JFP **20** (5 & 6): 417–461, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000146 First published online 3 September 2010

Space profiling for parallel functional programs

DANIEL SPOONHOWER, GUY E. BLELLOCH, ROBERT HARPER
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(*e-mail:* {spoons,blelloch,rwh}@cs.cmu.edu)

PHILLIP B. GIBBONS
Intel Labs Pittsburgh, 4720 Forbes Avenue, Suite 410, Pittsburgh, PA 15213, USA
(*e-mail:* phillip.b.gibbons@intel.com)

Abstract

We present 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 while hiding 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.

1 Introduction

Approaches to multicore and multiprocessor programming can be divided into two broad categories: concurrency and parallelism. In the case of **concurrency**, programmers reason explicitly about the order in which tasks are executed and must use some form of synchronization to ensure that programs yield correct results. In the case of **parallelism**, multicore execution is purely a means to improve performance: programmers provide opportunities for parallelism, but the results of program execution are defined independently of whether or not those opportunities are taken.

While some problems require concurrency (e.g., applications that must respond to network requests or user interaction), many algorithms can be expressed using only parallelism (e.g., those based on divide-and-conquer strategies). Reasoning about and debugging parallel programs can be performed independently of the number of processors or processor cores and is therefore much simpler than for concurrent programs. There are also many opportunities for profiling parallel programs with only a few assumptions about how tasks will be assigned to processor cores. We will investigate several such opportunities in this work.

Functional programming languages are an attractive means for writing parallel programs. By eschewing side effects, programmers can find many opportunities for parallel evaluation. Many sequential functional programming languages also offer a clear semantic definition that can serve as a foundation for a semantic definition of parallelism.

In practice, however, 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 & Wakeling 1993a; Sansom & Peyton Jones 1995; Röjemo & 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.

We present a **semantic space profiler** for a call-by-value parallel functional language and implementations of that language for shared memory architectures. Our method abstracts away from many details of the 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 (Blelloch & Greiner 1995; Sansom & Peyton Jones 1995). 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). One of these graphs is used to record sequential dependencies; parallel pebblings of this graph correspond to different scheduling policies. The second graph, novel to this work, records dependencies between parts of the program state.

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).

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

We emphasize that our method enables users to profile parallel *programs*. This stands in contrast to many existing profilers, which provide a means of profiling a program *based only on a particular language implementation*. While our approach 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 enables us to balance this tradeoff.

Our cost semantics also provides a formal specification that forces language implementations to be "safe-for-space" (Shao & 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 (Section 5) of a parallel extension of Standard ML (Milner *et al.* 1997) based on our cost semantics and the MLton optimizing compiler (Weeks 2006). 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 Data Parallel Haskell (Peyton Jones *et al.* 2008). By factoring out scheduling, our framework can also bring to light performance issues in language implementations that bake in a particular scheduling policy.

Our implementation includes three scheduling policies: two based on parallel breadth- and depth-first traversals of the cost graphs and a third based on work stealing (Burton & Sleep 1981; Blumofe & Leiserson 1999). As we 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).

Using our semantics, we also discovered a space leak in an optimization in MLton (Section 7.2). As a specification, a cost semantics determines which performance problems must be blamed on the programmer, and which can be attributed to the language implementation.

A preliminary version of this paper appeared in the *Proceedings of the 2008 International Conference on Functional Programming*.

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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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

