Cache and I/O Efficient Functional Algorithms

Guy E. Blelloch Robert Harper
Carnegie Mellon University
guyb@cs.cmu.edu rwh@cs.cmu.edu

Abstract

The widely studied I/O and ideal-cache models were developed to account for the large difference in costs to access memory at different levels of the memory hierarchy. Both models are based on a two level memory hierarchy with a fixed size primary memory (cache) of size M, an unbounded secondary memory organized in blocks of size B. The cost measure is based purely on the number of block transfers between the primary and secondary memory. All other operations are free. Many algorithms have been analyzed in these models and indeed these models predict the relative performance of algorithms much more accurately than the standard RAM model. The models, however, require specifying algorithms at a very low level requiring the user to carefully lay out their data in arrays in memory and manage their own memory allocation.

In this paper we present a cost model for analyzing the memory efficiency of algorithms expressed in a simple functional language. We show how some algorithms written in standard forms using just lists and trees (no arrays) and requiring no explicit memory layout or memory management are efficient in the model. We then describe an implementation of the language and show provable bounds for mapping the cost in our model to the cost in the ideal-cache model. These bound imply that purely functional programs based on lists and trees with no special attention to any details of memory layout can be as asymptotically as efficient as the carefully designed imperative I/O efficient algorithms. For example we describe an $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cost sorting algorithm, which is optimal in the ideal cache and I/O models.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Tradeoffs and Complexity Measures; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Algorithms, Design, Languages, Performance, Theory.

Keywords cost semantics, I/O algorithms

1. Introduction

On today's computers there is a vast difference in cost for accessing different levels of the memory hierarchy, whether it be registers, one of many levels of cache, the main memory, or a disk. On current processors, for example, there is over a factor of a hundred between the time to access a register and main memory, and another factor of a hundred or so between main memory and disk, even a solid state disk (SSD). This variance in costs is contrary to the standard Random Access Machine (RAM) model, which assumes that the cost of accessing memory is uniform. To account for non uniformity several cost models have been developed that assign difference costs to different levels of the memory hierarchy. The widely used I/O [2] and ideal-cache [9] models both assume a two level memory hierarchy with a fixed size primary memory (cache) of size M, an unbounded secondary memory partitioned into blocks of size B. Cost is measured in terms of the number of block transfers between primary and secondary memory—all other operations are considered free. The parameters M and B are considered variables for the sake of analysis and therefore show up in asymptotic bounds.

Algorithms that do well in these models are often referred to as I/O efficient or cache efficient—in this paper we will generically use the term cache efficient. The theory of cache efficient algorithms is now well developed (see e.g. the surveys [4, 6, 10, 15, 17, 22]) and the models indeed much more accurately capture the relative cost of algorithms on real machines than the RAM model does. This is true both in the context of algorithms that must run off disk when there is not enough main memory, and also in the context of algorithms that can fit in main memory, but not in various levels of the cache. For example, the models properly indicate that a blocked or hierarchical matrix-matrix multiply is much more efficient than the naïve triply nested loop ($\Theta\left(\frac{n^3}{B\sqrt{M}}\right)$ vs. $\Theta(n^3)$). In the RAM they have equal costs. The models also indicate that properly implemented versions of mergesort and quicksort are reasonably cache efficient but that samplesort and multiway mergesort are more efficient, and in fact optimal. Correspondingly all the fastest disk sorts indeed use some variant of samplesort or multiway mergesort, as the theory predicts [19].

Although the study of cache efficient algorithms has been very successful in identifying algorithms that are fast in practice, not surprisingly designing and programming algorithms for these models requires a careful layout of memory and careful management of space. Both temporal and spatial locality is critical in achieving good bounds. Spatial locality is important since memory is moved in blocks of size B, corresponding to either cache lines or memory pages. For example although merging two arrays of integers is reasonably efficient, the cost of merging two linked lists will depend on how the links are laid out in memory and needs to be considered with care. Care is also needed when allocating and freeing memory since touching unused memory incurs a cache miss. It is therefore important to reuse freed space immediately rather than returning it to a pool which might be evicted by the time it is reused—a generic memory allocator or garbage collection scheme will likely not do the right thing. To properly manage this problem, memory

POPL'13, January 23–25, 2013, Rome, Italy.

is typically preallocated and fully managed by the user/algorithm designer.

Needless to say, this form of programming is inconsistent with functional programming, especially when using recursive data types such as lists or trees. However, it is known experimentally that by using certain standard memory allocation schemes purely functional programs (no side effects) can be reasonably cache efficient with regards to both spatial and temporal locality [7, 8, 12, 23]. We give two examples.

