Space Profiling for Parallel Functional Programs

Daniel Spoonhower Guy E. Blelloch
Robert Harper
Carnegie Mellon University
{spoons,blelloch,rwh}@cs.cmu.edu

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

Abstract

This paper presents a semantic space profiler for parallel functional programs. Building on previous work in sequential profiling, our tools help programmers to relate runtime resource use back to program source code. Unlike many profiling tools, our profiler is based on a cost semantics. This provides a means to reason about performance without requiring a detailed understanding of the compiler or runtime system. It also provides a specification for language implementers. This is critical in that it enables us to separate cleanly the performance of the application from that of the language implementation.

Some aspects of the implementation can have significant effects on performance. Our cost semantics enables programmers to understand the impact of different scheduling policies yet abstracts away from many of the details of their implementations. We show applications where the choice of scheduling policy has asymptotic effects on space use. We explain these use patterns through a demonstration of our tools. We also validate our methodology by observing similar performance in our implementation of a parallel extension of Standard ML.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Performance attributes; D.3.2 [*Language Classifications*]: Parallel languages

General Terms languages, measurement, performance

Keywords profiling, parallelism, cost semantics, Standard ML

1. Introduction

Functional programming languages have been promoted as an attractive means for writing parallel programs. They offer many opportunities for parallel evaluation without requiring programmers to be explicit about concurrency. With a clear semantic definition, programmers can reason about the results of program evaluation independently of any particular implementation or target machine.

In reality, achieving better performance through parallelism can be quite difficult. While the extensional behavior of parallel functional programs does not depend on the language implementation, their performance certainly does. In fact, even sequential implementations of functional languages can have dramatic and unexpected effects on performance. To analyze and improve performance, functional programmers often rely upon profilers to analyze resource use (Appel et al. 1988; Runciman and Wakeling 1993a; Sansom and Peyton Jones 1995; Röjemo and Runciman 1996). With parallel implementations, the need for profilers is magnified by such issues as task granularity, communication, and scheduling policy—all of which can have a significant impact on time and space use.

In this paper, we present a *semantic* space profiler for a call-by-value parallel functional language and relevant to shared memory architectures. Our tools are the first to allow programmers to reason about the space use of parallel functional programs. Our method abstracts away from details of language implementation and yet allows programmers to reason about asymptotic performance. Because it is based on a semantics rather than a particular implementation, our profiling method remains true to the spirit of functional programming: thinking about program behavior does not require a detailed understanding of the compiler or target machine.

Our profiling method must account, at least abstractly, for some parts of the implementation. In this work, we focus on scheduling policy and its effects on application space use. Because the choice of scheduling policy often has dramatic, and even asymptotic, effects on space use (as detailed in this paper), it is critical that a programmer has the flexibility to choose a policy that is best-suited to his or her application. This flexibility must be reflected both in the language implementation and in any profiling tools.

Our profiling tools are based on a *cost semantics* (Sansom and Peyton Jones 1995; Blelloch and Greiner 1996). A cost semantics is a dynamic semantics that, in addition to the ordinary extensional result, yields an abstract measure of cost. In our semantics, this cost is a pair of directed graphs that capture essential dependencies during program execution (Section 3). These graphs are used by our tools to simulate the behavior of different scheduling policies and to make predictions about space use. For example, by generating graphs for a range of inputs, programmers can perform an asymptotic analysis of space use. Our profiling tools also allow programmers to visualize the parallel execution of programs and compare scheduling policies (Section 4).

We emphasize that our method allows users to profile parallel *programs*. This stands in contrast to many existing profilers, which only provide a means of profiling a program *based on a particular implementation*. While this leads to some loss of precision, there is a tradeoff between precision and offering performance results that can be easily related to the program text. Our cost semantics is the fulcrum that allows us to balance this tradeoff.

Our cost semantics also provides a formal specification that forces language implementations to be "safe-for-space" (Shao and Appel 1994). Besides acting as a guide for implementers, it maintains a clean separation between the performance of a program and the performance of the language implementation. This ensures that profiling results are meaningful and that programmers can expect the same asymptotic performance when moving from one compliant implementation to another.

To demonstrate that this specification does not place an onerous burden on implementers, we present an implementation of a parallel extension of Standard ML (Milner et al. 1997) based on our cost semantics (Section 5). Our framework also extends to other parallel extensions of ML (*e.g.,* Fluet et al. (2007)) as well as languages with eager parallelism such as NESL (Blelloch et al. 1994) and Nepal (Chakravarty and Keller 2000). One advantage of our framework is that by factoring out scheduling, we can bring to light performance issues in languages such as NESL that bake in a particular scheduling policy.

Our implementation includes three different scheduling policies. As we also anticipate the need for other policies, we have isolated the core decisions of such policies behind a simple signature.

We implemented several parallel algorithms to validate our work and measured performance using both our tools and by sampling memory use in our implementation. The results show that our cost semantics is able to correctly predict asymptotic trends in memory use (Section 6).

In summary, the contributions of our work include:

- the first space profiling tools for parallel functional programs,
- the first cost semantics supporting an analysis of space use under different scheduling policies,
- an extensible implementation in MLton (Weeks 2006), a high-performance compiler and runtime system.

In the course of our implementation, we also discovered a space leak in one of the optimizations in MLton. As a specification, a cost semantics determines exactly which performance problems must be blamed on the programmer and which can be attributed to the language implementation.

2. Motivating Example

In the next section, we introduce a profiling semantics that assigns a space cost to each program. This cost abstracts away from many details of the compiler, but enables programmers to predict (within a constant factor) the space behavior of different scheduling policies. To motivate this work, we present a small example (matrix multiplication), where the choice of scheduling policy has a dramatic effect on space use. We give a cursory discussion here, and consider this application in further detail in Section 6, along with three other applications (sorting, convex hull, and n-body simulation).

Matrix multiplication offers a tremendous amount of potential parallelism. For inputs of length and width n, each of the n^3 scalar multiplications may be computed in parallel. (Recall that there are n scalar multiplications for each of the n^2 elements in the result.) Figure 1 depicts the code written in our parallel extension of ML. The function `reduce` (not shown) aggregates data in parallel. Note that recursive calls to `loop` (shown as $\{\ldots\}$) in the definition of `tabulate` may also be computed in parallel.

Our framework can be used by programmers to predict the behavior of parallel programs such as matrix multiplication, as summarized below. In general, a programmer would:

```
fun tabulate f n =
  let fun loop i j = (* offset i, length j *)
        if j = 0 then array (0, f 0)
        else if j = 1 then array (1, f i)
        else let val lr = {loop i (j div 2),
                           loop (i + (j div 2)) (j − (j div 2))}
          in
            append (#1 lr, #2 lr) (* line flagged by tool *)
          end
  in
    loop 0 n
  end

fun mmm (m, n) =
  let
    fun vvm (b, a) =
        reduce op+
          (fn i ⇒ sub (b, i) ∗ sub (a, i)) 0.0 (length a)
    fun mvm (n, a) =
        tabulate (fn i ⇒ vvm (sub (n, i), a)) (length n)
  in
    tabulate (fn i ⇒ mvm (n, sub (m, i))) (length m)
  end
```

Figure 1. Matrix Multiplication code

1. Select a program and run it using the profiling interpreter based on our cost semantics.

   The cost semantics yields a pair of directed graphs. As these graphs are too detailed to present in their raw form, our tools summarize these graphs into a more compact form. An example of summarized graphs for matrix multiplication is shown in Figure 2(a). In these graphs, nodes represent sequential computation. In the top graph, edges point downward, and an edge from n_1 to n_2 indicates that n_1 must be executed before n_2. For matrix multiplication, we see the regular structure of its parallelism: work is evenly distributed among parallel branches. In the bottom graph, edges point upward, and an edge from n_2 to n_1 indicates that n_2 depends on the value allocated at n_1.

   At the first stage of analysis, these graphs allow programmers to make qualitative statements about their programs and to classify parallel algorithms visually: algorithms with different parallel structure will give rise to graphs with different shapes. These graphs are also used as input in the following step.