fun mm_mult (A, B) =
  let
    fun vv_mult (b, a) = reducei op+ (fn (i, x) ⇒ sub (b, i) ∗ x) 0.0 a
    fun mv_mult (B, a) = tabulate (fn i ⇒ vv_mult (sub (B, i), a), length B)
  in
    tabulate (fn i ⇒ mv_mult (B, sub (A, i)), length A)
  end
```

Fig. 1. Matrix multiplication code.

on space use. We give a cursory discussion here, and consider this application in further detail in Section 6, along with four other applications (sorting, convex hull, n-body simulation, and the n-queens problem).

Figure 1 shows the source code for matrix multiplication written in our parallel extension of ML. This example defines two top-level functions. The first is a parallel version of tabulate and the second, mm_mult, multiplies two matrices of appropriate sizes using persistent vectors. Within the definition of mm_mult, two additional functions are defined. The first, vv_mult, computes the dot product of two vectors. The second, mv_mult, computes a matrix-vector product. In this example, variables A and B stand for matrices, a and b for vectors, and x for scalars. Persistent vectors are created with one of two primitives. The first primitive, vector (s, x), creates a vector of size s where every element is given by the value of x. The other primitive, append, creates a new vector that is the result of appending its arguments. (For simplicity, this implementation of multiplication assumes that one of its arguments is in row-major form while the other is in column-major form.)

The parallelism in this example is derived in part from the implementation of tabulate. As in an ordinary implementation of tabulate, this function returns a vector where each element is initialized using the function supplied as an argument. In this implementation, whenever the result will have two or more elements, each half of the result may be built in parallel. In our parallel language, the two components of a pair written with curly braces ({ . . . }) may be computed in parallel. Thus the two recursive calls to loop in the body of tabulate maybe computed in parallel.

Matrix multiplication offers a tremendous amount of potential parallelism. In this example, multiplying two $n \times n$ matrices may result in n^3 scalar multiplications occurring in parallel. (Recall that there are n scalar multiplications for each of the n^2 elements in the result.)

Not shown here are the implementations of several other functions on vectors, including length, which returns the length of its argument, and reducei, which is a form of fold but may reduce two halves of a vector in parallel before combining the results.

With this example in hand, we will now briefly consider this example in a broader context of parallel programs, how these programs can be implemented, and the performance consequences of these implementation choices.

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:

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 graph on the left, 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 graph on the right, 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 (i.e., the smallest heap that can be used to run the program). It also determines the point during execution at which this mark is reached, as well as where in the source code these data are 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 tools 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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

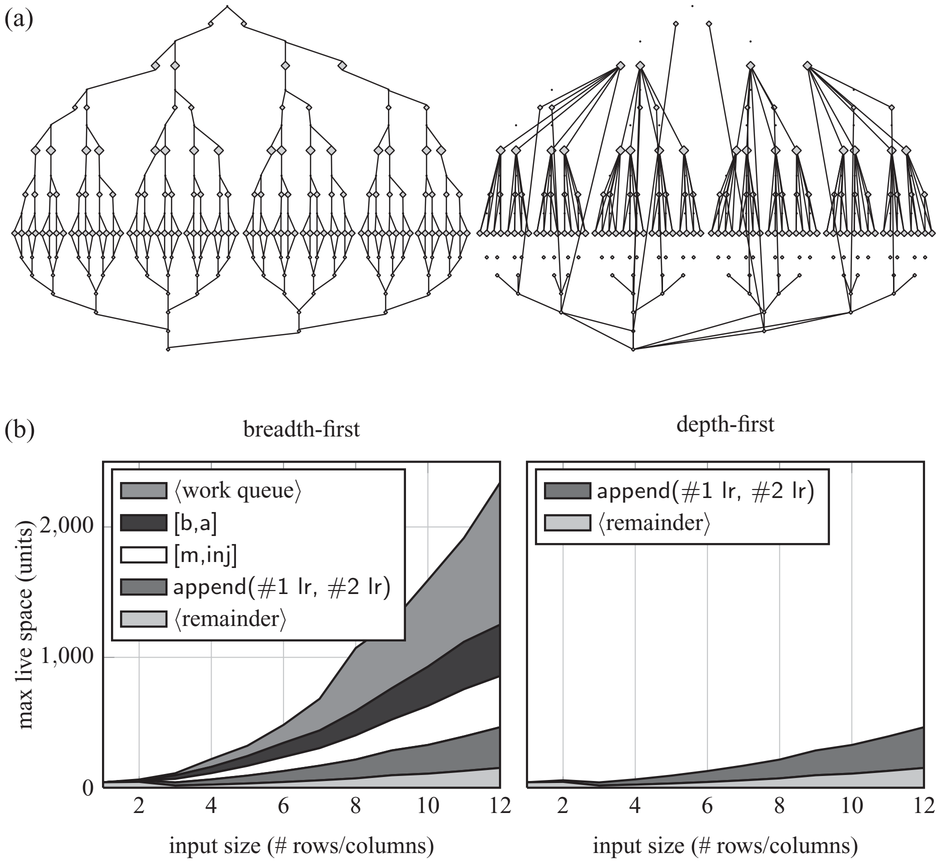


Fig. 2. Profiler output for matrix multiplication. (a) Summarized cost graphs; and (b) simulated space use as a function of input size for two scheduling policies: breadth-first (left) and depth-first (right).

implements a breadth-first traversal of the left cost graph. The scheduling policy on the right implements a parallel depth-first traversal (see Section 5.2 for details) of the left 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 vectors 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 reducei.

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

$$
\begin{array}{llll}
\text{(expressions)} & e & ::= & x \mid \mathsf{fun}\ f(x) = e \mid e_1\ e_2 \mid \{e_1, e_2\} \mid \#1\ e \mid \#2\ e \\
 & & & \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid v \\
\text{(values)} & v & ::= & \langle f.x.e \rangle^{\ell} \mid\ v_1, v_2 \rangle^{\ell} \mid \mathsf{true}^{\ell} \mid \mathsf{false}^{\ell} \\
\text{(locations)} & \ell & \in & L
\end{array}
$$

Fig. 3. Language syntax. 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. As in Standard ML, projections from a pair are denoted #1 and #2 .

> 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 how programs can be modified to improve performance. In our matrix multiplication example (right 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 vv_mult and correspond to the top three entries in the left plot of Figure 2(b).

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

3 Cost semantics

The cost semantics 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 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 of eager languages, 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 3 shows the syntax of the call-by-value language we discuss in this section. In addition to variables x, recursive functions $\mathsf{fun}\ f(x) = e$, and function application $e_1\ e_2$, it includes syntax for constructing pairs $\{e_1, e_2\}$ in parallel. As in Standard

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

ML, components of a pair may be projected using functions #1 and #2. We also include booleans and conditional expressions as they will play an interesting role in the cost semantics.

The syntax of values v (including functions, pairs, and booleans) is distinguished from the syntax of the corresponding source expressions: programmers never write values in their code. Associated with each value is a location ℓ drawn from an infinite set of locations L. The location of a value $\mathrm{loc}(v)$ is defined in Figure 6. Locations are used to express sharing and will be important in reasoning about the space use of programs. For example, given the pair value $\langle v_1, v_2\rangle^{\ell}$, if $\mathrm{loc}(v_1) = \mathrm{loc}(v_2)$ then both components occupy the same space. If $\mathrm{loc}(v_1) \neq \mathrm{loc}(v_2)$ then the two components are represented separately. For example, the two components may be extensionally identical (i.e., they are indistinguishable using language primitives) but represented using disjoint parts of the heap. We refer to the subset of expressions that does not contain any values as source expressions.

In the implementation of our profiler, we extend this fragment with integers, floating-point numbers, lists, trees, and vectors, but none of these extensions prove difficult. We also include an additional form of pairs that always evaluates components of the pair sequentially. The choice between parallel and sequential pairs allows programmers to control the granularity of parallel tasks explicitly. Finally, we assume that all values are allocated in the heap for the sake of simplicity, 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 single 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.*

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

This judgment is defined in Figure 4. 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 values.

Computation Graphs. Each node in the computation graph represents a single reduction of some subexpression, or more specifically an event associated with that reduction. Edges represent constraints on the order in which these events can occur. Computation graphs are similar to the graphs introduced by Blumofe & Leiserson (1998) and those used by Greiner & Blelloch (1999).

For the programs considered in this article, each computation graph is a series-parallel graph. Each such graph consists of a single node, or a sequential or parallel composition of smaller graphs. Every series-parallel graph has a unique **initial node**

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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

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

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

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

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

$$\frac{(\ell\ \text{fresh})}{\mathsf{true} \Downarrow \mathsf{true}^{\ell}; [\ell]; \emptyset}\ \textsc{c-True} \qquad \frac{(\ell\ \text{fresh})}{\mathsf{false} \Downarrow \mathsf{false}^{\ell}; [\ell]; \emptyset}\ \textsc{c-False}$$

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

$$\frac{e_1 \Downarrow \mathsf{false}^{\ell_1}; g_1; h_1 \qquad e_3 \Downarrow v_3; g_3; h_3 \qquad (n\ \text{fresh})}{\begin{array}{c} \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v_3; g_1 \oplus [n] \oplus g_3; \\ h_1 \cup h_3 \cup \{(n, \ell_1)\} \cup \bigcup_{\ell \in \mathrm{locs}(e_2)} (n, \ell) \end{array}}\ \textsc{c-IfFalse}$$

Fig. 4. Cost semantics. This semantics yields two graphs that can be used to reason about the parallel performance of programs. The substitution $[v/x]$ of a value v for a variable x is a capture-avoiding substitution defined in Figure 5. The outermost locations of values $\mathrm{loc}(v)$ and expressions $\mathrm{locs}(e)$ are defined in Figure 6. Parallel composition takes precedence over serial composition so $g_1 \otimes g_2 \oplus g_3$ should be read as $(g_1 \otimes g_2) \oplus g_3$.

that precedes all other nodes. Dually, every series-parallel graph also has a unique **final node** that is preceded by all other nodes. Nodes are denoted ℓ and n (and variants). Graph are written as tuples such as $(n; n'; E)$ where n is the initial node, n' is the final node, and E is a set of edges. The remaining nodes of the graph are implicitly defined by the edge list. The primitive operations used to build computation graphs are shown in Figure 7. (In this figure and elsewhere, a gray diamond stands for an arbitrary subgraph.) A graph consisting of a single node n is written $(n; n; \emptyset)$ or more briefly $[n]$. In the parallel composition (on the right),

$$
\begin{array}{lcll}
[v/x]x & = & v & \\
[v/x]y & = & y & (\text{if } x \neq y) \\
[v/x]\mathsf{fun}\ f(y) = e & = & \mathsf{fun}\ f(y) = [v/x]e & (\text{if } x \neq y \text{ and } x \neq f) \\
[v/x]e_1\ e_2 & = & [v/x]e_1\ [v/x]e_2 & \\
[v/x]\{e_1, e_2\} & = & \{[v/x]e_1, [v/x]e_2\} & \\
[v/x](\#i\ e) & = & \#i\ ([v/x]e) & \\
[v/x]\mathsf{true} & = & \mathsf{true} & \\
[v/x]\mathsf{false} & = & \mathsf{false} & \\
[v/x]\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & = & \mathsf{if}\ [v/x]e_1\ \mathsf{then}\ [v/x]e_2\ \mathsf{else}\ [v/x]e_3 & \\
[v/x]v' & = & v' & (\text{where } v' \text{ is a value})
\end{array}
$$

Fig. 5. Substitution. These equations define substitution for our semantics. This is a standard, capture-avoiding substitution.

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

Fig. 6. Locations of values and expressions. Locations are used in measuring the space required to represent a value and can appear in expressions after a substitution has occurred.

the node n is called the **fork** point and the node n' is called the **join** point. We identify graphs up to reordering of nodes and edges: binary operations $\oplus$ and $\otimes$ on computation graphs are commutative and associative. As suggested by the notation, $\otimes$ is assigned higher precedence than $\oplus$.

Edges in the computation graph point forward in time: an edge from node n_1 to node n_2 indicates that the expression corresponding to n_1 must be evaluated before that corresponding to n_2. In our figures, nodes will be arranged so that time passes from top to bottom. We follow standard terminology and say that if there is an edge from n_1 to n_2 then n_1 is a parent of n_2 and n_2 is a child of n_1. If there is a non-empty path from n_1 to n_2, then we write $n_1 \prec_g n_2$ and say that n_1 is a predecessor of n_2, and that n_2 is a descendant of n_1.

Heap Graphs. We extend the work of Greiner & Blelloch (1999) with heap graphs. Heap graphs are also directed, acyclic graphs but are not necessarily series-parallel

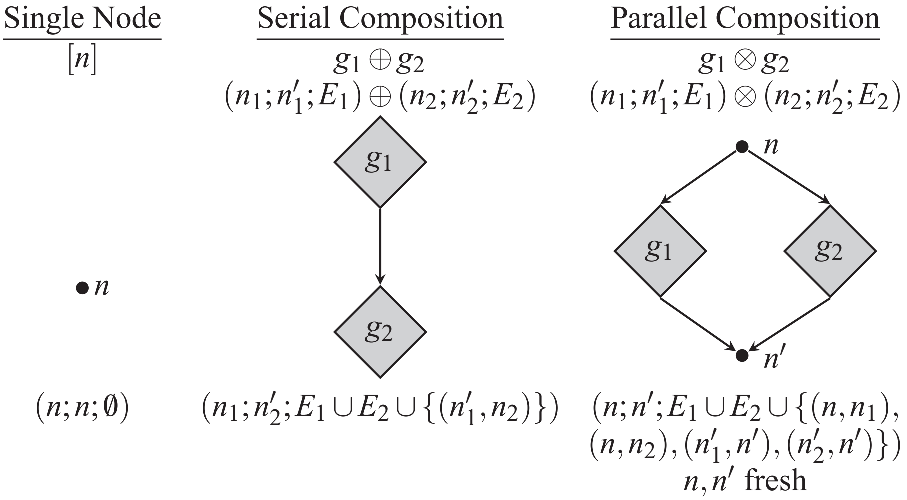

Single Node
$[n]$
n
$(n;n;\emptyset)$
Serial Composition
$g_1 \oplus g_2$
$(n_1;n'_1;E_1) \oplus (n_2;n'_2;E_2)$
g_1
g_2
$(n_1;n'_2;E_1 \cup E_2 \cup \{(n'_1,n_2)\})$
Parallel Composition
$g_1 \otimes g_2$
$(n_1;n'_1;E_1) \otimes (n_2;n'_2;E_2)$
n
g_1
g_2
n'
$(n;n';E_1 \cup E_2 \cup \{(n,n_1),$
$(n,n_2),(n'_1,n'),(n'_2,n')\})$
n,n' fresh

Fig. 7. Graph operations for pairs. The graphs associated with our language (a language with parallel pairs) are built from single nodes as well as the serial and parallel composition of smaller graphs.

graphs. Each node again represents the evaluation of a subexpression. For source expressions, 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. While edges in the computation graph point forward in time, edges in the heap graph point backward in time.

Each edge in the heap graph represents a dependency on a value: if there is an edge from n to ℓ then n depends on the value with location ℓ. The sink of a heap edge will always be a node ℓ corresponding to the value allocated at that point in time. The source of a heap edge may be either a node ℓ' (indicating a dependency among values) or node n (indicating a dependency on a value by an arbitrary computation). In the first case, the memory associated with ℓ cannot be reclaimed while the value represented by ℓ' is still in use. In the second case (where n represents a point in the evaluation of an expression) the memory associated with ℓ cannot be reclaimed until after the expression corresponding to n has been evaluated. Heap graphs will be written as a set of edges. As above, the nodes are left implicit. The evaluation rule for parallel pairs (c-Fork) is an example of a dependency between values: 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 (c-App), however, two edges are added to express the *uses* of 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 nonstrict functions, this is the first point at which the garbage collector might reclaim the space associated with the argument. As discussed in Section 7, these edges are critical in ensuring that the specification given by the cost semantics can be implemented.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

A similar concern arises in the rules for conditionals. In c-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. These edges represent our imperfect knowledge of program behavior at runtime: we cannot, in general, predict which branch will be taken nor which values may be reclaimed. The rule c-IfFalse is analogous. These rules will be discussed in more detail after we have made a more concrete connection between heap graphs and space use.

While there is some flexibility in designing these rules, 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, as the implications of rules can be 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. For a given program, each policy will give rise to a particular parallel execution that is constrained by the computation graph g (as well as other parameters such as the number of processors).

Definition 1 (*Schedule Order*)
A **schedule order** $\trianglelefteq$ (or more briefly a **schedule**) for a computation graph g is defined by a partial order on the nodes of g such that,

- $\forall n \in \text{nodes}(g).\ n \trianglelefteq n$,
- $\forall n_1, n_2 \in \text{nodes}(g).\ n_1 \prec_g n_2 \Rightarrow n_1 \vartriangleleft n_2$,

Where $\vartriangleleft$ is the corresponding strict partial order given by the reflexive reduction of $\trianglelefteq$.

An instance $n_1 \trianglelefteq n_2$ of this partial order can be read as "n_1 is scheduled no later than n_2." The definition above requires that every node in g is scheduled. It also requires that the corresponding strict partial order is a super-order of the precedence relation given by the edges of g. It follows that the initial node of a graph is the always the first node scheduled, and the final node is always the last node scheduled.