Firstly, consider applying `map` with some simple function (e.g. increment) over a list of integers, and then applying the same `map` to the output. If the allocator keeps a pointer that gets incremented on each allocation, then after the first `map` all the cells of the list will be allocated adjacently. On the second `map` since the allocations are adjacent, reading the whole list will only incur $O(n/B)$ cache misses, where B is the block size, and evicting the newly generated blocks will also incur only $O(n/B)$ misses. This gives $O(n/B)$ cost, which asymptotically matches the cost of an optimal array version in an imperative setting. If the list were in an arbitrary order, the cost would be $O(n)$. All we have done is noted that the temporal locality of the allocations will lead to spatial locality of how they are laid out in memory.

Secondly, consider a block recursive matrix multiply on two $n \times n$ matrices. Such an algorithm will never require more than $O(n^2)$ live space but if recursion stops at problems of a constant size it will allocate a total of $O(n^3)$ space. Assuming that the maximum live space fits within the cache we should be able to run our matrix multiply with only $O(n^2/B)$ cache misses, needed for loading the two matrices and storing the result, but this would require being careful about reusing freed space that is already in cache. Fortunately generational garbage collectors have approximately this effect [8]. In particular if we make the first generation smaller than the size of the cache (M) then we will reclaim the memory whenever the allocation area fills, and reuse memory that is already in cache. This does not quite work in general since what is live at the time of the minor collection might get bumped from cache, but it gives some indication that it is not hopeless to make the natural recursive matrix multiply algorithm, as well as similar recursive algorithms, cache efficient.

We show that one can indeed implement cache-efficient algorithms in a call-by-value functional setting using recursive data types, and get provably efficient bounds on cache complexity. In particular we show that one can express algorithms at a high level using standard techniques and achieve optimal asymptotic performance when implemented on the ideal cache. Of course we do not expect the algorithm designer to understand the intricacies the garbage collector works in order to analyze their algorithm. Instead our approach consist of providing a reasonably high-level cost semantics that abstracts away from implementation details such as the garbage collection method, but still admits precise analysis of the cost of an algorithm on a two-level memory architecture. We then describe a provably-efficient implementation of the language on the ideal-cache model. We show that by using this implementation the costs analyzed in the high-level cost model asymptotically match the number of cache misses in the underlying ideal-cache model. The general idea of using high-level cost models based on a cost semantics along with a provable efficient implementation that maps the cost onto a lower level machine model has previously been used in the context of parallel cost models [5, 11, 13, 21].

Our high-level cost model consists of an operational semantics for a call-by-value variant of PCF in which we make explicit the allocation of and access to data objects. The store consists of three parts: a main memory, an allocation cache and a read cache. Both caches have size M and the memory is organized in blocks of size B (both measured in terms of abstract data objects). Data can migrate from the allocation cache to memory and from memory to the read cache, always in blocks of size B. Allocations are made in the allocation cache, and if the number of live objects in the cache exceeds M, then the B oldest locations are evicted to memory as a block, having unit cost. The read cache contains a subset of the memory blocks. A read has no cost if its location is in the read or allocation cache, otherwise it requires loading a block from memory into the read cache, having unit cost, and possibly ejecting an existing block. Hence the only costs are for evicting a block from the allocation cache or loading a block into the read cache. Since we are only concerned with measuring the traffic between main memory and cache memory, garbage collection for main memory is not modeled, but we do account for the detection of live objects, and their migration to main memory, when the cache limit is exceeded.

The provable implementation uses a generational collector to maintain the allocation cache. It uses a nursery of size $2M$ and allocates until the space runs out. It then traces the nursery for the live data. If there is $L > M$ live data, then $L - M$ locations are written to memory in blocks of B, leaving the nursery with at most M locations. The implementation allocates the stack in the heap and must amortize the cost of loading old stack frames against other operations since they are not modeled in the high-level cost semantics. We emphasize that the algorithm designer need not know anything about the garbage collector or how the stack is managed to analyze their algorithm; these concepts are only part of the provable implementation. The cost model is described in Section 3 and the provable implementation is described in Sections 4 and 5.

To demonstrate the utility of our approach, in Section 6 we describe some general techniques for analyzing the cost of algorithms in our model and show three examples of how to analyze the cost of algorithms in the model: mergeSort, k-way mergeSort and matrix multiply. Importantly our results on sorting and matrix multiply match the bounds for algorithms implemented directly in the ideal-cache model ($O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ and $O\left(\frac{n^3}{B\sqrt{N}}\right)$ respectively). The bounds for sorting are optimal. Because of our provable implementation bounds these results imply that on the ideal-cache model our algorithms written in a functional style using lists and trees are asymptotically as efficient as the low-level imperative programs. To analyze the algorithms we introduce the notion of a data structure being compact with respect to a traversal order. This is the way we capture the spatial locality of data structures in a language that has no explicit way to express memory layout.