2. Use our simulator to predict the space performance for different scheduling policies and numbers of processors.

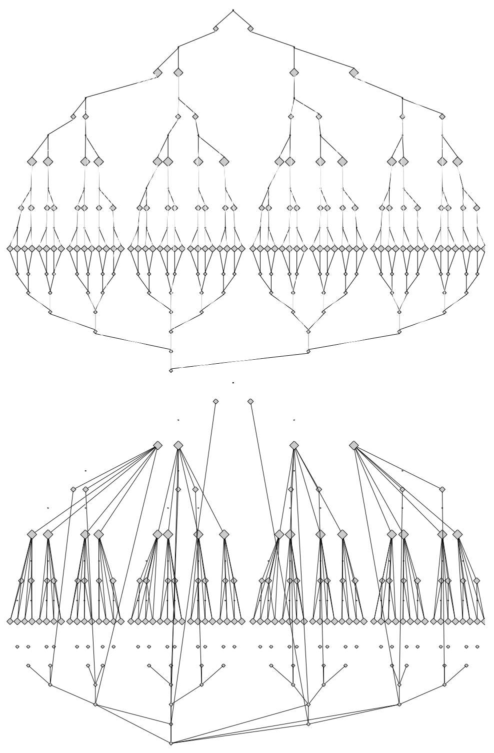
   Each scheduling policy determines a traversal of the cost graphs. By fixing a policy and the number of processors, our simulator uses these graphs to determine the high-water mark for space use. It also determines the point during execution at which this mark is reached, as well as where in the source code this data is allocated and used.

3. Repeat steps 1 and 2 for different inputs. Plot the results to draw conclusions about asymptotic performance.

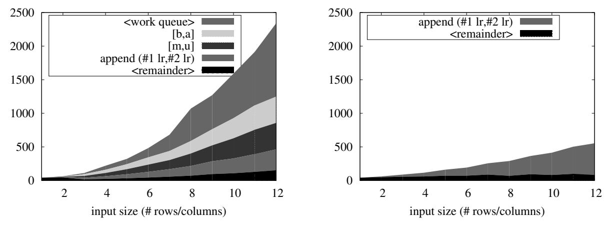
   For each input, programmers generate a new pair of graphs. Our tool can then be used to generate plots such as those shown in Figure 2(b). These plots show trends in space use as a function of input size for different schedulers. In this example, we compare two schedulers each using two processors. The scheduling policy on the left manages parallel tasks using a FIFO queue and implements a breadth-first traversal of the top cost graph. The scheduling policy on the right implements a parallel depth-first traversal (see Section 5.2 for details) of the top cost graph. Our tools also help explain the space use through a breakdown according to particular allocation points (as shown in the figure) or use points. As the figure shows, for both schedulers, a significant part of the space use at the high-water mark can be attributed to the arrays allocated in the implementation of `tabulate` (*i.e.*, `append (...)`, as marked in Figure 1). However, for the breadth-first scheduler (on the left), most of the space is attributed to the work queue and two forms of closure (denoted with "[. . .]" in the key). These two closures appear during the application of `reduce`.

4. Reexamine the cost graphs to isolate space use and elucidate the effects of the scheduling policy.

   While the plots generated in the previous step depict trends in space use, they provide little insight into how the scheduling policy affects these trends. The final step in an analysis often requires programmers to revisit the cost graphs, this time including information about the scheduling policy. In the course of analyzing our example applications, we will show how computing the difference between two executions based on different schedules can explain why one policy yields better performance and



(a) summarized cost graphs for 4×4 matrix multiplication



(b) space high-water mark

Figure 2. Cost Graphs and Simulated Results for Matrix Multiplication. This figure shows (a) summarized cost graphs, (b) space use as a function of input size for two scheduling policies: breadth-first (left) and depth-first (right).

how programs can be modified to improve performance. In our matrix multiplication example (bottom graph of Figure 2(a)), we see that the point where the graph is the widest (*i.e.,* where the most parallelism is available) also marks a shift in how the program uses space. From this point on, most of the data allocated by the program are no longer in use. Our tools can show that the high-water mark for space under a breadth-first policy arises because all these nodes at the widest point are concurrently active. These nodes represent the evaluation of the body of vvm and correspond to the top three entries in Figure 2(b).

We have presented a simple example here but the framework and tools also apply to complex programs with irregular parallel structure.

3. Cost Semantics

The cost semantics (Figure 3) for our language is an evaluation semantics that computes both a result and an abstract cost reflecting *how* that result is obtained. It assigns a single cost to each closed

(expressions)	e	$::=$	$x \mid \mathsf{fun}\ f.x.e \mid e_1\ e_2 \mid \{e_1, e_2\} \mid \#i\ e$
			$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid v$
(values)	v	$::=$	$\langle f.x.e \rangle^{\ell} \mid \langle v_1, v_2 \rangle^{\ell}$
(locations)	ℓ	$\in$	L

Figure 4. Language Syntax. We use a call-by-value functional language with recursive functions, parallel pairs, and booleans. Components of pairs written $\{e_1, e_2\}$ may be computed in parallel. We include a separate class of values with annotations that capture sharing explicitly, but these values do not appear in the surface syntax used by programmers.

program that enables us to construct a model of parallel execution and reason about the behavior of different scheduling policies.

In general, a cost semantics is necessary for any asymptotic analysis of running time or space use. For sequential implementations, there is an obvious cost semantics that nearly all programmers understand implicitly. For languages that fix the order of evaluation, the source code contains all the information necessary to reason about performance.

In this work, we give a cost semantics that serves as a basis for the asymptotic analysis of *parallel* programs, including their use of space. We believe that it is important that this semantics assigns costs to source-level programs. However, since the performance of programs depends on some aspects of the implementation, we must further interpret the results of the cost semantics, as discussed in Sections 3.2 and 3.3 below.

Figure 4 shows the fragment of our language we discuss in this section. In the implementation of our profiler, we extend this fragment with integers, floating-point numbers, lists, trees, and arrays, but none of these extensions prove difficult. We also include two forms of pair construction in our implementation: one that always evaluates components of the pair sequentially and one where components may be evaluated in parallel. Finally, we assume that all values are allocated in the heap for the sake of clarity, but this assumption may also be relaxed.

3.1 Semantics

A cost semantics is a *dynamic* semantics and thus yields results only for closed expressions, *i.e.* for a given program over a particular input. Just as in ordinary performance profiling, we must run a program over a series of inputs before we can generalize its behavior.

Our cost semantics is defined by the following judgment, which is read, *expression e evaluates to value v with computation graph g and heap graph h.*

$$e \Downarrow v; g; h$$

The extensional portions of this judgment are standard in the way that they relate expressions to values. As discussed below, edges in a computation graph represent control dependencies in the execution of a program, while edges in a heap graph represent dependencies on and between heap values.

As in the work of Blelloch and Greiner (1996), computation graphs g are directed, acyclic graphs with exactly one source node (with in-degree 0) and one sink node (with out-degree 0) (*i.e.* directed series-parallel graphs). Nodes are denoted ℓ and n (and variants). Edges in the computation graph point forward in time. An edge from n_1 to n_2 indicates that n_1 must be executed before n_2. Each computation graph consists of a single-node, or of the sequential or parallel composition of smaller graphs. Graphs are written as tuples such as (n_s, n_e, E) where n_s is the source or *start* node, n_e is the sink or *end* node, and E is a set of edges. The remaining nodes of the graph are implicitly defined by the edge set. A graph consisting of a single node n is written $(n, n, \emptyset)$ or simply $[n]$. Graph

$$\boxed{e \Downarrow v; g; h}$$

$$\frac{(\ell \text{ fresh})}{\mathsf{fun}\ f(x) = e \Downarrow \langle f.x.e\rangle^{\ell}; [\ell]; \{(\ell, \ell')\}_{\ell' \in \mathrm{locs}(e)}} \ (\text{E-}\textsc{Fun})$$

$$\frac{e_1 \Downarrow \langle f.x.e_3\rangle^{\ell_1}; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad [\langle f.x.e_3\rangle^{\ell_1}/f][v_2/x]e_3 \Downarrow v_3; g_3; h_3 \quad (n \text{ fresh})}{e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus [n] \oplus g_3; h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \mathrm{loc}(v_2))\}} \ (\text{E-}\textsc{App})$$