We can extend a schedule order from individual nodes to sets of nodes in a straightforward manner. Given two sets of nodes N_1 and N_2, N_1 is scheduled before N_2 if every node in N_1 is scheduled before every node in N_2.

$$N_1 \vartriangleleft N_2 \quad \text{iff} \quad \forall n_1 \in N_1, n_2 \in N_2.\ n_1 \vartriangleleft n_2$$

The partial order defining a schedule gives a global view of all the events that occur during the evaluation of a program. In some cases, it will be useful to consider more narrow views of the events that occur in a schedule. For example, we can view a schedule from a particular moment in time. Two nodes n_1 and n_2 are scheduled **simultaneously** (written $n_1 \bowtie n_2$) if neither n_1 nor n_2 is scheduled after the other.

$$n_1 \bowtie n_2 \quad \text{iff} \quad n_1 \trianglelefteq\ n_2 \wedge n_2 \trianglelefteq\ n_1$$

We shall use this notion of simultaneity to reason about space use in the next section.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

3.3 Roots

To understand the space use of a schedule $\trianglelefteq$, we first partition the set of nodes in the computation graph into a sequence of sets of simultaneously scheduled nodes. We call such a partition the **steps** of the schedule.

$$\text{steps}(\trianglelefteq) = N_1, \ldots, N_k \text{ such that } \begin{cases} \forall n \in \text{nodes}(g).\ \exists N_i . n \in N_i, \\ N_1 \vartriangleleft \cdots \vartriangleleft N_k, \text{ and} \\ \forall 1 \leqslant i \leqslant k, \forall n_1, n_2 \in N_i .\ n_1 \bowtie n_2 \end{cases}$$

The first two conditions ensure that these sets define a partition: no node is related to itself by the strict partial order $\vartriangleleft$. This partition is unique because every node in N_i is scheduled simultaneously and is therefore incomparable with every other node in N_i. Thus no node in N_i can be moved to any other set that is scheduled before or after N_i.

For any step N_i, we define the closure $\widehat{N_i}$ as the set of all the nodes that have been scheduled up to and including that step.

$$\widehat{N_i} = \bigcup_{1 \leqslant j \leqslant i} N_j$$

While nodes in $\widehat{N_i}$ are not necessarily scheduled simultaneously, it is still the case that $\widehat{N_i} \vartriangleleft N_{i+1}$. With these sets in mind, we can think of a schedule as defining a wavefront that advances across the computation graph, visiting some set of nodes at each step. We say that a node n is **ready** at step N_{i+1} if all of the parents of n appear in $\widehat{N_i}$ but $n \notin \widehat{N_i}$.

Given a schedule $\trianglelefteq$ of a graph g with steps $N_1, \ldots, N_k$, consider the moment in time represented by some N_i. Because $\widehat{N_i}$ contains all previously scheduled 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.

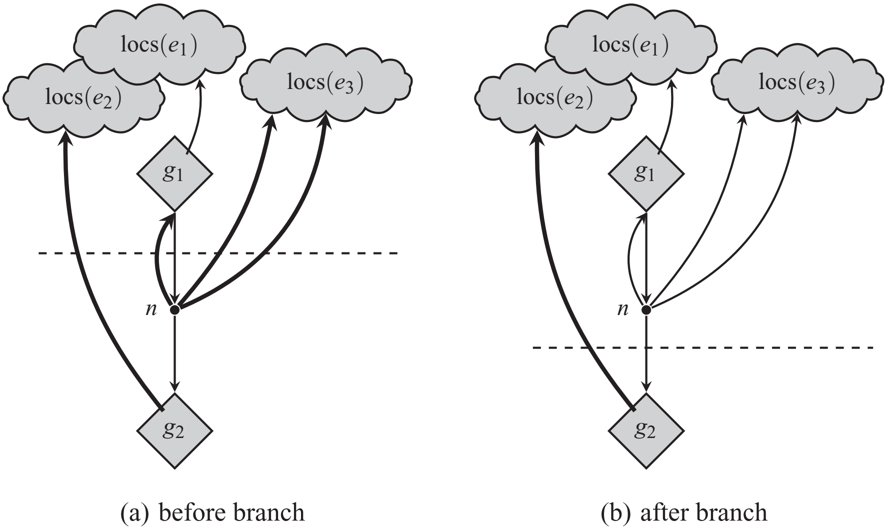
- Both $n, \ell \notin \widehat{N_i}$. As the value associated with ℓ has not yet been allocated, the edge (n, ℓ) does not contribute to the use of space at step N_i.
- Both $n, \ell \in \widehat{N_i}$. While the value associated with ℓ has been allocated, the use of this value represented by this edge is also in the past. Again, the edge (n, ℓ) does not contribute to the use of space at step N_i.
- We have $\ell \in \widehat{N_i}$, but $n \notin \widehat{N_i}$. The value associated with ℓ has been allocated, and n represents a possible use in the future. The edge (n, ℓ) *does* contribute to the use of space at step N_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 2 (*Roots*)
Given $e \Downarrow v; g; h$, the **roots** after scheduling the nodes in N and with respect to location ℓ, written $\text{roots}_{h,\ell}(N)$, is the set of nodes $\ell' \in N$ where $\ell' = \ell$ or h contains an edge leading from outside N to ℓ'. Symbolically,

$$\text{roots}_{h,\ell}(N) = \{\ell' \in N \mid \ell' = \ell \vee (\exists n.(n, \ell') \in h \wedge n \notin N)\}$$

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press



(a) before branch (b) after branch

Fig. 8. Modeling space use for conditional expressions. This figure shows cost graphs for a conditional expression evaluated using the rule c-IfTrue. Each part shows one step of a schedule of these graphs: before (a) and after (b) the branch is taken. A dashed line divides those nodes that have been scheduled from those that have not, and heap edges that point to roots are drawn with heavy lines.

The location of a value loc(v) is the outermost location of the value as defined in Figure 6, and serves to uniquely identify that value. The locations of an expression locs(e) are the locations of any values that appear in that expression as a result of substitution. As we will often be interested in the roots with respect to a *value*, we write $\text{roots}_{h,v}(N)$ as an abbreviation for $\text{roots}_{h,\text{loc}(v)}(N)$.

We use the term *roots* to evoke a related concept from work in garbage collection. For a reader that 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 a parallel implementation, it also includes those locations that are reachable directly from the scheduler queue.

Roots must be defined "with respect to a location" so that the final result of evaluating an expression is not lost. This corresponds to an observation of that result. Thus $\text{roots}_{h,v}(N)$ will account for the cost of representing v even if v is not used in the remainder of the computation.

As an example, Figure 8 shows simplified cost graphs derived from an expression of the form if e_1 then e_2 else e_3 and an application of the rule c-IfTrue. The two parts of the figure depict two distinct points in time for a single schedule of these graphs. In both parts, a dashed line divides those nodes that have been scheduled from those that have not. As in other figures, edges in a computation graph run from top to bottom, while edges in a heap graph run from bottom to top. Those heap edges that point to roots are drawn as thicker lines.

By assumption, e_1 evaluates to true (or more precisely, a value true^{ℓ_1}). This computation is represented by the subgraph g_1. Next, e_2 is evaluated (as represented

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

by the subgraph g_2). The three clouds at the top of each graph represent the sets of nodes in the heap graph corresponding to the locations of the expressions e_1, e_2, and e_3. (We assume that this conditional expression appears within the context of a larger expression and that this conditional may have subexpressions which are values.) The node labeled n represents the point at which the condition is tested and the branch is taken. Recall that the cost semantics adds heap edges from n to the locations of the branch that is *not* taken (here, e_3).

In Part (a), n has not yet been scheduled. At this point in time, the roots include the locations of e_3 as well as those of e_2 and the value that is being tested. Each of the four heavy heap edges in the figure points to one such root. A garbage collector cannot (in general) determine which values will actually be required in the remainder of the computation and therefore must preserve any reachable values until the branch is taken. The cost semantics admits such an implementation by including all these locations among the roots.

Part (b) depicts the point in time just after n has been scheduled, or equivalently, after the branch has been taken. At this point, a garbage collector may reclaim space used to represent any values that appear in e_3 (but do not appear in e_2). The figure shows how this is reflected in the cost semantics: the edges from n to the locations of e_3 are no longer roots.

It is possible to define a cost semantics that does *not* include edges pointing to the values of the branch not taken (e.g., those from n to the $\mathrm{locs}(e_3)$). However, building an implementation faithful to such a cost semantics would be impossible: it would require the language implementation (and in particular, the garbage collector) to predict which branch will be taken for each conditional expression in the program. Furthermore, it must make such predictions without consuming any additional space. These additional edges distinguish what might be called “true” garbage from “provable” garbage: though we would like to measure the memory required by only those values that will actually be used in the remainder of the computation, we cannot expect an implementation to efficiently determine that set of values. Instead, we settle for an approximation of those values based on reachability.

Our cost semantics maintains an invariant that at each point in a schedule, the roots of the heap graph correspond to values that must be preserved given information available *at that point in time*. Thus the heap graph encodes not only information about values in the heap, but also information about *uses* of those values over time.

The following theorem serves as a sanity check for our definition of roots and of the heap graphs defined in the cost semantics. Informally, it states that, given an expression e and cost graphs g and h, the locations of e are the same as the roots at the start of the evaluation of e. (When considered in the context of a larger computation, all of the locations of e will be scheduled before any of the nodes of g. Thus $\mathrm{roots}_{h,v}(\mathrm{locs}(e))$ is the smallest set of roots at the start of the evaluation of e in any parallel schedule.) If one of the locations in $\mathrm{locs}(e)$ did *not* appear in the set of roots, then either our definition of roots is incorrect or there are edges missing from the heap graph.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Theorem 1 (*Initial Roots*)
If $e \Downarrow v\,;g\,;h$ then $\mathrm{roots}_{h,v}(\mathrm{locs}(e)) = \mathrm{locs}(e)$.

Proof
By induction on the given derivation.
Case C-FORK**:** Here, we have $\mathrm{locs}(e) = \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2)$. For $i \in \{1,2\}$, the induction hypothesis yields $\mathrm{roots}_{h_i,v_i}(\mathrm{locs}(e_i)) = \mathrm{locs}(e_i)$. It suffices to show that for each $\ell \in \mathrm{locs}(e)$ either $\ell = \mathrm{loc}(v)$ or that there exists an edge $(n,\ell) \in h$ such that $n \notin \mathrm{locs}(e)$. Consider one location ℓ' such that $\ell' \neq \mathrm{loc}(v)$. Without loss of generality, assume that $\ell' \in \mathrm{locs}(e_1)$, and therefore $\ell' \in \mathrm{roots}_{h_1,v_1}(\mathrm{locs}(e_1))$. It follows that either $\ell' = \mathrm{loc}(v_1)$ or there exists an edge $(n,\ell') \in h_1$ such that $n \notin \mathrm{locs}(e_1)$. Take the first case; by construction, there is an edge $(\ell, \mathrm{loc}(v_1))$ and $\ell \notin \mathrm{locs}(e)$ since ℓ was chosen to be fresh. In the second case, the edge (n,ℓ') is also in h and it remains to show that $n \notin \mathrm{locs}(e_2)$. Since $(n,\ell') \in h_1$ it follows that $n \in \mathrm{nodes}(g_1)$ and since nodes in the computation graph are chosen to be fresh, $n \notin \mathrm{locs}(e_2)$.
Case C-PROJ$_i$**:** In this case, $e = \#i\ e'$ and $\mathrm{locs}(e) = \mathrm{locs}(e')$. As a subderivation, we have $e' \Downarrow \langle v_1, v_2\rangle^{\ell}\,;g'\,;h'$ with $h = h' \cup (n,\ell)$. Inductively, $\mathrm{roots}_{h',\ell}(\mathrm{locs}(e')) = \mathrm{locs}(e')$. It is sufficient to show that for each $\ell' \in \mathrm{locs}(e')$ either $\ell' = \mathrm{loc}(v)$ or there exists an edge $(n',\ell') \in h$ such that $n' \notin \mathrm{locs}(e')$. Assume that $\ell' \neq \mathrm{loc}(v)$. Since $\ell' \in \mathrm{locs}(e')$ we have $\ell' \in \mathrm{roots}_{h',\ell}(\mathrm{locs}(e'))$. Then either $\ell' = \ell$ or there exists an edge $(n',\ell') \in h'$ such that $n' \notin \mathrm{locs}(e')$. In the first case, we have an edge $(n,\ell') \in h$ and since n was chosen to be fresh $n \notin \mathrm{locs}(e')$. In the second case, the required edge is also in h since $h' \subseteq h$.
Case C-VAL**:** Here, $\mathrm{locs}(e) = \{\mathrm{loc}(v)\}$. From the definition, $\mathrm{roots}_{h,v}(\{\mathrm{loc}(v)\}) = \{\mathrm{loc}(v)\}$.
Case C-FUN**:** In this case, $e = \mathsf{fun}\ f(x) = e'$ and $\mathrm{locs}(e) = \mathrm{locs}(e')$. Assume $\mathrm{loc}(v) = \ell$. As there are edges in h from ℓ to each $\ell' \in \mathrm{locs}(e')$, it follows that $\mathrm{roots}_{h,v}(\mathrm{locs}(e')) = \mathrm{locs}(e')$.
Case C-APP**:** In this case, $e = e_1\ e_2$. Each of the locations in $\mathrm{locs}(e)$ comes from e_1 or e_2. As in the case for C-FORK, we apply the inductive hypothesis to each subexpression and then show that each location in $\mathrm{locs}(e_i)$ is also in $\mathrm{roots}_{h,v}(e)$, either because of an edge appearing in one of the heap subgraphs or because of one of the edges added in this rule.
Case C-TRUE (and C-FALSE)**:** Here, $\mathrm{locs}(e)$ is empty, and the result is immediate.
Case C-IFTRUE (and analogously C-IFFALSE)**:** As the locations of a conditional expression may come from either branch, we must ensure that they are also in the initial roots. In C-IFTRUE, the locations of e_2 are included in the set of roots as has been shown inductively in previous cases. For each location ℓ in e_3, there is an edge from n to each ℓ and, because n does not appear in the locations of e, each such location is also in the roots of e. □

3.4 Space use

Roots describe only those values which are *immediately* reachable from the current program state. To understand the total space required to represent a value, we must consider all reachable nodes in the heap graph. We write $\ell_1 \preccurlyeq_h \ell_2$ if there is a (possibly empty) path from ℓ_1 to ℓ_2 in h. Below, we define the space required for the parallel evaluation of expressions. Though these definitions do not precisely account for the space required to represent values in some implementations of the language, they will still enable us to account for the asymptotic space use of programs. Moreover, an implementation may use *more* space than is described here, depending on some details of the language implementation (e.g., how frequently the garbage collector runs).

The first definition gives the space required to represent the state of parallel evaluation at a single moment in time. It accounts for all values that have been allocated and may be used in the remaining evaluation of the expression.

Definition 3 (*Space Required at a Step*)
The **space required at a step** N of a schedule that eventually computes a value v is the number of nodes in the heap graph h that are reachable starting from the nodes in $\mathrm{roots}_{h,v}(N)$.

$$\mathrm{space}_{v,h}(N) = |\{\ell \in h \mid \exists \ell' \in \mathrm{roots}_{h,v}(N).\ell' \preccurlyeq_h \ell\}|$$

Definition 4 (*Space Required by a Schedule*)
The **space required by a schedule** $\trianglelefteq$ that computes a value v is the maximum amount of space required in any step in that schedule.

$$\mathrm{space}_{v,h}(\trianglelefteq) = \max(\{\mathrm{space}_{v,h}(N) \mid N \in \mathrm{steps}(\trianglelefteq)\})$$

These definitions are implemented as part of our profiler and are used to generate the plots such as those in Figure 2(b) and in the following section. Plots that show the simulated high-water mark for space-use (such as Figure 2(b)) are derived as follows: given a program and input, the cost semantics is used to derive a value v, a computation graph g, and a heap graph h; a scheduling policy is then used to determine a schedule $\trianglelefteq$ for g; and the minimum required space is computed as $\mathrm{space}_{v,h}(\trianglelefteq)$.

3.5 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 of a language implementation. 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 allow some processor to remain idle even when there is work to be performed.

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, **greedy** scheduling policies are those that avoid leaving processors idle whenever possible. Using the definitions above, a schedule $\trianglelefteq$ that uses p processors is greedy if

$$\forall n_1, n_2.\ n_1 \triangleleft n_2 \Rightarrow \exists n_3.(n_3 \prec_g n_2 \wedge n_1 \bowtie n_3) \ \vee\ |\{n \mid n \bowtie n_1\}| = p$$

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