Related Work

Although there has been a large amount of experimental work on showing how good garbage collection can lead to efficient use of caches and disks ([7, 8, 12, 23] and many references in [14]), we know of none that try to prove bounds for algorithms for functional programs when manipulating recursive data types such as lists or trees. Abello et. al. [1] show how a functional style can be used to design cache efficient graph algorithms. They however assume that data structures are in arrays (called lists), and that primitives for operations such as sorting, map, filter and reductions are supplied and implemented with optimal asymptotic cost (presumably at a lower level using imperative code). Their goal is therefore to design graph algorithms by composing these high-level operations on collections. They do not explain how to deal with garbage collection or memory management.

2. Background

I/O and Caching Models

The two-level I/O model of Aggarwal and Vitter [2] assumes a memory hierarchy consisting of main memory of size M and an un-

bounded secondary memory.[1] Both memories are partitioned into blocks of size B of consecutive memory locations. All computation must be performed from main memory, which is treated like a standard RAM, but there is an additional instruction for moving a block of memory from secondary memory to main memory and one for moving the other way. The cost of an algorithm is analyzed in terms of the number of block transfers—the cost of operations within the main memory is ignored. Many algorithms can be analyzed in this model and it is perhaps surprising how accurately it is able to capture the relative performance of algorithms. In their original work, for example, Aggarwal and Vitter showed tight upper and lower bounds for sorting n keys, with I/O cost $\Theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$. The two algorithm that match this bound are a multiway mergesort and a distribution sort, which are the standard algorithms used for disk based sorting, and they both perform significantly better than quicksort or standard mergesort. These algorithms are more efficient since they do not need to pass over the data as many times.

The I/O model can capture either the distinction between cache and main memory or between main memory and disk. In the first case the memory size corresponds to the cache size and the block size to the cache-line size, and in the second case the memory size corresponds to the main memory size and the block size to the page size (or whatever the transfer size between the disk and main memory is). One might note, however, that while the I/O model assumes two address spaces and the user explicitly moves data between them, a machine with caches assumes a single address space and makes its own decisions about what gets evicted from cache, e.g., using a least recently used (LRU) policy.

The ideal-cache model [9] can be used to better model a cache. It is similar to the I/O model but assumes the primary memory is treated as a cache with an ideal eviction policy. In particular the programmer only accesses one address space and a block is brought into the cache when a memory location is accessed whose block is not already in cache. Bringing in a block might require evicting another block from the cache. The model assumes that the best decision is always made, which is to evict the line used furthest in the future (the optimal off-line replacement policy). Since in practice we don't know the future, this is not possible on-line, but it is proved by Sleator and Tarjan's seminal work on competitive paging [20] that an LRU policy is always competitive with the optimal strategy (within constant factors in time and cache size). Therefore from a theoretical point of view the models are asymptotically the same. In this paper we will be using the ideal-cache model for simplicity although the results are also apply to the I/O model.

The ideal-cache model is often used in the context of cache-oblivious algorithms. These are simply algorithms for which the algorithm does not make any decisions based on the cache parameters M and B, although of course the analysis of cache complexity will depend on M and B. The advantage of cache-oblivious algorithms is that since they are oblivious to the cache parameters they work across multiple levels of a cache hierarchy simultaneously. Most of the algorithms in this paper are cache oblivious, but our k-way mergesort is not. We leave it as an open question whether it is possible to develop an I/O-efficient cache-oblivious sorting algorithm in our model.

Cache Efficient Algorithms

We now review some basic well known results on cache efficient algorithms in the imperative setting. The functional algorithms we present in Section 6 are based on the algorithms described here, but do not require arrays or explicitly memory management. We first consider mergesort. Throughout our discussion we assume that the elements being sorted each fit in a single machine word. All cache efficient algorithms we know of for sorting store the input and output elements directly in arrays.

First consider merging two arrays of keys A and B into an output array C of length n (as usual we assume the inputs are sorted in A and B in increasing order). The standard sequential algorithm for merging starts at the beginning of each array keeping a finger on each, finding the the lesser of the two keys at the fingers, copying this key to C, and incrementing the appropriate finger. This algorithm has a cache complexity $O(n/B)$ as long as $M \geq 3B$. This can be seen by noting that at any given time we only need one block from each of A, B and C resident in cache, and that we fully process the block before needing the next block. Therefore every block is only needed once.

For mergesort we assume the standard divide-and-conquer version, which recursively sorts each half of the array and then merges the result. Since merging as described cannot be done in place we have to be specific on how to manage memory. In particular allocating a new array for the result and then freeing the two old arrays using a general purpose memory allocator will likely not lead to the desired bounds (unless one can ensure special properties of the memory allocator). Instead the algorithm needs to pre-allocate a temporary array of length n and pass parts of this array to all subcalls. In particular mergesort could take as arguments both the input array and an equal length temporary array. The result is returned in the input array and the temporary array is used to merge into. Although these optimizations are relatively obvious and standard to programmers of imperative code, we bring them up to emphasize the care that needs to be taken to ensure the cache bounds—it is not simply an issue of reducing the number of calls to the memory allocator, it can actually asymptotically affect the cache bounds.