$$\frac{e_1 \Downarrow v_1; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad (\ell \text{ fresh})}{\{e_1, e_2\} \Downarrow \langle v_1, v_2\rangle^{\ell}; g_1 \otimes g_2 \oplus [\ell]; h_1 \cup h_2 \cup \{(\ell, \mathrm{loc}(v_1)), (\ell, \mathrm{loc}(v_2))\}} \ (\text{E-}\textsc{Fork})$$

$$\frac{e \Downarrow \langle v_1, v_2\rangle^{\ell}; g; h \quad (n \text{ fresh})}{\#i\ e \Downarrow v_i; g \oplus [n]; h \cup \{(n, \ell)\}} \ (\text{E-}\textsc{Proj}_i) \qquad \frac{(\ell \text{ fresh})}{\mathsf{true} \Downarrow \mathsf{true}^{\ell}; [\ell]; \emptyset} \ (\text{E-}\textsc{True})$$

$$\frac{e_1 \Downarrow \mathsf{true}^{\ell_1}; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad (n \text{ fresh})}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_2; g_1 \oplus [n] \oplus g_2; h_1 \cup h_2 \cup \{(n, \ell_1)\} \cup \{(n, \ell)\}_{\ell \in \mathrm{locs}(e_3)}} \ (\text{E-}\textsc{IfTrue})$$

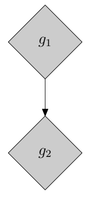
Figure 3. Profiling Cost Semantics. This semantics yields two graphs that can be used to reason about the parallel performance of programs. Computation graphs g record dependencies in time while heap graphs h record dependencies among values. Several omitted rules (*e.g.,* E-IfFalse) follow from their counterparts here. The locations of an expression $\mathrm{locs}(e)$ or a value $\mathrm{loc}(v)$ are the outermost locations of that expression or value. These auxiliary functions are defined in Appendix A.

operations are defined below. Operands are shown first while the results are shown both graphically and symbolically below. (Here, a gray diamond stands for an arbitrary subgraph.)

Serial Composition

$g_1 \oplus g_2$

$(n_s, n_e, E) \oplus (n'_s, n'_e, E')$

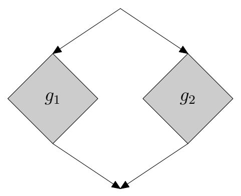


$(n_s, n'_e, E \cup E' \cup \{(n_e, n'_s)\})$

Parallel Composition

$g_1 \otimes g_2$

$(n_s, n_e, E) \otimes (n'_s, n'_e, E')$



$(n, n', E \cup E' \cup$
$\{(n, n_s), (n, n'_s),$
$(n_e, n'), (n'_e, n')\})$
with n, n' fresh

In the parallel composition on the right, the node n is called the *fork* point and the node n' is called the *join* point.

We extend the work of Blelloch and Greiner (1996) with heap graphs. Heap graphs are also directed, acyclic graphs but do not have distinguished start or end nodes. Each heap graph shares nodes with the computation graph arising from the same execution. In a sense, computation and heap graphs may be considered as two sets of edges on a shared set of nodes. As above, the nodes of heap graphs are left implicit.

While edges in the computation graph point forward in time, edges in the heap graph point backward in time. If there is an edge from n to ℓ then n depends on the value at location ℓ. It follows that any space associated with ℓ cannot be reclaimed until after n has executed and any space associated with n has also been reclaimed.

Edges in the heap graph record both the dependencies among heap values and dependencies on heap values by other parts of the program state. As an example of the first case, in the evaluation rule for parallel pairs (E-Fork), two edges are added to the heap graph to represent the dependencies of the pair on each of its components. Thus, if the pair is reachable, so is each component. In the evaluation of a function application (E-App), however, two edges are added to express the *use* of heap values. The first such edge marks a use of the function. The second edge is more subtle and denotes a *possible* last use of the argument. For strict functions, this second edge is redundant: there will be another edge leading to the argument when it is used. However, for non-strict functions, this is the first point at which the garbage collector might reclaim the space associated with the argument. A similar issue is raised in the rules for conditionals. In E-IfTrue, the semantics must record the locations that appear in the branch that is *not* taken. (In this case, these are the locations of e_3.) Again, the intuition is that this is the first point at which the storage corresponding to these locations might be reclaimed. In a sense, these edges represent our imperfect knowledge of program behavior at runtime: even though e_3 will never be executed, that fact is not known until the conditional is executed.

While there is some flexibility in designing these rules in our cost semantics, we choose the versions presented here because they can be implemented in a reasonable way and yet seem to constrain the implementation sufficiently. Care must be taken, however, because the implications of rules are sometimes subtle–see the example in Section 7.1.

3.2 Schedules

Together, the computation and heap graphs enable a programmer to analyze the behavior of her program under a variety of scheduling policies and numbers of processors. For a given program, each policy and processor count will give rise to a particular parallel execution that is determined by the constraints described by the computation graph g.

Definition (Schedule). *A* schedule *of a graph $g = (n_s, n_e, E)$ is a sequence of sets of nodes $N_0, \ldots, N_k$ such that $N_0 = \emptyset$, $n_e \in N_k$, and for all $i \in [0, k)$,*

- *$N_i \subseteq N_{i+1}$, and*
- *for all $n \in N_{i+1}$, $\mathrm{pred}_g(n) \subseteq N_i$.*

Here, $\mathrm{pred}_g(n)$ is the set of nodes n' such that there is an edge (n', n) in g. To restate, a schedule is a traversal of the computation graph where each node may only be visited after all of its parents and where several nodes may be visited in the same step.

3.3 Roots

To understand the use of space, the programmer must also account for the structure of the heap graph h. Given a schedule $N_0, \ldots, N_k$ for a graph g, consider the moment of time represented by some N_i. Because N_i contains all previously executed nodes and because edges in h point backward in time, each edge (n, ℓ) in h will fall into one of the following three categories.

- Both $n, \ell \notin N_i$. As the value associated with ℓ has not yet been allocated, the edge (n, ℓ) does not contribute to the use of space at time i.
- Both $n, \ell \in N_i$. While the value associated with ℓ has been allocated, the use of this value represented by this edge is also in the past. Again, the edge (n, ℓ) does not contribute to the use of space at time i.
- $\ell \in N_i$, but $n \notin N_i$. The value associated with ℓ has already been allocated, and n represents a possible use in the future. Here, the edge (n, ℓ) *does* contribute to the use of space at time i.

In the definition below, we must also explicitly account for the location of the final value resulting from evaluation. Though this value may never be used in the program itself, we must include it when computing space use.

Definition (Roots). *Given $e \Downarrow v; g; h$ and a schedule $N_i, i = 0, \ldots, k$ of g, the* roots *after the evaluation of the nodes in N_i, written $\mathrm{roots}_{v,h}(N_i)$, is the set of nodes ℓ in N_i where $\ell = \mathrm{loc}(v)$ or h contains an edge leading from outside N_i to ℓ. Symbolically,*

$$\mathrm{roots}_{\ell,h}(N_i) = \{\ell \in N_i \mid \ell = \mathrm{loc}(v) \vee (\exists n.(n, \ell) \in h \wedge n \notin N_i)\}$$

The location of a value $\mathrm{loc}(v)$ is simply the outermost location of a value as defined in Appendix A.

We use the term *roots* to evoke a related concept from work in garbage collection. For the reader who is most comfortable thinking in terms of an implementation, the roots might correspond to those memory locations that are reachable directly from the processor registers or the call stack. In the case of a parallel implementation, it also includes those locations that are reachable directly from the scheduler queue.

3.4 Provable Implementations

In the following section, we will use cost graphs, schedules, and roots as a basis for our profiling tools. However, these concepts can also serve as part of a formal specification and verification of an implementation. For example, one can present an implementation of our language using a small-step parallel semantics. Such a semantics is known as a *provable implementation* (Blelloch and Greiner 1996) because each transition can be related to a step in a schedule of the corresponding cost graphs.