This states that a schedule that uses p processors is greedy if, whenever n_2 is scheduled after n_1 then either there was a sequential dependency to n_2 from some node scheduled simultaneously with n_1, or there were $p-1$ other nodes that were scheduled simultaneously with n_1.

Similarly, we can define the behavior of any scheduling policy that is based on a schedule for a single processor or a 1-schedule (Blelloch *et al.* 1999). This includes both the depth-first and breadth-first scheduling policies used in this work. A 1-schedule is defined by a strict linear order $<_1$ on the nodes of the computation graph. The following implication must hold for any parallel schedule $\trianglelefteq$ based on such a 1-schedule.

$$\forall n_1, n_2.\ n_1 \lhd n_2 \Rightarrow n_1 <_1 n_2 \ \vee \ \exists n_3.(n_3 \prec_g n_2 \wedge n_1 \bowtie n_3)$$

This implication states that whenever a node n_1 is scheduled before node n_2 then either n_1 came before n_2 in the strict order $<_1$ or there was a predecessor of n_2 that was scheduled simultaneously with n_1.

Though it is beyond the scope of the current work, one can also give an implementation of our parallel language that makes the scheduling policy explicit. For example, one can present such an implementation using a small-step parallel semantics. Such a semantics is known as a provably efficient implementation (Greiner & Blelloch 1999) because each transition can be related to a step in a schedule of the corresponding cost graphs. The constraints described in this section serve as concise specifications for these implementations.

4 Profiling

Profiling has long been used as a means for improving the performance of programs, and space profiling has been used to great effect in improving the performance of sequential call-by-need programs (Runciman & Wakeling 1993a). Measurements are usually taken by instrumenting either the source code or the runtime system and then running the program. One issue that arises is that measurements derived using this method reflect not only the original program but also the compiler and runtime system. In the case of our implementation in MLton and for one particular program, there was a significant difference in space use between the original and the compiled versions of that program, due to a space leak introduced by one of the existing optimizations in MLton.

We address this issue by considering an alternative method of profiling. **Semantic profiling** relies only on the semantics of the source program. While more traditional profiling can be viewed as a set of measurements about a program together with a particular implementation of the language, semantic profiling gives results independently of any implementation. Using the profiling tools described in this section, we discovered and helped to fix the aforementioned bug in MLton (see Section 7).

To achieve this implementation-independence, our profiler is based on the cost semantics presented in the previous section. Though this method offers less precision than a traditional profiler, its results predict asymptotic space use for any provable

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

implementation of the language. In addition, a semantic profiler can be used to reason about the performance of scheduling policies or hardware configurations even without full implementations of these policies or access to the hardware.

As with other forms of profiling, semantic profiling gives answers specific to a particular input and, similarly, only yields results for inputs where the program terminates. For each closed program, the cost semantics is used to derive a pair of cost graphs. We have implemented our cost semantics as an interpreter in Standard ML (Milner *et al.* 1997) using a straightforward transcription of the inference rules. The cost graphs are then interpreted using the techniques described in this section, and the results are presented to the programmer along with more concrete measures of performance. We remind the reader that cost graphs, and therefore the results of the profiler, are not abstract in the sense of abstract interpretation (Cousot & Cousot 1977): they do not approximate the extensional behavior of programs. Instead, they are abstract in that they do not explicitly account for many aspects of the language implementation, including the compiler and garbage collector.

4.1 Simulation

Using cost graphs, our profiler can simulate the parallel execution of programs. This simulation 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. Below, we describe how different scheduling policies are integrated into our profiler and how information about memory use is computed and presented to the programmer.

Scheduling Policies. To include different scheduling policies in our profiler, one must simply implement a function that, given a graph and the set of all ready nodes, determines which nodes should be scheduled in the next step. Policies can usually be implemented with a few lines of code.

For example, the depth-first scheduling policy is simulated using a list to represent the set of ready nodes. (Recall that a node is ready if all its parents have been visited, but it has not yet been visited.) To simulate P processors, the first P nodes are removed from the list. The profiler must then determine the set of nodes that are enabled in this step. This is accomplished by mapping a function defined by the edges of the computation graph across these nodes. The results are then concatenated together and added at the front of the list of ready nodes. These concatenated nodes must be kept in the same order as the corresponding nodes that were removed from the ready list to ensure a depth-first traversal of the graph (Blelloch *et al.* 1999). Note that the profiler performs none of the actual computation of the program. The process of simulating a scheduling policy requires only the nodes and edges of the computation graph. The breadth-first policy uses a similar implementation, but instead of adding new nodes to the beginning of the list, it adds them at the end.

The work-stealing scheduling requires slightly more effort: it must maintain a separate set of nodes for each processor. Each set is maintained using a list similar to that used by the depth-first scheduler. If a list becomes empty, another list from which to steal is chosen randomly. Though work-stealing derives many of its benefits

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

from efficient techniques for managing these sets of tasks (e.g., using doubly ended queues as described by Frigo *et al.* 1998), these techniques have little if any effect on space use and we are free to omit them in our profiler.

We have also implemented a variant of each of these policies that simulates the more coarsely grained schedules that are used by our runtime implementation. That is, the versions described above schedule tasks at the granularity of individual nodes. Put another way, those versions simulate the behavior of threads that can be preempted at arbitrary points in the execution of the program. In our runtime implementation (as described in Section 5), scheduling is performed cooperatively by processors, and thus a thread will only be interrupted if it makes a call to the scheduling library, for example, to spawn a new task. In terms of the cost graphs, which means that whenever there is exactly one node n_2 that becomes ready as a result of processor p visiting a node n_1, then p will always visit n_2 in the next step, regardless of what nodes are enabled by other processors.

As the profiler is a sequential program, simulating scheduling policies in the profiler avoids most of the complexity associated with implementing a scheduling policy as part of the runtime: an online scheduler implemented as part of a language runtime is an example of a *concurrent* program, since its behavior depends crucially on the interleaving of memory accesses by the hardware. Such an implementation must carefully use synchronization primitives to guard any state shared among different processors. This is not necessary in our profiler. While the profiler could easily be implemented as a *parallel* program, we leave such an implementation as future work. Note, however, that the extensional semantics of such a implementation would be identical, by definition, to the one we describe in this article. We should further note that our implementation is not a hardware simulator. It is an abstraction of the actual implementation: it does not account for the precise cost of scheduling each node, and it only approximates the interleaving of tasks that would occur in the runtime implementation. In the simulation, processors move in lock-step as they visit nodes and fully "synchronize" after each step.

Space Use. Once the profiler has computed the set of nodes visited at a given step, the amount of memory required at this step can be determined by following the steps outlined in the previous section. In particular, the profiler uses the set of visited nodes to determine the set of roots (i.e., visited nodes that are the sink of a heap edge that leads from an unvisited node). The total space use is determined by the set of nodes in the heap graph that are reachable from these roots. Because the number of nodes visited in each step is small compared to the total number of nodes in the computation graph, our profiler explicitly maintains the set of edges that cross between unvisited and visited nodes. At each step, it simply adds and removes edges from that set based on which nodes are visited.

While it is possible to use this technique to show the total space use as a function of time since the beginning of program execution (as in Runciman & Wakeling 1993a), we have found it more useful to record the maximum amount of space required at any point during an execution. We refer to this quantity as the high-water mark of space use. By iterating over different inputs, this quantity may then be plotted as a function of input size to show trends in space use.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

The profiler is also able to attribute space use to variables and expressions found in the source program. To do so, it uses an extended version of the cost semantics that associates a source expression or other annotation with each node in the heap graph. For example, the node corresponding to the allocation of a sequential pair comprised of the values bound to variables x and y would be annotated with the source expression (x,y). A node representing the allocation of a function value is annotated with the list of variables that appear freely in the body of the function (excluding the function argument). These lists are written within square brackets (e.g., [a,b,c]). For uses of values (e.g., in the applications of array primitives such as append) nodes are annotated with the source code of the application. Note that this means that the annotations of some nodes may be substrings of the annotations of other nodes. For example, there may be a node with an annotation such as #1 lr as well as one annotated append (#1 lr, #2 lr). While it is possible that this could lead to very long annotations, this has not been a problem in our experience. In general, ML programmers tend to name intermediate results (as is good programming practice), and this limits the length of annotations. In addition, if we only consider the annotations of nodes at the sources or sinks of root edges (as we do in the subsequent development), there will be little duplication of source code in the annotations in which we are interested.

These annotations support profiling both the creation of values and the uses of those values. If we are interested in what sort of values are present in the heap, then we look at the annotations associated with the sinks of heap edges. As noted in Section 3, the sink of a heap edge always represents a value. When showing space use (or "retainers" in the terminology of Runciman & Röjemo 1996), we instead consider the sources of heap edges. These correspond both to values as well as uses of those values (e.g., function applications, pair projections).

In this article, we focus on the uses of values (or retainers). Using annotations, our profiler is able to break down the total space use by attributing part of this total to each root. As described previously, the total space use is proportional to the total number of heap nodes reachable from the roots. For nodes that are reachable from more than one root, the space associated with that node is divided equally among all such roots. For example, if there are six nodes in the heap graph that are each reachable from three roots, then each root would be charged two units of space.

4.2 Visualization

The cost graphs given by our semantics are often quite large and thus difficult for programmers to comprehend. We have implemented a method that distills these graphs and yields their essential components. As our 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 using generic graph layout

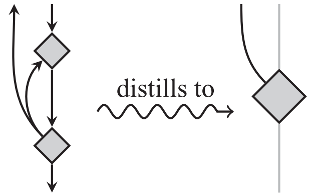
https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

software.[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 n has out-degree one, n has in-degree one, and 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 have 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.
4. Constrain the layout so that the horizontal position of nodes reflects the sequential schedule: nodes that are scheduled earlier in the sequential schedule should appear further to the left.

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. Nodes are also drawn so that the sequential schedule follows a left-to-right traversal. Finally, 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 nonunique 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

[1] We used Graphviz (`http://www.graphviz.org/`) to generate the graphs shown in this article.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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 space use. We use the color of both nodes and heap edges to highlight the behavior of the scheduling policy at step N_i, the moment when space use reaches it high-water mark.

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

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.

7. Avoid coalescing two nodes if one has been executed at or before step N_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.

8. Weight each heap edge according to its share of the total amount of heap space reachable from that edge at step N_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 as follows.

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

The sizes of graphs for several example applications are shown in Table 1, both before and after coalescing nodes and edges. These applications are described in Section 6.

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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Table 1. *Graph sizes before and after distillation*

		Original		Distilled	
Application	Input size	Nodes	Heap edges	Nodes	Heap edges
Matrix multiplication	4	4,028	1,371	314	168
”	6	12,690	4,239	1,074	534
”	8	29,032	9,579	2,554	1,212
Quicksort	16	9,720	6,773	79	76
”	24	21,496	15,341	119	116
”	32	37,880	27,365	159	156
Quickhull	16	15,259	8,788	459	618
”	24	29,343	17,264	859	1,218
”	32	40,327	23,912	1,219	1,794
n-queens	4	16,985	7,959	79	91
”	5	82,377	38,252	264	402
”	6	397,873	175,821	759	1051

code can be written in a high-level language, we implemented as much of 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 & Blelloch 2001) could be carried over in a straightforward manner.

Our runtime supports an additional runtime parameter that indicates how many processors to use.[2] For each processor, the runtime sets up the local processor

[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.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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

  (* the first argument is the identifier of the current processor *)
  (* 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
```