The cache complexity of this mergesort can be analyzed by considering two cases. The first is when the full computation fits in cache. In this case the two arrays need only be loaded into cache once and all the work can be done in cache. The problem fits in cache as long as $2n + \log n \leq M$, where the $\log n$ accounts for the stack size. The second case is when the problem does not fit in cache. In this case we have to pay for the cache misses on the two recursive calls plus the cache misses of the merge. This gives the following recurrence for the cache complexity $Q(n)$:

$$Q(n) \quad = \quad \begin{cases} 2Q(\frac{n}{2}) + O(\frac{n}{B}) & 2n + \log n > M \\ O(\frac{n}{B}) & \text{otherwise} \end{cases} \qquad (1)$$

The solution to this recurrence can be derived by noting that the top $\log_2(2n/M)$ levels of the recursion do not fit in memory while the lower levels do. The total cache complexity across each of the upper levels is $O(n/B)$ so the total overall cache complexity is $O(n/B \log_2(n/M))$. We note that this does not match the optimal cache complexity for sorting but is significantly better than simply assuming every access is a cache miss—specifically, a factor of $B \log n/(\log n - \log M)$ better. For sorting 10^{12} words in a memory with 10^9 words and a block size of 10^3 words, it is about a factor of about 4000 better. Quicksort has basically the same complexity as mergesort, although in the expected case. This is because scanning the input array to split it into the lesser and larger elements can be done using two fingers like in merging so again each block only needs to be loaded once.

We now describe a sort that is optimal for the I/O model. The idea is instead of partitioning the input array into two and recursively calling sort on each, to partition the input array into k parts, sort each part, and then merge all the parts. Since instead of having just two arrays to merge we have k arrays, we require a k-way merge. Without going into too much detail, such a merge

[1] Aggarwal and Vitter also considered a version of the model with "parallel" disk access, but most interesting results are explained with the single disk version.

can be implemented using one block of memory for each of the inputs needing to be merged as well as one block for the output. We keep a finger on each input and on each step select the minimum key at the fingers, move it to the output buffer and increment that finger. As long as all input blocks, the output block and any data for maintaining the fingers fit in cache, then the k-way merge will run with cache complexity $O(n/B)$, which is the same as the binary merge. Since there will be k input blocks and 1 output blocks, the space needed for the blocks is $(k+1)B$. Therefore accounting for overheads everything will fit in cache as long as $ckB \leq M$, or equivalently $k \leq M/(cB)$ for some constant c. We therefore pick k to be as large as possible, giving $k = M/(cB)$ As in the two way merge we need to be careful about allocation and preallocate temporary arrays to copy the output. We again can analyze the algorithm by considering the case when the problem fits in memory and when it does not. This gives the recurrence:

$$Q(n) \quad = \quad \begin{cases} \frac{M}{B} Q(\frac{n}{M/cB}) + O(\frac{n}{B}) & nc' > M \\ O(\frac{n}{B}) & \text{otherwise} \end{cases} \qquad (2)$$

where c and c' are constants, but n, B and M are variables. This solves to $O(n/B \log_{M/B}(n/B))$. This bound matches the lower bound for sorting in the I/O model [2] and hence also the ideal-cache model. The k-way mergesort is therefore asymptotically optimal.

3. Cost Semantics

In this section we define an *evaluation dynamics* that assigns a cost to a complete execution of a program. Following the I/O model, the cost measures the *cache complexity*, which is defined to be the traffic caused by the transfer of objects between the main memory and the memory cache. Accesses to objects in cache are considered to be cost-free, whereas migration of objects from cache into memory and from memory into the cache are charged unit cost. The dynamics is based on a two-level model of storage that includes a fixed-size allocation cache and a fixed-size read cache together with a main memory of unbounded size.

The evaluation dynamics provides the basis for assessing both the correctness and the cache complexity of programs. It is formulated at a sufficiently abstract level to free the programmer from having to reason directly about the compiler and run-time system, but is sufficiently concrete as to admit an implementation with a provable bound on its cache complexity. Thus, we may achieve the same overall results as are obtained using only low-level machine models in previous work on I/O algorithms, while working at the much more practical level of abstraction offered by functional programming languages.