The definition of schedules given above includes any execution that respects the dependencies present in the original program. This includes schedules that use an unbounded amount of parallelism in a single step as well as those that stall execution for an unbounded length of time. We can refine this definition to limit ourselves to more realistic classes of schedulers and even to particular policies. One advantage of using cost graphs is that such refinements can be stated in a simple and clear manner. For example, a *p-bounded* scheduler uses at most p processors at each step. In terms of the definition above,

$$\forall i \text{ such that } 0 < i \leq k, |N_i \backslash N_{i-1}| \leq p$$

Similarly, the behavior of a depth-first scheduler should correspond to a depth-first traversal of the computation graph.

In our accompanying technical report (Spoonhower et al. 2008), we give several parallel implementations, ranging from a nondeterministic semantics that describes every possible parallel evaluation to implementations of specific scheduling policies. For each such implementation, we prove that the cost of evaluation (in terms of space use) is reflected in the corresponding schedule of the cost graphs.

4. Profiling

We have implemented this cost semantics as an interpreter, and the resulting cost graphs are used by the two tools described below.

4.1 Parallel Simulation

Our simulator can be instantiated to measure many different performance characteristics, but each instance can be broken down into three parts: a generic component that maintains information about the graph traversal, an implementation of a scheduling policy, and a function that measures some aspect of program performance.

The simulator is a sequential program so implementing scheduling policies in the simulator requires no synchronization or other concurrency control. The breadth- and depth-first scheduling policies mentioned above can each be implemented with a single line of SML using standard list primitives.

We focus on the high-water mark of memory use as the primary measure of performance. As described above, the *roots* of the heap graph at an intermediate point during execution represent those values that are immediately reachable by the program itself. The total space in use at one point during execution is determined by the set of all nodes in the heap graph reachable from these roots. We use the size of the set of reachable nodes in the heap graph as the measure of space use.

By iterating over different inputs, we can compute the high-water mark as a function of the input size. This allows us to plot results such as those shown in Figure 2(b).

4.2 Visualization

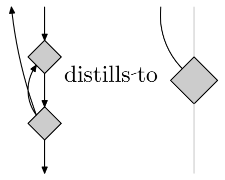
The cost graphs given by our semantics are often quite large and difficult to process visually. We have implemented a method that distills these graphs and yields their essential components. As the goal is to produce a meaningful visual output, we associate a size and color with each node and edge. (In this write-up, we restrict ourselves to black and shades of gray.) As the computation and heap graphs share nodes, we show one superimposed over the other. The resulting graph can then be rendered into various image formats by a freely available graph layout package.[1] All of the graphs shown in this paper are mechanically generated from our cost semantics and an implementation of the following numbered rules. We determined these rules in part through experimentation, but in a large part upon the principles discussed below.

We are most interested in the parallel structure of program execution. Thus a series of nodes that describes sequential computation can be rendered as a single node. We use node size to indicate the amount of sequential computation and node position to indicate computational dependencies.

1. For each node n in the computation graph if 1) n has out-degree one, 2) n has in-degree one, and 3) the (sole) predecessor of n also has out-degree one, then coalesce n with its predecessor. The area of the resulting node is the sum of the area of the two coalesced nodes. Nodes in the original graph are assigned unit area.
2. Remove edges between coalesced nodes in both the computation and heap graphs. (There are no self-edges in the result.)
3. Constrain the layout so that the vertical position of nodes respects the partial order determined by computation edges.

[1] Graphviz - Graph Visualization Software, `http://www.graphviz.org/`

In the output graph, we draw computation edges in a light gray as the structure of the computation graph can be roughly derived from the layout of the nodes: among vertically aligned nodes, those closer to the top of the graph must be executed first. We also omit arrowheads on edges as they add to visual clutter. An example of node coalescing is shown here.



Due to the structure of the cost graphs, coalescing will never create non-unique edges in the computation graph (*i.e.,* more than one edge between the same pair of nodes). On the other hand, it will often be the case that there are several *heap* edges between the same pair of nodes. We considered trying to represent these duplicate heap edges, for example, by weighting heap edges in the output according to number of duplicates. This, however, ignores any sharing among these heap edges and may lead to confusing visual results (*e.g.,* a group of heavy edges that represents a relatively small amount of heap space). For graphs distilled independently of a particular point in time, duplicate heap edges are removed, and all heap edges are given the same weight.

If we restrict ourselves to a single moment in time, we can overcome the difficulties with representing sharing and also focus on the behavior of a specific scheduling policy and its effect on memory use. We use the color of both nodes and heap edges to highlight the behavior of the scheduling policy at time i, the moment when memory use reaches it high-water mark.

4. Fill nodes with black if they are executed at or before time i, and gray otherwise.
5. Draw heap edges that determine roots at time i in black and other heap edges in gray.

When coloring nodes and edges according to these rules, we must be careful about which nodes we coalesce, as we expect those heap edges that determine roots to connect only executed and unexecuted nodes.

6. Avoid coalescing two nodes if one has been executed at or before time i and the other has not.

Finally, now that we have restricted ourselves to a single moment in time, we can properly account for sharing in the heap graph.

7. Weight each heap edge according to its share of the total amount of heap space reachable from that edge at time i.

Thus the total space use at the high-water mark may be determined from the total weight of the black heap edges (*i.e.,* those representing the roots). Generally, the above rules mean that the visual properties of distilled graphs can be interpreted according to the following table.

The greater the ...	then the more...
graph height	sequential dependencies
graph width	possible parallelism
node size	computation
edge thickness	space use

5. Implementation

In this section we describe an implementation of a parallel functional language based on our semantics. This serves to validate our profiling results and demonstrate that implementations of our specification can achieve good parallel speed-ups.

Our implementation is an extension of MLton (Weeks 2006), a whole-program, optimizing compiler for Standard ML (Milner et al. 1997). This is the first parallel implementation of MLton. In keeping with the philosophy that performance-critical code can be written in a high-level language, we implemented as much as the runtime support for parallelism in SML as we could. That said, we were also required to make some changes and additions to the existing runtime system, which is written in C.

5.1 Runtime System

MLton is comprised of a compiler, a set of libraries, and a uniprocessor runtime system. Our first task was to make modifications to the runtime to ensure that shared resources would be safely accessed by concurrently executing processors. In our initial revision, we added a global mutex around all accesses to these shared resources. We then found the hottest code paths and replaced this mutex with lighter-weight mechanisms or restructured the code to avoid synchronization altogether. In some cases, this required adding per-processor state. For example, each processor maintains a local allocation pool that it may use to satisfy allocation requests without synchronization. When the local pool is exhausted, the runtime uses an atomic compare-and-swap operation to claim a portion of memory from the global pool. We were also required to make some minor changes to the compiler and standard basis library to ensure thread safety.

We have not yet addressed the issue of parallel garbage collection in our implementation. However, we believe that previous work in parallel collection for SML (Cheng and Blelloch 2001) could be carried over in a straightforward manner.

Our extended runtime supports an additional runtime parameter that indicates how many processors to use.[2] For each processor, the runtime sets up the local processor state and invokes the main scheduling loop. The remaining parallel functionality, including the scheduling loop, is handled by a set of SML modules, described below.

5.2 Scheduling Policies

At the core of our parallel library is the scheduler loop. The loop is run in parallel by each processor and repeatedly executes the highest priority task that is ready to be run. It is the role of the scheduling policy to determine the highest priority task. In order to plug-and-play with different schedulers, we developed a simple signature that any scheduling policy must implement (Figure 5).

Given the purpose of scheduling policies, the functions add and get should be self-explanatory. The finish function is called once for each task removed from the queue. For many scheduling policies, finish does nothing. The final function shouldYield is used to avoid some non-trivial thread operations in cases where they are unnecessary. This operation is discussed in more detail in the description of the work-stealing scheduler below. Though we present this interface as an SML signature, we believe that this abstraction would be useful for other data parallel implementations.