Fig. 9. Signature for scheduling policies. Scheduling policies are defined by implementing this signature.

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 scheduling loop. The loop is run in parallel by each processor and consists of running a single task. It is the role of the scheduling policy to determine which task is run in each iteration of the loop. To plug-and-play with different scheduling policies, we developed a simple signature that any policy must implement (Figure 9).

Given the purpose of scheduling policies, the functions add and get should be self-explanatory. With respect to add, "priority" means the order in which tasks would have been evaluated by a sequential implementation. Each scheduling policy chooses whether or not to respect this priority when returning tasks from the get function: a policy's notion of priority may differ depending on how that policy organizes tasks.

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 nontrivial thread operations in cases where they are unnecessary. This function returns a boolean value that indicates whether or not there is any other ready task that has a higher priority (as assigned by the scheduling policy) than the task being evaluated by the current processor. If the result is true, then the current task should be suspended and enqueued. Otherwise, the current task can continue. The shouldYield 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 parallel implementations of other languages.

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. These policies are all greedy, in that processors will not be kept idle if there are available tasks (as discussed in Section 3). 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.

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 performs a left-to-right, breadth-first traversal of the computation graph and is equivalent to a round-robin scheduler. It is the "fairest" of the three schedulers we implemented in the following sense: if we consider each chain of nodes in the computation graph as a thread, it alternates scheduling each of these threads among the available processors.

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: to ensure that our implementation obeys the global left-to-right depth-first priority, the children of the leftmost task must be given higher priority 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.

We also experimented with a scheduling policy without this complication (where the global queue is maintained as a LIFO stack). However, as noted above, this policy is not faithful to a depth-first scheduling policy. There is little theoretical work addressing this form of scheduling, and we have not considered it in our experimental results in Section 6.

Work-Stealing Scheduling. A work-stealing scheduling policy (Burton & Sleep 1981; Blumofe & 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. This leads to less contention and significantly smaller overhead compared to the breadth- and depth-first schedulers, both of which use a single global queue in our implementation.

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.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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. 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, vector, 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

One method to measure space use is to record the maximum amount of live data found in the heap at the end of any garbage collection. Given the default behavior of most collectors, however, there is no way to understand the accuracy of this measurement. 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.

To rectify this problem, we have modified MLton's garbage collector to measure memory use with bounded error and relatively small effects on performance. To measure the high-water mark within a fraction E of the true value, we restrict the amount of memory available for new allocation as follows. At the end of each collection, given a high-water mark of M bytes and L bytes of live data currently in the heap, we restrict allocation so that no more than $M * (1 + E) - L$ 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 $M * (1 + E)$ bytes. That is, consider the worst-case scenario, where $M * (1 + E)$ bytes of data was live at some point between collections. No more data could be live because there was L bytes live after the previous collection and only $M * (1 + E) - L$ additional bytes were allocated since then. Regardless of how many bytes are live at the next collection, the runtime will always report a high-water mark of at least M, because the high-water mark is a monotonically increasing value. (It will report more than M if more than M bytes of live data are discovered by the collector.) We will, therefore, achieve the desired level of accuracy. This technique differs from most collector implementations, which

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

use only the current amount of live data L in determining when to perform the next collection.

For example, suppose that during the execution of a program, we have already observed 10 MB of live data in the heap but at the present moment (just after completing a collection), there is only 5 MB of live data. If we are satisfied with knowing the high-water mark to within 20% of the true value, then we constrain the collector to allocate at most $10 * (1 + 0.2) - 5 = 7$ MB before performing another collection. Even if all of this newly allocated data were to be live at some point in time, the high-water mark would never exceed 12 MB. At the end of the second collection, the runtime will report a high-water mark no less than 10 MB.

The two key properties of this technique are, first, that it slowly increases the threshold of available memory (yielding an accurate result) and second, that it dynamically sets the period between collections (as measured in bytes allocated) to avoid performing collections when doing so would not give any new information about the high-water mark. Continuing the example above, if a second collection found only 1 MB of live data, then there is no reason to perform another collection until at least 11 MB of new data have been allocated.

Note that this technique may also be combined with a generational garbage collector (Lieberman & Hewitt 1983; Ungar 1984), for example, as in MLton. In this case, limiting allocation simply means limiting the size of the nursery. As in an Appel-style generational collector (Appel 1989), all of the remaining space may be assigned to the older generation. Thus, this technique need not increase the frequency of major collections. When reporting the high-water mark after a minor collection, the collector must assume that all of the data in the older generation is live.

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

6 Empirical results

Predictions about program performance based on a high-level semantics are only useful if those predictions are also realized by an efficient implementation. We validate our profiling tools empirically using an instrumented version of our MLton implementation. Because our profiling tools do not measure exact memory usage, we can only confirm that these predictions match the asymptotic behavior of our implementation on equivalent inputs: where the semantics predicts space use as a constant, linear, or super linear function of the input size, our MLton implementation delivers similar 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 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.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Table 2. *Lines of code in applications. This table shows the size of each of the applications described in the section. This does not include library routines such as the implementations of parallel vectors or the collection of matrix operations used in Barnes–Hut simulation*

Name	Lines of code
Matrix multiplication	165
Quicksort	29
Mergesort	132
Selection sort	41
Insertion sort	13
Quickhull	68
n-queens	40
Barnes–Hut	379

We implemented several small parallel applications as part of our study. These include the matrix multiplication example described above, several sorting algorithms, the Quickhull algorithm for finding the convex hull of a set of points (Preparata & Shamos 1988), the *n*-queens problem, and the Barnes–Hut simulation of gravitational bodies (Barnes & Hut 1986). Table 2 provides a list of these applications along with their sizes. Note that this includes a more sophisticated version of matrix multiplication, discussed in more detail in Section 6.2 below.