We give the dynamics of a call-by-value variant of Plotkin's PCF language [18]. The syntax of expressions is summarized by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid \mathtt{z} \mid \mathtt{s}(e) \mid \mathtt{ifz}(e; e_0; x.e_1) \mid \\ & & \mathtt{fun}(x, y.e) \mid \mathtt{app}(e_1; e_2) \end{array}$$

The conditional tests whether a number is zero or not, and passes the predecessor to the non-zero case. Functions are equipped with a name for themselves to allow for recursion. The typing rules are standard, and are omitted here for the sake of concision. (See, for example, Chapter 10 of [13].)

For illustrative purposes natural numbers are treated as heap-allocated data structures of unbounded size (as will become evident shortly). It is straightforward to extend the language to account for a richer variety of data structures, including sum, product, finite sequence, and recursive types, and to account for typical hardware-oriented concepts such as machine words and floating point numbers.

Storage Model

Following Morrisett, *et al.* [16], the dynamics distinguishes *large* from *small* values, with large values being allocated in memory and represented by a location, and small values being those that are manipulated directly. In the present case the only small values are locations, but it is also possible to consider, for example, fixed-sized numbers as forms of small value. Correspondingly, all other forms of value (numbers and functions) are large. We also allocate *stack frames*, which reify the control state of evaluation, in memory. A *memory object* is either a large value of a stack frame.

The two-level memory model is parameterized by two constants, the *block size* B, and the *cache size* $M = c \times B$ determined by some constant c representing the number of blocks in the cache.

A *memory* μ is a finite mapping assigning a memory object to each of a finite set $\mathrm{dom}(\mu)$ of abstract locations. The memory may grow without bound. (We do not consider here the separate problem of garbage collection for main memory, for which see Morrisett, *et al.* [16].) As a technical convenience, we assume that locations are divided into two classes, *value locations*, l, and *stack locations*, s, and require that a memory map value locations to large values and stack locations to stack frames. When the distinction is immaterial, we speak simply of locations and objects in memory.

A memory μ comes equipped with an equivalence relation $l \equiv_\mu l'$ over $\mathrm{dom}(\mu)$ specifying that l and l' are *neighbors* in μ. Additionally, we require that each equivalence class in the domain of a memory is of size B. A memory whose domain consists of a single equivalence class of size B is called a *block*. The *neighborhood* $\mathrm{nbhd}(\mu, l)$ of a location $l \in \mathrm{dom}(\mu)$ is the restriction of μ to the neighbors of l in μ, a single block. The *expansion* $\mu \oplus \beta$ of a memory μ by a block β such that $\mathrm{dom}(\beta) \cap \mathrm{dom}(\mu) = \emptyset$ is the memory μ' that agrees with μ and β on their respective domains and for which $l \equiv_{\mu'} l'$ iff $l \equiv_\mu l'$ or $l \equiv_\beta l'$.

There are two forms of cache mediating access to memory. A *read cache* ρ for a memory μ is the restriction of μ to a finite set of locations of size at most M. The *contraction* $\rho \ominus \beta$ of a read cache ρ by a block $\beta \subseteq \rho$ is the read cache ρ' such that $\rho = \rho' \oplus \beta$. A *nursery* ν is a finite mapping that associates an object to each a finite set $\mathrm{dom}(\nu)$ of locations. A nursery comes equipped with a linear ordering $l \prec_\nu l'$ of $\mathrm{dom}(\nu)$, called the *allocation ordering*. If $l \prec_\nu l'$ we say that l is older than l' and that l' is newer than l in ν. The *extension* $\nu[l \mapsto o]$ of a nursery ν binding a location $l \notin \mathrm{dom}(\mu)$ to an object o is the nursery ν' such that (1) $\nu'(l) = o$ and $\nu'(l') = \nu(l')$ for each $l' \in \mathrm{dom}(\nu)$, and (2) $l' \prec_{\nu'} l$ for every $l' \in \mathrm{dom}(\nu)$. The *contraction* $\nu \ominus \beta$ of a nursery ν by a block $\beta \subseteq \nu$ is the restriction of ν to $\mathrm{dom}(\nu) \setminus \mathrm{dom}(\beta)$.

The *live* locations $\mathrm{live}(R, \nu)$ in a nursery ν relative to a subset $R \subseteq \mathrm{dom}(\nu)$ consists of those locations in $\mathrm{dom}(\nu)$ that are (transitively) reachable from locations in R. The *scan* $\mathrm{scan}(R, \nu)$ of a nursery ν with respect to a subset $R \subseteq \mathrm{dom}(\nu)$ is the block β of consisting of the oldest B live locations in $\mathrm{live}(R, \nu)$. (See Morrisett, *et al.* [16] for formal definitions of these standard concepts.) It will be an invariant of the dynamics that the nursery contains at most M live objects relative to the roots of the computation.