We include three scheduling policies in our analysis and implementation: depth-first, breadth-first, and work-stealing. (Each is between 50 and 125 lines of SML in our implementation.) Each of these policies is *greedy*, in that processors will not be kept idle if there are available tasks. Moreover, each permits rescheduling only at fork points and join points. These are features of the particular schedulers we study and not limitations of our framework.

[2] We envision a version that allows users to dynamically add and remove processors from the active set, but in the current implementation, this set remained fixed.

```
signature SCHEDULING_POLICY =
sig
  (* processor identifier *)
  type proc = int
  (* abstract type of work, defined elsewhere as unit → unit *)
  type work

  (* these take the identifier of the current processor as their
   first argument *)
  (* add new work to the queue; highest priority appears first *)
  val add : proc → work list → unit
  (* remove the next, highest priority work *)
  val get : proc → work option
  (* mark the most recent unit of work as done *)
  val finish : proc → unit
  (* is there higher priority work for the given processor? *)
  val shouldYield : proc → bool
end
```

Figure 5. Signature for Scheduling Policies. Scheduling policies are defined by implementing this signature.

Breadth-First Scheduling The breadth-first policy is the simplest policy in our implementation. It maintains a single FIFO queue and uses a global lock to serialize concurrent access to the queue. This scheduling policy is equivalent to a left-to-right, breadth-first traversal of the computation graph. It is the "fairest" of the three schedulers we implemented.

Depth-First Scheduling The parallel depth-first policy (Blelloch et al. 1999) prioritizes tasks according to a left-to-right depth-first traversal of the computation graph. Our implementation uses a single global queue and runs tasks from the front of the queue. This it not strictly a LIFO queue, however. To ensure that our implementation obeys the global left-to-right depth-first priority, the children of the leftmost task must be prioritized more highly than the children of nodes further to the right. (In a sense, priority is inherited.) To assign proper priorities, our implementation also maintains one "finger" for each processor that indicates where new tasks should be inserted into the queue (Blelloch et al. 1999). The finish function is used to clean up any state associated with this finger.

Work-Stealing Scheduling The work-stealing policy (Blumofe and Leiserson 1999) maintains a separate queue for each processor. Locally, each queue is maintained using a LIFO discipline. However, if one of the processors should exhaust its own queue, it randomly selects another processor to "steal" from and then removes the *oldest* task from that queue. In the common case, each processor only accesses its own queue, so we can use a more finely-grained synchronization mechanism than in the other two scheduling policies to serialize concurrent access.

Because a work-stealing policy favors local work, a dequeue that immediately follows an enqueue will always return the task that was enqueued. Our implementation avoids these two operations (and also avoids suspending the current thread) by always returning false as the result of shouldYield. The remainder of the parallel library checks the result of this function in cases where a dequeue will follow an enqueue.

5.3 Parallel Library

The lowest-level parallel interface in our library provides methods for suspending and resuming computation along with adding new tasks to the work queue. It is built as a thin wrapper around MLton's user-level thread library.[3] This wrapper adds the proper calls to the scheduling policy to ensure that tasks are initiated in the proper order and finished correctly. This interface, however, is not intended for programmers. Instead, we also provide routines for parallel pairs, futures, and array manipulation based on these primitives. For example, the parallel pair construct used in our cost semantics is implemented by the following function.

```
(* run two functions, possibly in parallel, and return their results
   as a pair *)
val fork : (unit → α) * (unit → β) → α * β
```

This function is implemented by (possibly) suspending the current computation and adding two new parallel tasks, one for each branch of the fork. Through the use of shared state and an atomic compare-and-swap operation, these tasks agree which of the two finished second. This task is responsible for adding a third task that will resume the suspended computation with the new pair. The other routines in our library are implemented in a similar manner, or by building upon functions such as fork.

5.4 Space Profiling in MLton

In the course of gathering our empirical results, we needed to measure space use for applications compiled with MLton. Like many garbage collectors, the MLton implementation can easily compute and report the live data after each collection and thus an approximation of the high-water mark. However, given the default behavior of the collector, there is no way to understand the accuracy of that approximation. In fact, using the collector to determine the high-water mark of space use with perfect accuracy would require a collection after every allocation or pointer update. This would be prohibitively expensive.

However, if we accept *bounded* errors in our measurement, we use the collector to measure this quantity with relatively little effect on performance. To measure the high-water mark of space use within a fraction R of the true value, we restrict the amount of space available for new allocations as follows. At the end of each collection, given that the current high-water mark is M bytes and there are currently L bytes of live data in the heap, we restrict allocation so that so more than $M * (1 + R) - L$ bytes will be allocated before the next collection. In the interim between collections (*i.e.*, between measurements) the high-water mark will be no more than $M * (1 + R)$ bytes. Since the collector will report at least M bytes, it will achieve the desired level of accuracy.

As we report in the section on empirical results below, this technique has enabled us to measure space use with low overhead and predicable results and without any additional effort by the programmer.

6. Empirical Results

We performed our experiments on a four-way dual-core x86-64 machine with 32 GBs of physical RAM running version 2.6.21 of the GNU/Linux kernel. Each of the four processor chips is a 3.4 GHz Intel Xeon, and together they provide eight independent execution units. In the remainder of this section, we will refer to each execution unit as a processor. We focus on measurements of space use, but also report on scalability. Each application is described in more detail below.

6.1 Space Use

For each application, we report the effect of scheduling policy and number of processors on the amount of memory required by

[3] MLton's thread library implements one-time continuations with functionality similar callcc and throw, except that the argument to "callcc" must return another thread to switch to, and a "continuation" may only be thrown to once.

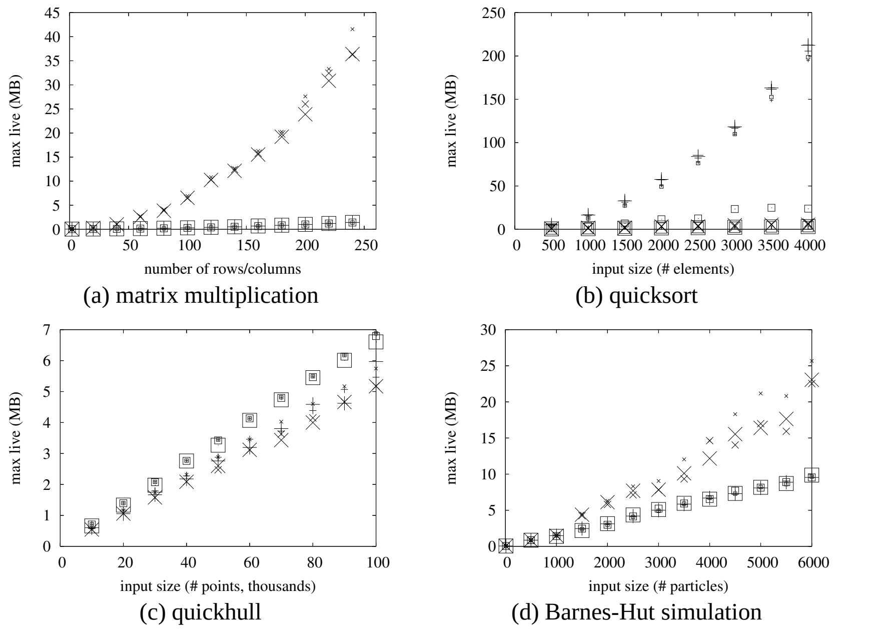


Figure 6. Space Use vs. Input Size. Each plot shows the high-water mark of space use for one of four applications. We tested three scheduling policies, depth-first ($+$), breadth-first ($\times$), and work-stealing ($\boxdot$), with up to four processors. (Larger symbols indicate that more processors were used.) Different scheduling policies yield dramatically different performance, as discussed in the text.