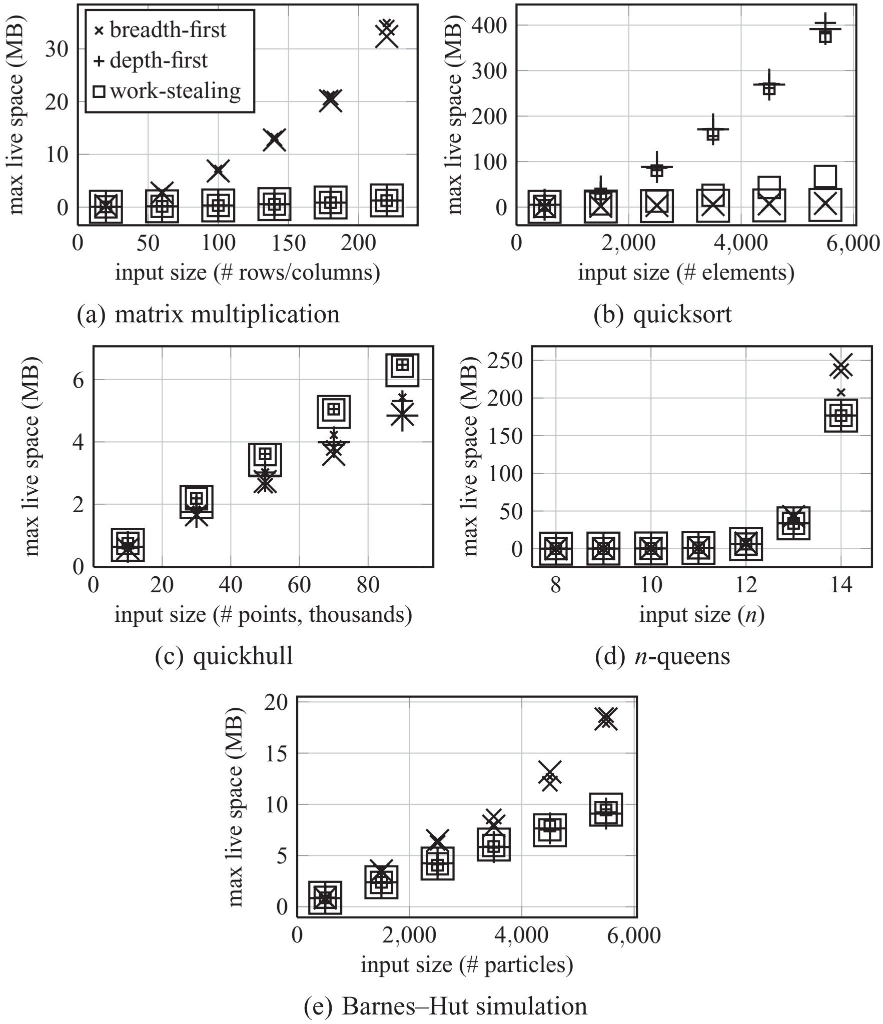
6.1 Space use

For each application, we report the effect of scheduling policy and number of processors on the amount of memory required by 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 10 shows the high-water mark of space use for five of the applications in our study. Smaller values indicate better performance. We use different shapes to represent different policies: × for breadth first, + for depth-first, and □ for work-stealing. Larger symbols indicate more processors were made available; we show results for space use on one, two, and four processors.

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 10(a).

Sorting. We implemented several sorting algorithms including quicksort, mergesort, insertion sort, and selection sort. Figure 10(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. In each recursive call, the first element is taken as the pivot element. 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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press



(a) matrix multiplication

(b) quicksort

(c) quickhull

(d) *n*-queens

(e) Barnes–Hut simulation

Fig. 10. Space use versus input size. Each plot shows the high-water mark of space use for one of five applications. We tested three scheduling policies (depth-first, breadth-first, and work-stealing) 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.

quadratic space. This behavior is also predicted by the cost semantics. The plot in Figure 11 is generated by our semantic profiler and shows the consumers of space for the depth-first policy at the high-water mark. The shape of this plot agrees with the data in Figure 10(b). The plot also shows that this space is referenced by various parts of the filter function. In particular, most of the live heap data is referenced by the variable xs that appears in the expressions tail xs, head xs, isnil xs, and filter xs as well as the closure [xs,filter,f].

The cost graphs in Figure 12 show 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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

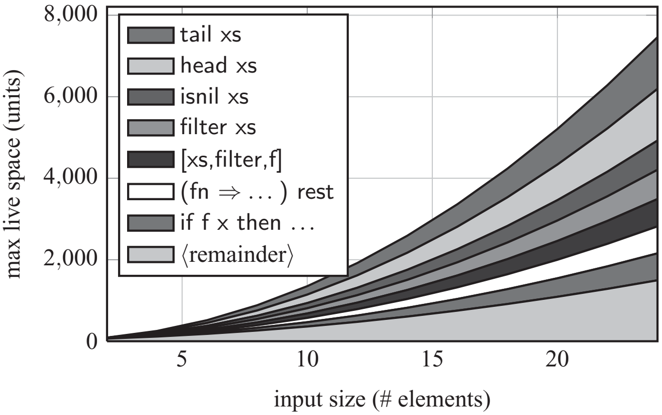


Fig. 11. Simulated space use for quicksort. This plot shows space use, as predicted by our profiler, as a function of input size when using a depth-first scheduling policy and a single processor. Overall, this plot agrees with the measurements shown in Figure 10(b). The legend shows how space use can be attributed to subexpressions of the program.

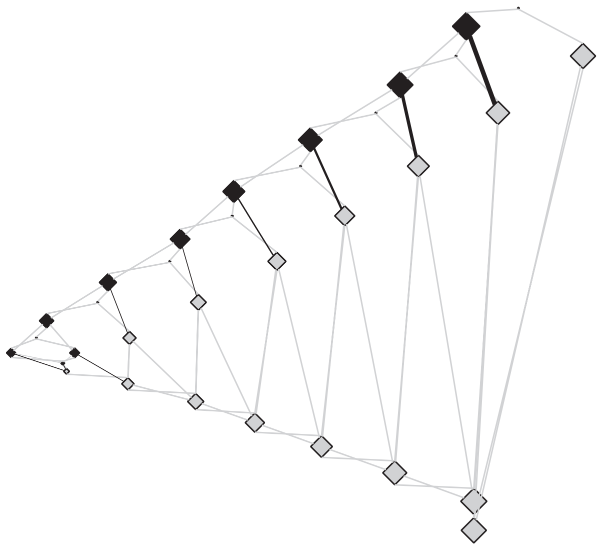


Fig. 12. Cost graphs for quicksort. These summarized graphs show the point at which the program reaches the high-water mark of space use under a depth-first scheduling policy: after the first completion of a recursive invocation with a nonempty input.

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. (This name was first used by Preparata & Shamos (1988).) We

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

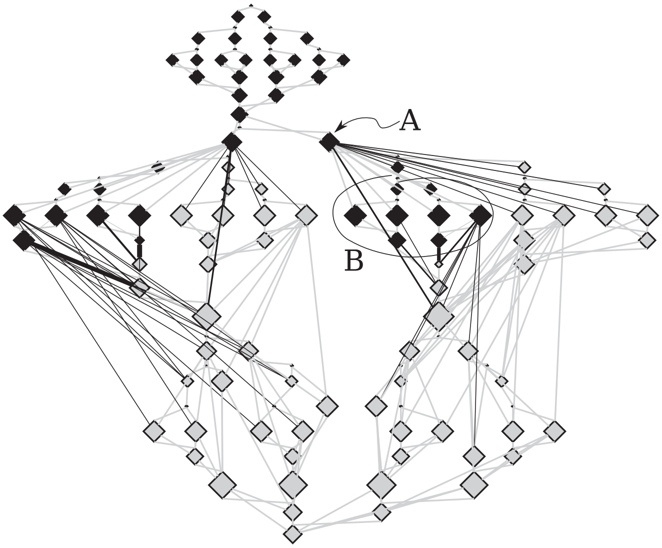


Fig. 13. Cost graphs for quickhull. Quickhull displays a more irregular form of parallelism. Heap edges show how data are split between different parallel branches. Portions of the graph labeled “A” and “B” are discussed in the text.

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 10(c) shows the high-water mark of space use, which again matches our cost semantics-based predictions (not shown).

Quickhull offers more parallelism than the previous example, but this parallelism is still more constrained than that found 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 graphs shown in Figure 13.

Nodes are colored to illustrate one point in the execution of the work-stealing scheduling policy with two processors. 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-**Queens.** The *n*-queens problem asks for a placement of *n* tokens on an *n*-by-*n* grid such that no two tokens appear in the same row or column, or on the same diagonal. In the terminology of the game of chess, this would be a placement of *n* queens on the board such that no queen could attack any other queen. For a given *n* there may be zero or more solutions; our implementation finds all solutions for each board

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

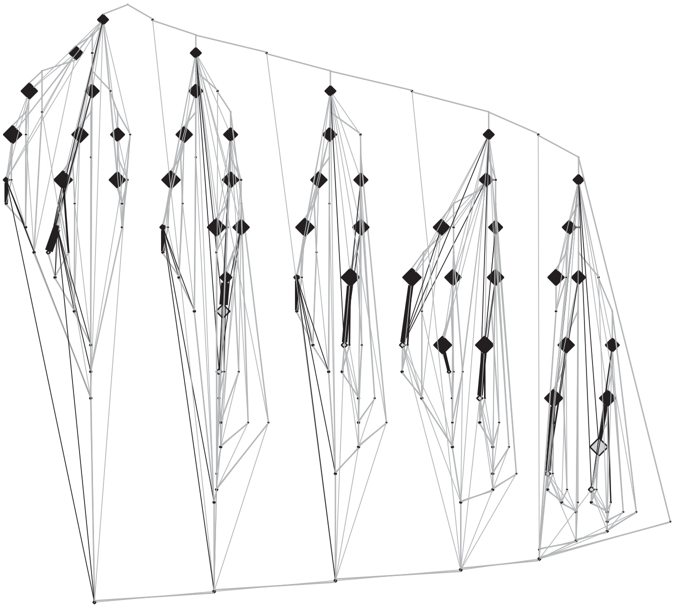


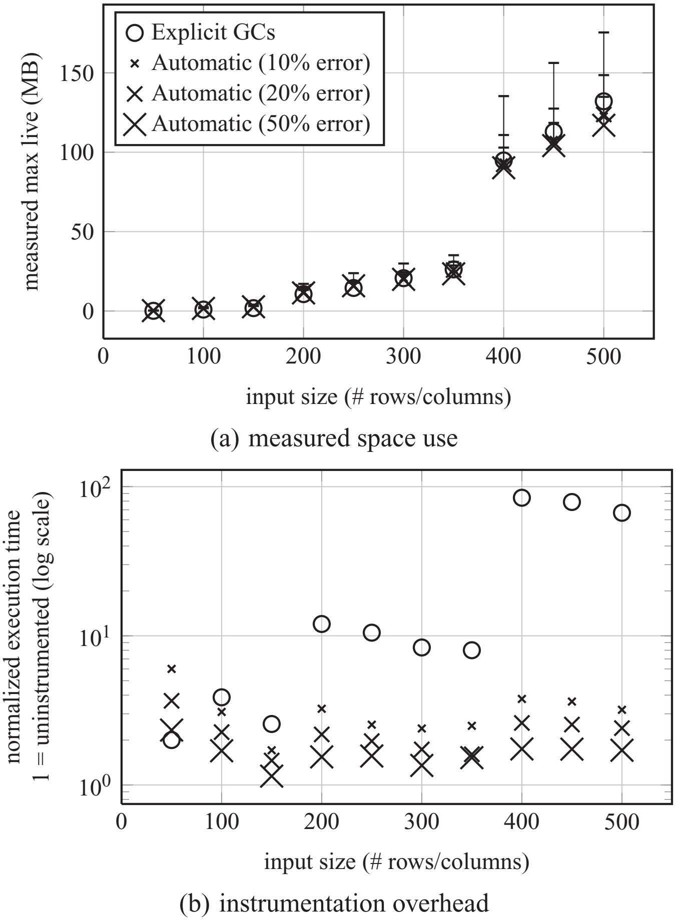
Fig. 14. Cost graphs for n-queens. This figure shows the graphs for $n = 5$ and the breadth-first scheduler. This scheduler requires the most space when it is maintaining a large set of partial placements.

size. This problem is representative of a wide class of constraint problems that can be solved by some form of search. In this example, these constraints take the form of partial placements of less than n pieces. We represent these placements as lists of integer pairs, one element for each queen denoting the row and column where it is placed.