A *store* σ is a triple (μ, ρ, ν) consisting of a memory μ, a read cache ρ for μ, and a nursery ν such that $\mathrm{dom}(\nu) \cap \mathrm{dom}(\mu) = \emptyset$. The *domain* of a store σ is defined by $\mathrm{dom}(\sigma) = \mathrm{dom}(\mu) \cup \mathrm{dom}(\nu)$. An *initial store* is a store in which the main memory contains only large values and in which the read cache and allocation area are empty.

Evaluation Dynamics

The overall goal of the evaluation dynamics is to define the evaluation of a closed expression by an inductive definition of a relation between an expression and its value, which is always small, and its cost, a non-negative integer. The cost is computed by tracking the

$$\frac{\sigma \mathbin{@} \mathtt{z} \uparrow^{n}_{R} \sigma' \mathbin{@} l'}{\sigma \mathbin{@} \mathtt{z} \Downarrow^{n}_{R} \sigma' \mathbin{@} l'} \tag{3a}$$

$$\frac{\left\{\begin{array}{c}\sigma \mathbin{@} \mathtt{s}(-) \uparrow^{n}_{R \cup \mathrm{locs}(e')} \sigma' \mathbin{@} s' \quad \sigma' \mathbin{@} e' \Downarrow^{n'}_{R \cup \{s'\}} \sigma'' \mathbin{@} l'' \\ \sigma'' \mathbin{@} \mathtt{s}(l'') \uparrow^{n''}_{R} \sigma''' \mathbin{@} l'''\end{array}\right\}}{\sigma \mathbin{@} \mathtt{s}(e') \Downarrow^{n+n'+n''}_{R} \sigma''' \mathbin{@} l'''} \tag{3b}$$

$$\frac{\left\{\begin{array}{c}\sigma \mathbin{@} \mathtt{ifz}(-; e_2; x.e_3) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} s_1 \\ \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{s_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_2 \mathbin{@} \mathtt{z} \quad \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R} \sigma' \mathbin{@} l'\end{array}\right\}}{\sigma \mathbin{@} \mathtt{ifz}(e_1; e_2; x.e_3) \Downarrow^{n_1+n_1'+n_1''+n_2}_{R} \sigma' \mathbin{@} l'} \tag{3c}$$

$$\frac{\left\{\begin{array}{c}\sigma \mathbin{@} \mathtt{ifz}(-; e_2; x.e_3) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} s_1 \\ \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{s_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_3 \mathbin{@} \mathtt{s}(l_1'') \quad \sigma_3 \mathbin{@} [l_1''/x]e_3 \Downarrow^{n_3}_{R} \sigma' \mathbin{@} l'\end{array}\right\}}{\sigma \mathbin{@} \mathtt{ifz}(e_1; e_2; x.e_3) \Downarrow^{n_1+n_1'+n_1''+n_3}_{R} \sigma' \mathbin{@} l'} \tag{3d}$$

$$\frac{\sigma \mathbin{@} \mathtt{fun}(x, y.e) \uparrow^{n}_{R} \sigma' \mathbin{@} l}{\sigma \mathbin{@} \mathtt{fun}(x, y.e) \Downarrow^{n}_{R} \sigma' \mathbin{@} l} \tag{3e}$$

$$\frac{\left\{\begin{array}{c}\sigma \mathbin{@} \mathtt{app}(-; e_2) \uparrow^{n_1}_{R \cup \mathrm{locs}(e_1)} \sigma_1 \mathbin{@} s_1 \quad \sigma_1 \mathbin{@} e_1 \Downarrow^{n_1'}_{R \cup \{s_1\}} \sigma_1' \mathbin{@} l_1' \\ \sigma_1' \mathbin{@} l_1' \downarrow^{n_1''} \sigma_1'' \mathbin{@} \mathtt{fun}(x, y.e) \quad \sigma_1'' \mathbin{@} \mathtt{app}(l_1'; -) \uparrow^{n_1'''}_{R} \sigma_2 \mathbin{@} s_2 \\ \sigma_2 \mathbin{@} e_2 \Downarrow^{n_2}_{R \cup \{s_2\}} \sigma_2' \mathbin{@} l_2' \quad \sigma_2' \mathbin{@} [l_1', l_2'/x, y]e \Downarrow^{n_2}_{R} \sigma' \mathbin{@} l'\end{array}\right\}}{\sigma \mathbin{@} \mathtt{app}(e_1; e_2) \Downarrow^{n_1+n_1'+n_1''+n_1'''+n_2+n_2'}_{R} \sigma' \mathbin{@} l'} \tag{3f}$$

