369, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1995.

[27] Randy B. Osborne. *Speculative Computation in Multilisp*. PhD thesis, Massachusetts Institute of Technology, December 1989.

[28] Andrew S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*. PhD thesis, Department of Computer Science, University of Tasmania, 1991.

[29] Andrew S. Partridge and Anthony H. Dekker. Speculative parallelism in a distributed graph reduction machine. In *Proceedings Hawaii International Conference on System Sciences*, volume 2, pages 771–779, 1989.

[30] Simon L Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[31] Abhiram G. Ranade. *Fluent Parallel Computation*. PhD thesis, Yale University, New Haven, CT, 1989.

[32] Paul Roe. Calculating lenient programs' performance. In Simon L Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Proceedings Functional Programming, Glasgow 1990*, Workshops in computing, pages 227–236. Springer-Verlag, August 1990.

[33] Paul Roe. *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[34] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990.

[35] David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. To appear in Journal of Parallel and Distributed Computing.

[36] Guy Tremblay and G. R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In A. P. Wim Bohm and John T. Feo, editors, *Proceedings High Performance Functional Computing*, pages 119–133, April 1995.

[37] C K Yuen, M D Feng, and J J Yee. Speculative parallelism in BaLinda Lisp. Technical Report TR31/92, Department of Information Systems and Computer Science, National University of Singapore, November 1992.

[38] Wolf Zimmermann. Automatic worst case complexity analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

[39] Wolf Zimmermann. Complexity issues in the design of functional languages with explicit parallelism. In *Proceedings International Conference on Computer Languages*, pages 34–43, 1992.