Figure 14 shows the cost graphs for the instance of this problem with $n = 5$. Each branch of the search after the initial placement can be computed independently, and these branches are arranged roughly as five columns of nodes. The graphs are highlighted to show the point at which the breadth-first scheduler reaches its high-water mark for space use. The breadth-first scheduler performs relatively poorly in this application because it explores too many branches in parallel and must maintain the sets of constraints for each of these branches. The profiler predicts that the space use for the breadth-first scheduler will grow faster than either of the other two scheduling policies. This is also reflected in our MLton implementation, as shown by the last two data points in Figure 10(d).

Barnes-Hut Simulation. Figure 10(e) 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.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press



(a) measured space use

(b) instrumentation overhead

Fig. 15. Profiling space use. Four different space measurements (a) and the cost of obtaining these measurements (b). Execution times are normalized to an uninstrumented version and shown on a logarithmic scale.

Due to its complexity (e.g., use of modules, pattern matching), we have not implemented a version of this application that is compatible with our profiling tools. Applying our methodology informally, however, we expect the program to generate very wide cost graphs. Like the matrix multiplication example, the breadth-first scheduling policy 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 15(a), we show space use for

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Table 3. *Profiling space use. These data are presented graphically in Figure 15. Measured space use is the amount of space reported by each method. A bound on the actual space use can be derived by adding the appropriate percentage. Time overhead is reported as the ratio between the instrumented and uninstrumented execution times. Averages are reported for the time overhead of restricted allocation measurements.*

	Measured space (MB)				Time overhead			
		Automatic (% error)			Explicit GCs	Automatic (% error)		
Input size	Explicit GCs	10%	20%	50%		10%	20%	50%
50	0.12	0.24	0.24	0.24	2.00	6.00	3.67	2.33
100	0.86	1.54	1.54	1.55	3.87	3.09	2.26	1.70
150	1.73	2.69	2.65	2.55	2.56	1.71	1.46	1.15
200	10.16	11.49	11.51	10.87	12.00	3.24	2.18	1.54
250	13.85	15.38	14.94	15.16	10.50	2.54	1.97	1.56
300	19.73	19.88	19.28	19.02	8.35	2.39	1.73	1.35
350	25.00	24.89	24.61	22.34	8.00	2.49	1.61	1.52
400	90.43	89.22	88.09	86.04	84.27	3.77	2.60	1.74
450	107.64	102.72	101.31	99.33	78.92	3.62	2.54	1.74
500	125.96	116.93	118.05	111.54	66.79	3.19	2.40	1.71
average						3.20	2.24	1.63

blocked matrix multiplication measured four different ways. Blocked multiplication is a version of multiplication where the input matrices are partitioned and the products of the partitions are computed before summing them to yield the final result. It is often chosen because it offers much better cache behavior. We found it to be an example that was difficult to instrument in a way that was accurate but not prohibitively slow.

All data shown in this figure use the breadth-first scheduler and one processor. The first series ○ 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 bounds 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. The gap between inputs of sizes of 150 and 200 as well as the one between 350 and 400 are due to an additional level of recursion in the algorithm. That is, the algorithm has a fixed maximum size at which submatrices are multiplied directly rather than by partitioning them, and input sizes such as 200 and 400 require an additional partition. Table 3 shows the data used in these plots. The space use measurements for the restricted allocation technique are reported values; the true values may be 10%, 20% or 50% higher. (For an input size of 500, the 50% datum indicates that as much as much as 167.31 MB were used.)

We take the measurements derived from the explicitly added garbage collections to be the most accurate measurement of memory use. In each case, this technique reported the greatest values. The data show that the restricted allocation measurements

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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 15(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. By “uninstrumented,” we mean a version of the algorithm without any explicit calls to the garbage collector or additional restrictions on the amount of allocation between collections; that is, the allocator and garbage collector are the unmodified MLton versions. As in the case of the space measurements themselves, the time required to take these measurements increases significantly at input sizes of 200 and 400, when the size of the input requires an additional level of recursion. In Table 3, we also report averages over the range of input sizes for the restricted allocation technique. On average, the ratio between the time required to measure space use with a 10% (20%, 50%) bound and the time required by the uninstrumented version is 3.20 (2.24, 1.63, respectively).

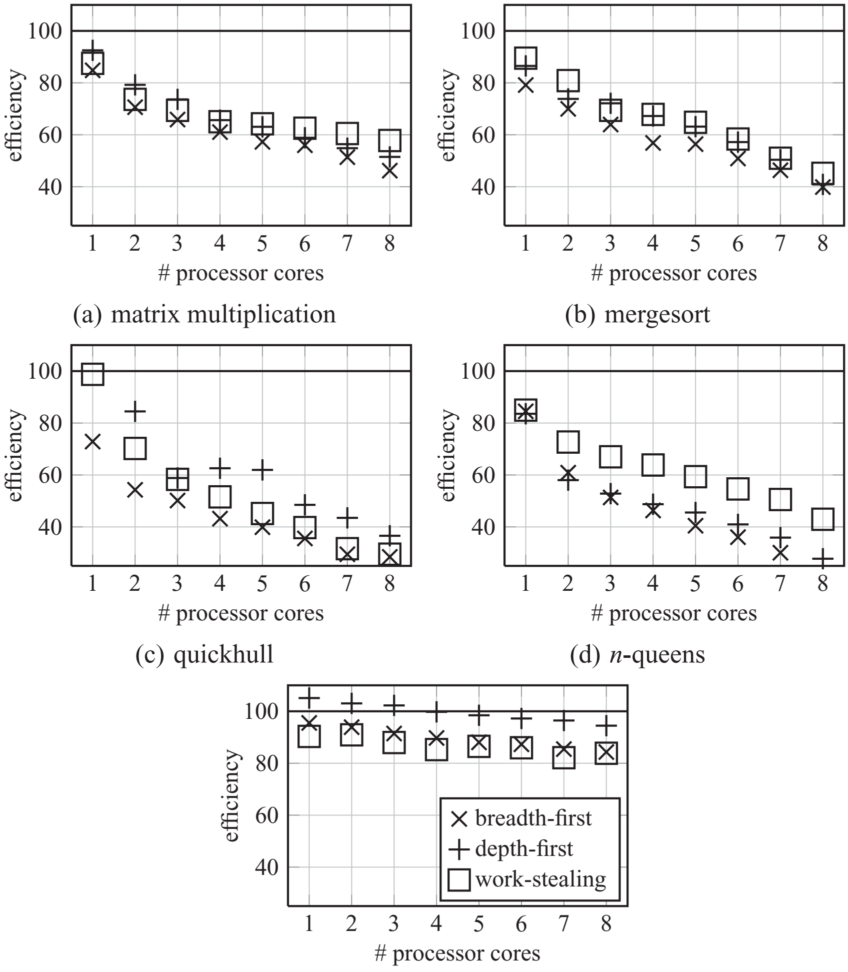
6.3 Parallel efficiency

As the purpose of parallelism is to improve performance, we also report parallel efficiency. Efficiency is a measure of how well an implementation scales up as more processors are added. 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.

We depart from common practice by plotting efficiency rather than execution time or speed-up. While execution time is easy to interpret, it is difficult to use plots of execution time to ascertain trends in data or to make comparisons between different applications or scheduling policies. Plots showing speed-up ameliorate these problems by normalizing parallel execution time using the sequential execution time. However, it is still difficult to understand the effect on performance when using a small number of processors (including the overhead for a single processor) as speed-up devotes little space to this portion of the plot. In addition, comparing trends in the performance requires the reader to compare the relative curvature of different data sets. Finally, efficiency is a direct indication of the cost (or overhead) of adding additional processors and makes it more clear how this cost might be related to the number of processors.

Parallel efficiency is defined as $100 \cdot T_1/(T_P \cdot P)$, where T_1 is the sequential execution time and T_P is the execution time on P processors. Equivalently, efficiency is equal to the speed-up divided by the number of processors. Efficiency can be interpreted as the percentage of processor cycles that are being used to run the application rather than performing some form of communication or synchronization. Thus 100% efficiency corresponds to a linear speed-up. As an example, if a scheduling policy is 80% efficient with two processors, then it will achieve a speed-up of 1.6. If efficiency falls to 70% with three processors, then it will only achieve a speed-up of

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press



(a) matrix multiplication

(b) mergesort

(c) quickhull

(d) *n*-queens

(e) Barnes-Hut simulation

Fig. 16. Parallel efficiency. As a measure of scalability, we plot efficiency as a function of the number of processors. As discussed in the text, efficiency is defined as $100 \cdot T_1/(T_P \cdot P)$ where T_1 is the sequential execution time and T_P is the execution time on P processors. 100% efficiency is equivalent to a linear speed-up.

2.1. Intuitively, efficiency tells us how much more computational throughput we get for each additional processor.

Though we might hope for applications and scheduling policies that are 100% efficient, we expect some overhead from parallel execution. Practically speaking, we are looking for applications and scheduling policies where efficiency is independent of the number of processors. This indicates that adding additional processors will continue to improve performance at the same rate. In plots such as those shown below, efficiency that is independent of the number of processors will appear as a horizontal line.

Figure 16 shows parallel efficiency for one to eight processors for five applications. In these plots, we do not include the cost of garbage collection. As we argued above,

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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.

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. In each case the sequential execution time T_1 is the execution time on a single processor without any of the support required for parallel execution. In our implementation, support for parallel execution is controlled using a compile-time constant. Though we performed some experiments with variations in task granularity, we report on only a single granularity for each example. In each case, this granularity was chosen to limit the overhead when run on a single processor to an acceptable amount (typically no more than 10%–20%).

Figure 16(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 on a circle. Part (d) shows the overhead for the n-queens problem with $n = 14$ and where no parallelism is used for subproblems of size 6 or less. Part (e) shows the overhead for the Barnes–Hut simulation with points distributed uniformly randomly. Though all of these applications show some potential for faster execution, some scale better than others. In addition, there are some clear differences in the performance trends for different scheduling policies. For example, in the matrix multiplication example, the initial cost of adding parallelism is greater for the work-stealing scheduler when compared to the depth-first scheduler (leading to lower efficiency with one processor). This cost levels off as more processors are added: work stealing is more efficient for five or more processors. In this case, the efficiency of the work-stealing scheduler is less dependent on the number of processors. In Part (c), the point for the depth-first schedule with one processor is not shown due to the ranges we selected for the plots; this point lies at 113% efficiency. That is, the depth-first scheduler with one processor is 13% faster than the sequential version of this algorithm. We attribute this to the fact the scheduler is fixed at compile-time and that MLton is a whole-program compiler: we believe this choice of scheduler causes a small perturbation of the code in the inner loop of the algorithm that results in faster execution. In the cases of mergesort and quickhull, there does not seem to be enough parallelism to efficiently leverage all of the processors available on this hardware. In all cases, however, using more processors leads to lower efficiency per processor, most likely because of contention on the memory bus. In contrast, the n-queens problem provides many opportunities for parallel execution: given a placement of pieces in the first k rows, each valid placement in row $k + 1$ is considered in parallel. In this case, the more efficient data structures of the work-stealing scheduler (e.g. per-processor queues) lead to better use of parallel resources. The final point for the breadth-first scheduler does not appear within the selected range; with eight processors, the breadth-first scheduler is only 23% efficient. Finally, the Barnes-Hut simulation scales the best of any of our applications. It performs significantly more sequential computation than the other examples relative to the size of the data structures it builds. As in the case of the Quickhull algorithm, the depth-first scheduler achieves an efficiency greater

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

than 100% (or equivalently a super-linear speed-up for up to three processors); we believe this is again due to the whole-program optimizations performed by the compiler.