Figure 1. Cost Dynamics

$$\frac{l \in \mathrm{dom}(\rho)}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{0} (\mu, \rho, \nu) \mathbin{@} \rho(l)} \tag{4a}$$

$$\frac{l \in \mathrm{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{0} (\mu, \rho, \nu) \mathbin{@} \nu(l)} \tag{4b}$$

$$\frac{l \notin \mathrm{dom}(\rho) \cup \mathrm{dom}(\nu) \quad |\mathrm{dom}(\rho)| \leq M - B}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{1} (\mu, \rho \oplus \mathrm{nbhd}(\mu, l), \nu) \mathbin{@} \mu(l)} \tag{4c}$$

$$\frac{l \notin \mathrm{dom}(\rho) \cup \mathrm{dom}(\nu) \quad |\mathrm{dom}(\rho)| = M \quad \beta \subseteq \rho}{(\mu, \rho, \nu) \mathbin{@} l \downarrow^{1} (\mu, \rho \ominus \beta \oplus \mathrm{nbhd}(\mu, l), \nu) \mathbin{@} \mu(l)} \tag{4d}$$

$$\frac{|\mathrm{live}(R \cup \mathrm{locs}(o), \nu)| < M \quad l \notin \mathrm{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} o \uparrow^{0}_{R} (\mu, \rho, \nu[l \mapsto o]) \mathbin{@} l} \tag{5a}$$

$$\frac{|\mathrm{live}(R \cup \mathrm{locs}(o), \nu)| = M \quad \beta = \mathrm{scan}(R \cup \mathrm{locs}(o), \nu) \quad l \notin \mathrm{dom}(\nu)}{(\mu, \rho, \nu) \mathbin{@} o \uparrow^{1}_{R} (\mu \oplus \beta, \rho, (\nu \ominus \beta)[l \mapsto o]) \mathbin{@} l} \tag{5b}$$

Figure 2. Reading and Allocation

movement of objects among the components of the store, charging one unit of cost whenever a block of objects must be moved to or copied from main memory, and charging zero cost otherwise. (So, for example, a computation that runs entirely in cache will be assigned zero cost, consistently with the I/O model.) To account for the memory traffic involving values, the dynamics makes explicit the allocation of objects in the nursery, their eviction to main memory when the capacity of the nursery is exceeded, and their movement into the read cache as they are required by the computation. To account for the memory traffic attributable to the implicit control stack, the dynamics also allocates (but does not otherwise use) stack frames, and ensures that any data that would appear in the stack is kept live by the dynamics.

These considerations lead to the evaluation judgment

$$\sigma \mathbin{@} e \Downarrow^{n}_{R} \sigma' \mathbin{@} l$$

stating that the expression e, when evaluated with respect to a store σ such that $\mathrm{dom}(\sigma) \supseteq \mathrm{locs}(e)$ and to roots $R \subseteq \mathrm{dom}(\sigma)$, results in a modified store σ', a location l representing the (large) value of the expression, and a cost n representing the cache complexity of the execution. The modifications to the store consist of allocations in the nursery, migrations of objects from the nursery to the main memory, and copying of objects from the main memory to the read cache. All memory traffic occurs in blocks of B objects, corresponding to loading a cache line or reading a block from disk. The roots R represent locations that are to be kept live by virtue of their being present in the implicit control stack or expression under evaluation.

The evaluation judgment is defined by the rules in Figure 1, making use of two auxiliary judgments for reading and allocating objects defined in Figure 2. It may be helpful to read through the rules once while ignoring all but the evaluation judgments to see that the rules define a conventional eager dynamics for a functional language. On such a reading the root set plays no role, and can be ignored. Moreover, the cost assignment has no significance under such a simplification.

Next, let us consider the roles of the *read* judgments $\sigma \mathbin{@} l \downarrow^{n} o \mathbin{@} \sigma'$ and the *allocate* judgments $\sigma \mathbin{@} v \uparrow^{n}_{R} \sigma' \mathbin{@} l$, where v is a value, in the dynamics. The quoted read judgment states that the result of reading location l in store σ results in the object o and the modified store σ', and has cost $n = 0$ or $n = 1$. The cost is non-zero only if the read causes a block to be loaded into the read cache. The modified store represents the possible effect of loading a block into the read cache. The quoted write judgment states that allocating the large value v in store σ results in a modified store σ' and location $l \in \mathrm{dom}(\sigma')$, and has cost $n = 0$ or $n = 1$. The cost is non-zero only if the allocation causes the eviction of a block from the nursery in order to maintain the live-size invariant. The read and allocate operations in the dynamics record the memory traffic engendered by the creation and examination of values during computation.

It remains to consider the role of the allocation judgments of the form $\sigma \mathbin{@} s \uparrow^{n}_{R} \sigma' \mathbin{@} f$, which represent the allocation of a stack frame in the store at stack location s. The purpose of allocating these frames is purely to ensure that the cost assigned to a computation is accurate with respect to the underlying implementation. Although an evaluation semantics has no explicit control stack, it is nevertheless the case that an implementation must allocate space for the representation of the control state, and this space allocation does influence the cache behavior of the computation. It may not, therefore, be ignored. Our method for accounting for the memory effects of the control stack is to allocate explicitly frames that would appear in the control stack to ensure that space usage is properly accounted for, and that required liveness information (to be

detailed shortly) is properly maintained. The frames are denoted as $\texttt{app}(-; e_2)$ and $\texttt{app}(l_1'; -)$ in the cost dynamics.