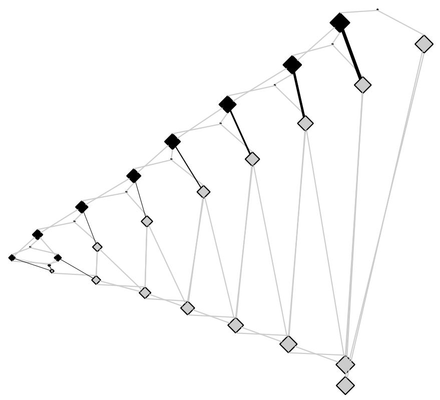
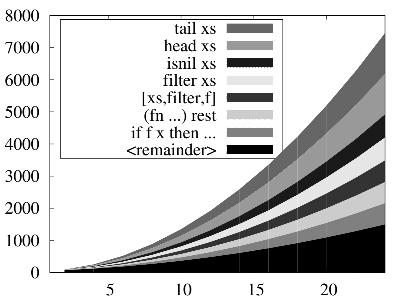


Figure 7. Cost Graphs for Quicksort. These summarized graphs show the point at which the program reaches the high-water mark of space use under a depth-first scheduling policy.

the application. We measured the high-water mark of space use including both stacks and reachable objects in the heap. Measuring this quantity with complete accuracy would require traversing all reachable objects in the heap after every allocation and pointer mutation. Instead, we use the technique described in Section 5.4 to measure this quantity within a tunable bound. Figure 6 shows the high-water mark of space use for four of the applications in our study. Smaller values indicate better performance. We use different shapes to represent different policies: $\times$ for breadth first, $+$ for depth-first, and $\boxdot$ for work-stealing. Larger symbols indicate more processors were made available.

Matrix Multiplication The analysis in Section 2 (recall Figure 2(b)) predicts that the breadth-first scheduling policy uses asymptotically more space than the depth-first policy. A similar analysis predicts that breadth-first is far worse than work-stealing. Both these predictions are confirmed by the space use in our implementation, as plotted in Figure 6(a).

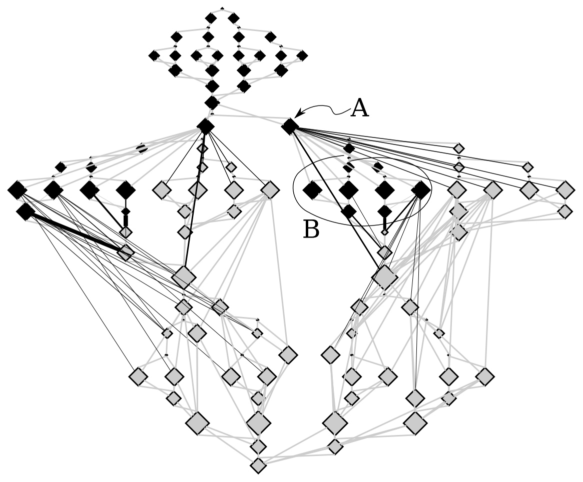
Sorting We implemented several sorting algorithms including quicksort, mergesort, insertion sort, and selection sort. Figure 6(b) shows the space use of a functional implementation of quicksort where data are represented as binary trees with lists of elements at the leaves. This plot shows the behavior for the worst-case input: the input elements are given in reverse order. While we would expect quicksort to take time quadratic in the size of the input in this case, it is perhaps surprising that it also requires quadratic space. This behavior is also predicted by the cost semantics. The plot below shows the consumers of space for the depth-first policy at the high-water mark.



The plot shows that this space is referenced by various parts of the filter function. The graph in Figure 7 shows that the high-water mark for this policy occurs after the left branch of each parallel fork has executed. As expected, there are few opportunities for parallel execution with this input because at each level of the recursion, quicksort splits its input into a single element on the right and all the remaining elements on the left. However, until the partition is complete each branch on the right-hand side is still holding on to the recursive input. This analysis suggests an alternative implementation. If we introduce a join point between partitioning the elements and recursively sorting them, we can avoid the asymptotic increase in space use.

Convex Hull This application computes the convex hull in two dimensions using the quickhull algorithm. We again show results for the worst-case input: the input points are arranged in a circle and so every point in the input is also in the hull. Figure 6(c) shows the high-water mark of space use, which again matches our cost semantics-based predictions (not shown).

Unlike in the quicksort case, there is still significant available parallelism in this example. However, it is more constrained than the parallelism available in matrix multiplication. The algorithm proceeds by partitioning the point set by dividing the plane in half (in parallel) and then recursively processing each half (in parallel). Between these two phases there is a synchronization. This is shown through the widening and narrowing of the graph below.



Nodes are colored to illustrate one point in the execution of the work-stealing scheduling policy. In this case, the work-stealing policy performs more poorly than either of the other two policies because it starts, but does not finish, computing these partitions. The point labeled “A” represents the code that allocates one set of points. The black lines extending to the right of this point indicate part of the program that will compute one half of a partition of these nodes. The circled nodes labeled “B” also compute half a partition, but have already completed their work and have allocated the result. At this point in time, the program is holding onto the entire original set plus half of the resulting partition. The same pattern appears for each processor. Neither of the other two scheduling policies exhibit this behavior.

n-Body Simulation Figure 6(d) shows space use for our implementation of the Barnes-Hut simulation. This algorithm approximates the gravitational force among particles in 3-space. The force on each particle is calculated either by computing the pairwise force between two particles or by approximating the effect of a distant set of particles as a single, more massive particle. Particles are organized using an octree. This algorithm and representation are easily adapted to a parallel setting: not only can the forces on each particle be computed in parallel, but the individual components of this force can also be computed in parallel.

Like the matrix multiplication example, the breadth-first scheduling policies performs poorly due to the large amount of available parallelism and the size of the intermediate results. Though its performance is not as bad as in the multiplication example, it is still significantly worse than the other two policies.

6.2 Overhead of Space Measurements

Measuring space use precisely can be expensive. Using the technique described above, however, we can measure the high-water mark of space use within a fixed bound. Here, we compare the quality and cost of these measurements with those derived from a hand instrumented version. In Figure 8(a), we show space use measurements for the matrix multiplication example, but measured four different ways. All these data series use the breadth-first scheduler and one processor. The first series ($\odot$) shows the measurements obtained when additional garbage collections are explicitly added by the programmer. The other three series show the results using the restricted allocation technique with a bound of 10%, 20%, and 50%, respectively. These bounds are shown with error bars, but only positive errors are shown (as the true high-water mark cannot be smaller than the reported value). The reported values appear at the bottom of the indicated ranges.

We take the measurements derived from the explicit garbage collection to be the most accurate measurement of memory use. (In each case, this technique reported the greatest values.) The figure shows that the restricted allocation measurements are much more accurate than we might expect. For example, the 50% bound seems to be overly conservative: the actual measurements are within 15% of the measurement derived using explicit garbage collections.

In addition to requiring less knowledge on the part of the programmer and yielding measurements with a bounded error, this technique requires less time to perform the same measurements. Figure 8(b) shows the execution time of these four instrumented versions. Values are normalized to the execution time of an uninstrumented version and shown on a logarithmic scale.

6.3 Scalability

As the purpose of parallelism is to improve performance, we also report on scalability. While we are still working to improve the performance of our implementation, these data should be sufficient to convince the reader that we have not “cooked” our implementation simply to match the space use predictions of our semantic profiler.

Figure 9 shows normalized parallel overhead for one to eight processors for several different applications. Parallel overhead is defined as (execution time $\times$ number of processors). We normalize this value to the execution time of the sequential version. Smaller values are better. A value of 1.0 is an ideal value indicating perfect speed-up. A value of 2.0 indicates, for example, that a four processor execution would take 50% of the time of the sequential version or that an eight processor execution would take 25% of the time of the sequential version.

In these plots, we do not include the cost of garbage collection. As we argued above, previous work has shown that garbage collection can be performed efficiently in parallel. We do, however, include overhead due to synchronization and contention for other shared resources.

Though the ideal value is 1.0, we generally expect some overhead from parallel execution. Practically speaking, we are looking for applications (and scheduling policies) where points fall on a line with a slope close to zero. This indicates that adding additional processors will continue to improve performance at roughly the same rate.

We chose benchmarks for this set of measurements not based on interesting space use patterns, but instead by looking for applications with a reasonable potential for parallel execution. Part (a) shows overhead for a blocked version of matrix multiplication. Part (b) shows overhead for parallel mergesort on uniformly randomly distributed input. Part (c) shows the overhead of quickhull for points distributed uniformly a circle. Part (d) shows the overhead for the Barnes-Hut simulation with points distributed uniformly randomly.