7 Discussion

7.1 *Alternative rules*

There are a number of design choices latent in the inference rules given in Figure 4. 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 4 is highlighted with a rectangle.

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

This rule yields the same extensional result as the version given in Figure 4, 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. 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 & 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 validating our work, we discovered one example where we were required to change the language 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 of MLton. The fix requires this optimization to consider any record that contains a field with a recursive type to be of potentially unbounded size. With this change, the example that previously required quadratic space could be run using only linear space (as a function of its input size).

As flattening and other program optimizations affect space use, it is critical that the language provides a specification of performance at the level of the source code. This specification can then be used to guide programmers expectations about performance and validate compiler optimizations.

7.3 Nested parallelism

Another form of "flattening" appears in implementations of nested parallelism (e.g., Blelloch & Sabot 1990; Blelloch *et al.* 1994; Chakravarty & 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 simultaneously exposing all available parallelism.

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: flattening can be described as a constrained form of breadth-first scheduling. This characterization agrees with our experience with the flattening implementation of NESL (Blelloch *et al.* 1994): 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.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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 & Keller 2000; Peyton Jones *et al.* 2008). 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 & 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 & 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 & Peyton Jones (1992, 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 & Peyton Jones 1992; Runciman & Wakeling 1993b; Hammond *et al.* 1995; Charles & 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 space use.

Scheduling. Scheduling policies and their effects on space use have been studied extensively in the algorithms community (e.g., Blelloch *et al.* 1999; Blumofe & Leiserson 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 as 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.* (2008) make scheduling policies explicit using an intermediate language and support nested scheduling policies. It would be interesting to see how our cost semantics could serve as a specification for such policies.

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 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 & 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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

policy did not usually affect the running time of programs, with the exception of 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. Language semantics extended with some notion of cost have been used to reason about the time complexity of sequential functional programs (Sands 1990; Rosendahl 1989; Roe 1991). Sansom & Peyton Jones (1995) used a cost semantics in the design of their profiler. However, their profiling results are derived from an instrumented version of their compiler and runtime system, not from the semantics itself.

A cost semantics similar to the one used in the current work was introduced by Greiner & Blelloch (1999). 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 & Sands (1999) give a semantic definition of what it means for a transformation to be "safe-for-space" (Shao & Appel 1994). They provide several laws to help prove that a given transformation does not asymptotically increase the space use 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 & 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 parallel schedules 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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

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 a part of the scheduling policy.

We invite readers to download and experiment with our prototype implementation, available from the first author's website[3] or the `shared-heap-multicore` branch of the MLton repository.

Acknowledgments

This work was funded in part by an IBM OCR gift, the Center for Computational Thinking sponsored by Microsoft, and a gift from Intel. We would like to thank the MLton developers for their support and advice. We would also like to thank Simon Peyton Jones, the ICFP reviewers, and the referees for their helpful comments.

References

Aditya, S., Arvind, Jan-Willem, M. & Augustsson, L. (June 1995) *Semantics of pH: A Parallel Dialect of Haskell*. Tech. Rep. Computation Structures Group Memo 377-1. MIT.

Appel, A. W. (1989) Simple generational garbage collection and fast allocation, *Softw. Prac. Exp.*, 19 (2): 171–183.

Appel, A. W., Duba, B. & MacQueen, D. B. (Nov. 1988) *Profiling in the Presence of Optimization and Garbage Collection*. Tech. Rep. CS-TR-197-88. Princeton University.

Arvind, N., Rishiyur, S. & Pingali, K. K. (1989) I-structures: Data structures for parallel computing, *ACM Trans. Program. Lang. Syst.*, 11 (4): 598–632.

Barnes, J. & Hut, P. (1986) A hierarchical O(N log N) force-calculation algorithm, *Nature*, 324 (4): 446–449.

Blelloch, G. E., Hardwick, J. C., Sipelstein, J., Zagha, M. & Chatterjee, S. (1994) Implementation of a portable nested data-parallel language, *J. Parallel Distrib. Comput.*, 21 (1): 4–14.

Blelloch, G. E., Gibbons, P. B. & Matias, Y. (1999) Provably efficient scheduling for languages with fine-grained parallelism, *J. ACM*, 46 (2): 281–321.

Blelloch, G. & Greiner, J. (1995) Parallelism in sequential functional languages. In *Proceedings of the International Conferance on Functional Programming Language and Computer Architecture*. ACM, pp. 226–237.

Blelloch, G. E. & Sabot, G. W. (1990) Compiling collection-oriented languages onto massively parallel computers, *J. Parallel Distrib. Comput.*, 8 (2): 119–134.

Blumofe, R. D. & Leiserson, C. E. (1998) Space-efficient scheduling of multithreaded computations, *SIAM J. Comput.*, 27 (1): 202–229.

Blumofe, R. D. & Leiserson, C. E. (1999) Scheduling multithreaded computations by work stealing. *J. ACM*, 46 (5): 720–748.

Burton, F. W. & Sleep, M. R. (1981) Executing functional programs on a virtual tree of processors. In *Proceedings of the Conference on Functional Program Language and Computer Architecture*. ACM, pp. 187–194.

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

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Chakravarty, M. M. T. & Keller, G. (2000) More types for nested data parallel programming. In *Proceeding of the International Conference on Functional Programming*. ACM, pp. 94–105.

Charles, N. & Runciman, C. (1998) An interactive approach to profiling parallel functional programs. In *Selected Papers of the Workshop on the Implementation of Functional Language*. Springer, pp. 20–37.

Cheng, P. & Blelloch, G. E. (2001) A parallel, real-time garbage collector, *Sigplan Not.*, 36 (5): 125–136.

Conchon, S. & Fessant, F. L. (1999) Jocaml: Mobile agents for Objective-Caml. In *Proceedings of the International Symposium on Agent Systems and Applications*. IEEE, pp. 22–29.

Cousot, P. & Cousot, R. (1977) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Language*. ACM, pp. 238–252.

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

Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007) Status report: The Manticore project. In *Proceedings of the Workshop on ML*. ACM, pp. 15–24.

Fluet, M., Rainey, M. & Reppy, J. (2008) A scheduling framework for general-purpose parallel languages. In *Proceedings of the International Conference on Functional Programming*. ACM, pp. 241–252.

Frigo, M., Leiserson, C. E. & Randall, K. H. (1998) The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conferance on Programming Language, Design and Implementation*. ACM, pp. 212–223.

Greiner, J. & Blelloch, G. E. (1999) A provably time-efficient parallel implementation of full speculation, *ACM Trans. Program. Lang. Syst.*, 21 (2): 240–285.

Gustavsson, J. & Sands, D. (1999) A foundation for space-safe transformations of call-by-need programs. In *Proceedings of Workshop on Higher Order Operational Techniques in Semantics*. Elsevier, pp. 69–86.

Hammond, K., Berthold, J. & Loogen, R. (2003) Automatic skeletons in template Haskell, *Parallel Process. Lett.*, 13 (3): 413–424.

Hammond, K., Loidl, H.-W. & Partridge, A. S. (1995) Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In *Proceedings of Conferance on High Performance Functional Computing*. Denver, CO, pp. 208–221.

Hammond, K. & Peyton Jones, S. L. (Sept. 1992) Profiling scheduling strategies on the GRIP multiprocessor. In *Proceedings of the International Workshop on the Parallel Implementation of Functional Language*. RWTH Aachen, Germany, pp. 73–98.

Jay, C. B., Cole, M., Sekanina, M. & Steckler, P. (1997) A monadic calculus for parallel costing of a functional language of arrays. In *Proceedings of the International Euro-Par Conferance on Parallel Processing*. Springer-Verlag, pp. 650–661.

Lieberman, H. & Hewitt, C. (1983) A real-time garbage collector based on the lifetimes of objects, *Commun. ACM*, 26 (6): 419–429.

Loidl, H.-W. & Hammond, K. (1996) A sized time system for a parallel functional language. In *Proceedings of the Glasgow Workshop on Functional Programming*. Department of Computing Science, University of Glasgow.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (Revised)*. MIT Press.

Peyton J., Simon, L., Roman, K., Gabriele, K. & Chakravarty, M. M. T. (2008) Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the IARCS Annual Conferance on Foundations of Software Technology and Theoretical Computer Science*. Leibniz International Proceedings in Informatics.

Preparata, F. P. & Shamos, M. I. (1988) *Computational Geometry – An Introduction*, 2nd ed. Springer-Verlag.

Reppy, J. H. (1999) *Concurrent Programming in ML*. Cambridge University Press.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press

Roe, P. (1991) *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow.

Röjemo, N. & Runciman, C. (1996) Lag, drag, void and use–heap profiling and space-efficient compilation revisited, *Sigplan Not.*, 31 (6): 34–41.

Rosendahl, M. (1989) Automatic complexity analysis. In *Proceedings of the International Conferance on Functional Programming Language and Computere Architecture*. ACM, pp. 144–156.

Runciman, C. & Röjemo, N. (1996) New dimensions in heap profiling, *J. Funct. Program.*, 6 (4): 587–620.

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

Runciman, C. & Wakeling, D. (1993b) Profiling parallel functional computations (without parallel machines). In *Proceedings of the Glasgow Workshop on Functional Programming*. Springer-Verlag, pp. 236–251.

Sands, D. (Sept. 1990) *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London.

Sansom, P. M. & Peyton Jones, S. L. (1992) Profiling lazy functional programs. In *Proceedings of the Glasgow Workshop on Functional Programming*. Springer-Verlag, pp. 227–239.

Sansom, P. M. & Peyton Jones, S. L. (1995) Time and space profiling for non-strict, higher-order functional languages. In *Proceedings of the Symposium on Principles of Program. Languange*. ACM, pp. 355–366.

Shao, Z. & Appel, A. W. (1994) Space-efficient closure representations. In *Proceedings of the Conferance on LISP and Functional Programming*. ACM, pp. 150–161.

Trinder, P. W., Hammond, K., Loidl, H.-W. & Peyton Jones, S. L. (1998) Algorithm + Strategy = Parallelism, *J. Funct. Program.*, 8 (1): 23–60.

Ungar, D. (1984) Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*. ACM, pp. 157–167.

Weeks, S. (2006) Whole-program compilation in MLton. In *Proceedings of the Workshop on ML*. ACM, p. 1.

https://doi.org/10.1017/S0956796810000146 Published online by Cambridge University Press