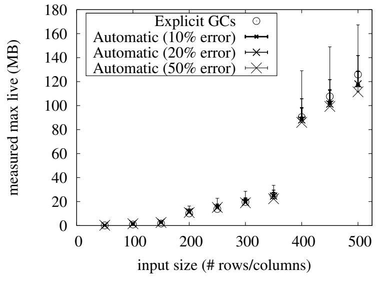
7. Discussion

7.1 Alternative Rules

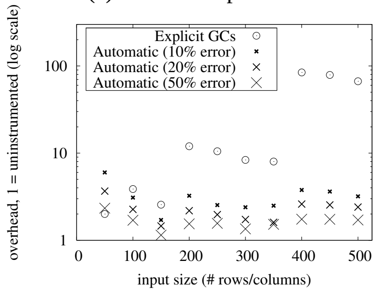
There are a number of design choices latent in the rules given in Figure 3. Different rules would have led to constraints that were either too strict (and unimplementable) or too lax.

Consider as an example the following alternative rule for the evaluation of function application. The premises remain the same, and the only difference from the conclusion in Figure 3 is highlighted with a rectangle.

$$\frac{e_1 \Downarrow \langle f.x.e_3\rangle^{\ell_1}; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad \cdots}{\begin{array}{l} e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus \boxed{g_3 \oplus [n]}; \\ \qquad h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \mathrm{loc}(v_2))\} \end{array}}$$

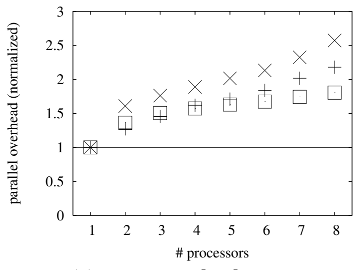


(a) measured space use

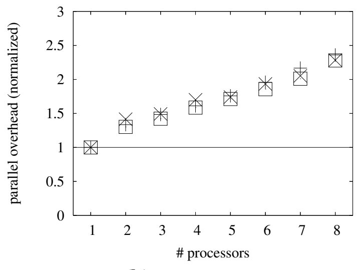


(b) instrumentation overhead

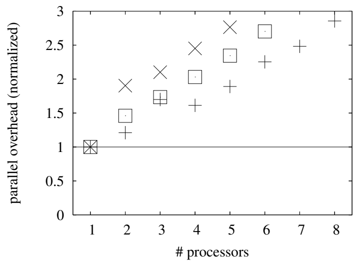
Figure 8. Profiling Space Use. Four different measurements are shown on the left (a). On the right (b), we show the cost of obtaining these measurements. Execution times are normalized to an uninstrumented version and shown on a logarithmic scale.



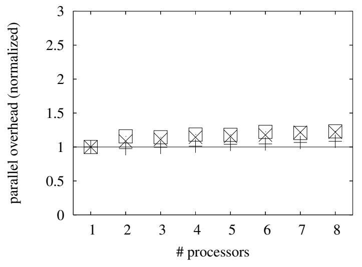
(a) matrix multiplication



(b) mergesort



(c) quickhull



(d) Barnes-Hut simulation

Figure 9. Parallel Scalability. These plots show parallel overhead, defined as (execution time $\times$ number of processors), normalized to the sequential execution time. Smaller values are better with the ideal being 1.0. As an example, a value of 2.0 indicates that an 8 processor execution achieves a 4X speed-up over a sequential execution, a 4 processor execution achieves a 2X speed-up, etc. Overhead does not include garbage collection.

This rule yields the same extensional result as the version given in Figure 3, but it admits more implementations. Recall that the heap edges (n, ℓ_1) and $(n, \mathrm{loc}(v_2))$ represent possible last uses of the function and its argument, respectively. This variation of the rule moves those dependencies until *after* the evaluation of the function body.

This rule allows implementations to preserve these values, even in the case where they are not used during the function application or in subsequent evaluation. This includes the case where these values are bound to variables that do not appear in the program text of the function body or after the application. This is precisely the constraint described by Shao and Appel (1994) as "safe-for-space."

In contrast, the original version of this rule requires that these values be considered for reclamation by a garbage collector as soon as the function is applied. Note, however, that the semantics does not specify *how* the implementation makes these values available for reclamation: it is left to the implementer to determine whether this is achieved through closure conversion, by clearing slots in the stack frame, or by some other means.

As this example suggests, there is some leeway in the choice of the semantics itself. Our goal was to find a semantics that describes a set of common implementation techniques. When they fail to align, our experience suggests that either the semantics or the implementation can be adjusted to allow them to fit together.

7.2 Program Optimizations

In the course of validating our work, we discovered one example where we were required to change the implementation. We came across an example where our implementation failed to distinguish two scheduling policies as our profiler had predicted. Specifically, instead of one policy leading to linear use of space and the other to quadratic use of space, the program required quadratic space under *both* policies.

Some investigation revealed that the problem was not in our semantics or our parallel implementation, but in an existing optimization in MLton: reference flattening. Analogously to tuple flattening, references that appear in a heap-allocated data structure (e.g. a record) may be *flattened* or treated as a mutable field within that structure. At run time, such a reference is represented as a pointer to the containing structure. Accesses to the reference must compute an offset from that pointer.

This optimization can save some space by eliminating the memory cell associated with the reference. However, it can also increase the use of space. As the reference is represented as a pointer to the entire structure, all of the elements of the structure will be reachable anywhere the reference is reachable. If the reference would have outlived its enclosing structure then flattening will extend the lifetime of the other components.

To avoid asymptotically increasing the use of space, MLton uses the types of the other components of the structure to conservatively estimate the size of each component. If any component could be of unbounded size then the reference is not flattened. The problem was that this analysis was not sufficiently conservative in its treatment of recursive types. Though a single value of a recursive type will only require bounded space, an unbounded number of these values may be reachable through a single pointer. Based on our reporting, this problem has been corrected in a recent version in the MLton source repository.

7.3 Nested Parallelism

Another form of “flattening” appears in implementations of nested parallelism such as NESL (Blelloch et al. 1994) or Nepal (Chakravarty and Keller 2000). Here, flattening refers to a compilation technique where by nested parallelism (where parallel tasks may spawn new parallel tasks) is transformed into flat parallelism (where parallel tasks are spawned only at the top level). This automatically balances work by exposing all available parallelism at the same time. Though this technique is largely implemented by the compiler and less so by the language runtime, we can still reason about it using cost graphs and our scheduling formalism. This is discussed in more detail in the accompanying technical report (Spoonhower et al. 2008).

This characterization of the form of breadth-first scheduling agrees with our experience in NESL: examples such as matrix multiplication required so much space that they were unrunnable for even moderately sized inputs, requiring the user to scale back parallelism explicitly. These space use problems in NESL are an important part of the motivation for the current work.

8. Related Work

Parallelism and Concurrency Many researchers have studied how to improve performance by exploiting the parallelism implicit in side-effect free languages. This includes work on data-flow languages (*e.g.,* Arvind et al. (1989)), lazy parallel functional languages (*e.g.,* Aditya et al. (1995)), and nested data parallel languages (*e.g.,* Blelloch et al. (1994), Chakravarty and Keller (2000)). Like these languages, we provide a deterministic semantics and rely on the language implementation to spawn new tasks and synchronize on results. Unlike languages such as Concurrent ML (Reppy 1999) or JoCaml (Conchon and Fessant 1999), we provide no explicit concurrency constructs to the programmer. Manticore (Fluet et al. 2007) subsumes both paradigms and provides for both implicit parallelism and explicit concurrency.

Profiling In their seminal work on space profiling for lazy functional programs, Runciman and Wakeling (1993a) demonstrate the use of a profiler to reduce the space use of a functional program by more than two orders of magnitude. Like the current work, they measure space use by looking at live data in the heap. However, their tool is tied to a particular sequential implementation. Sansom and Peyton Jones (1993, 1995) extend this work with a notion of “cost center” that enables the programmer to designate how resource use is attributed to different parts of the source program.

There has been a series of works on profiling methods and tools for parallel functional programs (Hammond and Peyton Jones 1992; Runciman and Wakeling 1993b; Hammond et al. 1995; Charles and Runciman 1998). This work focuses on the overhead of parallel execution instead of how different parallel implementations affect the performance of the application. None of this work measures how different scheduling policies affect the space use of applications.

Scheduling Scheduling policies and their effects on space use have been studied extensively in the algorithms community (*e.g.,* Blumofe and Leiserson 1998; Blelloch et al. 1999). Our representation of parallel tasks as directed graphs is inspired by this work. However, we use these graphs as part of a formal semantics rather than simply an abstract model of computation.

Most implementations of data parallel languages have provided only a single scheduling policy that is either left unspecified or fixed as part of a compilation technique. In contrast, Fluet et al. (2007) make scheduling policies explicit using a *coordination language*. It would be interesting to see how our cost semantics could serve as a specification for schedulers written in this language.

Evaluation strategies (Trinder et al. 1998) enable the programmer to explicitly control the parallel evaluation structure (*e.g.,* divide-and-conquer). Like much of the work on profiling parallel functional programs, this focuses on when to spawn to parallel tasks and how much work to perform in each task (*i.e.,* granularity) instead of the order in which parallel tasks are evaluated.

In the course of their profiling work, Hammond and Peyton Jones (1992) considered two different scheduling policies. While one uses a LIFO strategy and the other FIFO, both use an evaluation strategy that shares many attributes with a work-stealing policy. These authors found that for their implementation, the choice of policy did not usually affect the running time of programs, with the exception in the case where they also throttled the creation of new parallel threads. In this case, the LIFO scheme gave better results. Except for the size of the thread pool itself, they did not consider the effect of policy on space use.

Cost Semantics A cost semantics was first used as a basis for profiling by Sansom and Peyton Jones (1995). They describe a method of profiling both time and space for programs written in a sequential, lazy language. As in our work, they use the cost semantics as a formal specification of performance. However, they use the semantics only as guide for their profiler: profiling results are derived from an instrumented version of the actual language implementation.

A cost semantics similar to the one used here was introduced by Blelloch and Greiner (1996). That work gave an upper bound on space use and assumed a fixed scheduling policy (depth-first). Our semantics extends this work by adding heap edges to the cost associated with each closed program. This enables us to reason about different scheduling policies and attribute space use to different parts of the program.

Lazy, purely functional programs can be evaluated in many different ways, and different strategies for evaluation can yield wildly different performance results. Ennals (2004) uses a cost semantics to compare the work performed by a range of sequential evaluation strategies, ranging from lazy to eager. Like the current work, he also uses cost graphs with distinguished types of edges, though his edges serve different purposes. He does not formalize the use of space by these different strategies. Likewise, program transformations that change the order of evaluation can also affect performance. Gustavsson and Sands (1999) give a semantic definition of what it means for a transformation to be “safe-for-space” (Shao and Appel 1994). They provide several laws to help prove that a given transformation does not asymptotically increase the space usage of sequential, call-by-need programs.

Jay et al. (1997) describe a static framework for reasoning about the costs of parallel execution using a monadic language. Static cost models have also been used to automatically choose a parallel implementation at compile-time based on hardware performance parameters (Hammond et al. 2003) and to inform the granularity of scheduling (Loidl and Hammond 1996). This work complements ours in that it focuses on how the sizes of program data structures affect parallel execution (*e.g.,* through communication costs), rather than how different execution models affect the use of space at a given point in time.

9. Conclusion

We have described and demonstrated the use of a semantic space profiler for parallel functional programs. One beauty of functional programming is that it isolates programmers from gritty details of the implementation and the target architecture, whether that architecture is sequential or parallel. However, when profiling functional programs, and especially *parallel* functional programs, there is a tension between providing information that relates to the source code and information that accurately reflects the implementation.

In our profiling framework, a cost semantics plays a critical role in balancing that tradeoff.

We have focused on using our framework to measure and reason about the performance effects of different scheduling policies. One possible direction for future work is to study other important aspects of a parallel implementation such as task granularity. We believe there is a natural way to fit this within our framework, by viewing task granularity as an aspect of scheduling policy.

We invite readers to download and experiment with our prototype implementation, available from the first author's website[4] or the `shared-heap-multicore` branch of the MLton repository.

References

Shail Aditya, Arvind, Jan-Willem Maessen, and Lennart Augustsson. Semantics of pH: A parallel dialect of Haskell. Technical Report Computation Structures Group Memo 377-1, MIT, June 1995.

Andrew W. Appel, Bruce Duba, and David B. MacQueen. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989. ISSN 0164-0925.

Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. of the ACM*, 46(2): 281–321, 1999.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994. ISSN 0743-7315.

Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. of Comp.*, 27(1):202–229, 1998.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. of the ACM*, 46(5):720–748, 1999.

Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 94–105, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-202-6.

Nathan Charles and Colin Runciman. An interactive approach to profiling parallel functional programs. In *Implementation of Functional Languages, 10th International Workshop, IFL'98, London, UK, September 9-11, Selected Papers*, pages 20–37, 1998.

Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. *SIGPLAN Not.*, 36(5):125–136, 2001. ISSN 0362-1340.

Silvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, page 22, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0340-3.

Robert Ennals. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, University of Cambridge, 2004.

Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Status report: the manticore project. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 15–24, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-676-9.

Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. In *Proceedings of Workshop on Higher Order Operational Techniques in Semantics*, number volume 26 of Electronic Notes in Theoretical Computer Science, September 1999.

Kevin Hammond and Simon L. Peyton Jones. Profiling scheduling strategies on the GRIP multiprocessor. In *International. Workshop on the Parallel Implementation of Functional Languages*, pages 73–98, RWTH Aachen, Germany, September 1992.

[4] `http://www.cs.cmu.edu/~spoons/parallel/`

Kevin Hammond, Hans-Wolfgang Loidl, and Andrew S. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In A. P. Wim Böhm and John T. Feo, editors, *HPFC'95 – High Performance Functional Computing*, pages 208–221, 1995.

Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic skeletons in template haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003.

C. Barry Jay, Murray Cole, M. Sekanina, and Paul Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 650–661, London, UK, 1997. Springer-Verlag. ISBN 3-540-63440-1.

Hans-Wolfgang Loidl and Kevin Hammond. A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 1996.

Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-48089-2.

Niklas Röjemo and Colin Runciman. Lag, drag, void and use–heap profiling and space-efficient compilation revisited. *SIGPLAN Not.*, 31(6):34–41, 1996. ISSN 0362-1340.

Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *J. Funct. Program.*, 3(2):217–245, 1993a.

Colin Runciman and David Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *Functional Programming, Glasgow '93*, pages 236–251. Springer-Verlag, 1993b.

Patrick M. Sansom and Simon L. Peyton Jones. Profiling lazy functional programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 227–239, London, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.

Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–366, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 150–161, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-643-3.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. Technical Report CMU-CS-08-110, Carnegie Mellon University, April 2008.

Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.

Stephen Weeks. Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-483-9.

A. Definitions

Locations The location of a value is the outermost label of that value and serves to uniquely identify that value. The locations of an expression are the labels of any values that appear in that expression.

$$
\begin{array}{lcl}
\mathrm{loc}(\langle f.x.e\rangle^{\ell}) & = & \ell \\
\mathrm{loc}(\langle v_1, v_2\rangle^{\ell}) & = & \ell \\
\mathrm{loc}(\mathsf{true}^{\ell}) & = & \ell \\
\mathrm{loc}(\mathsf{false}^{\ell}) & = & \ell \\
\\
\mathrm{locs}(\mathsf{fun}\ f.x.e) & = & \mathrm{locs}(e) \\
\mathrm{locs}(e_1\ e_2) & = & \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \\
\mathrm{locs}(\{e_1, e_2\}) & = & \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \\
\mathrm{locs}(\#i\ e) & = & \mathrm{locs}(e) \\
\mathrm{locs}(\mathsf{true}) & = & \emptyset \\
\mathrm{locs}(\mathsf{false}) & = & \emptyset \\
\mathrm{locs}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \cup \mathrm{locs}(e_3)
\end{array}
$$