With this in mind, let us examine in detail Rule 3f in Figure 1. We are to evaluate and determine the cost of $\texttt{app}(e_1; e_2)$ in store σ with given roots R. First, we allocate a stack frame s_1 representing the pending evaluation of e_2 during the evaluation of e_1. This frame is now considered live, even though it does not appear in any expression under consideration. Accordingly, we evaluate e_1 relative to the store containing this frame, treating the just-allocated stack pointer to be live (as indicated by $\Downarrow_{R\cup\{s_1\}}$). This results in a location l_1', which we then read from the store to obtain a function abstraction (as would be guaranteed by the static type discipline omitted here). We then create another frame s_2 corresponding to the suspended application of the function at location l_1', and evaluate e_2 with this stack pointer considered live (as indicated by $\Downarrow_{R\cup\{s_2\}}$) to obtain location l_2'. Finally, we evaluate the function body, replacing the "self" variable by l_1' and the argument by l_2'. The overall cost of the computation is the sum of the costs of each of these steps, which are given either inductively or by the uses of the read and allocate judgments. Observe that this rule properly accounts for tail recursion in that no extra space is held during tail recursive calls (as indicated by $\Downarrow_R$).

It remains to explain the read and allocate judgments defined in Figure 2. The read judgment assigns cost zero to any read from a location in either the nursery or the read cache (Rules 4b and 4a). Such reads have no effects, and hence induce no cache traffic. A read of a location that is only in main memory induces a load of the neighborhood of that location (a block of memory) into the read cache. If there is sufficient room for it in the read cache, the block is added to the cache and the contents is returned, at a cost of one unit (Rule 4c). If there is insufficient room in the read cache, a block is selected non-deterministically to be replaced by the required block, and once again a unit cost is charged to the read (Rule 4d). At the end of the section we discuss the use of non-deterministic eviction.

The allocate judgement defines the procedure for creating new objects in the store. Of course, new objects are considered newer in the allocation ordering than the objects already present in the nursery. If the new object fits within the nursery, it is allocated there at zero cost (Rule 5a). If the new object will not fit within the nursery, then the block consisting of the oldest B live objects in the nursery is evicted to main memory, making room for the newly allocated object; such an allocation is charged unit cost (Rule 5b). It is important to our method that the oldest objects be evicted from the cache as a block forming the neighborhood of each of its locations. Whether an object fits within the nursery is determined as follows. The nursery is *full* if the number of *live* objects in it is exactly M. (It is for the sake of assessing liveness that the allocation judgment is parameterized by a root set.) Eviction of a block reduces this to at most $M - B$ objects, so that the next $B - 1$ allocations will not cause an eviction. Thus we are, in effect, charging at most $1/B$ units of cost to each allocation (less if objects die before needing to be evicted).

It is essential to our results that the liveness of objects in the nursery may be assessed *without accessing main memory*. Given roots R we need only trace objects in the nursery itself, and need never consider locations lying outside of it. This is ensured by two properties of the dynamics. First, since the model is purely functional, the dependency graph of objects in the nursery is acyclic; an object may only refer to objects allocated earlier in the computation as defined by the allocation ordering. Second, implicit stack frames are explicitly allocated in the nursery to ensure that liveness may be assessed solely by examining the nursery itself, starting from the root set. Put another way, an object in the nursery cannot be live solely because of a pointer from main memory back to the nursery. This property is a consequence of immutability and the explicit allocation of stack frames in the semantics.

In Section 6 we will make use of a *deep copy* operation on values of certain types. In the illustrative language considered here this operation is definable on natural numbers as follows:

$$\texttt{fun}(\mathit{copy}, x.\texttt{ifz}(x; \texttt{z}; x'.\texttt{s}(\texttt{app}(\mathit{copy}; x')))).$$

Calling this function on a number n has the effect of creating a "fresh copy" of n in the heap. No such operation is definable, or required, for function types. Deep copying is easily extended to product, sum, and inductive types, but would need to be provided as a primitive for base types such as fixed precision integers or floating point numbers.

Discussion

We briefly discuss some of the motivation for the decisions we made in formulating the dynamic semantics. The overall goal is to allow a simple analysis for the algorithm developer while capturing all the costs needed to prove asymptotic implementation bounds.

The separate allocation cache is important both for convenience of analysis and properly accounting for costs. It ensures that all short lived allocations never need to be allocated to memory. For a subcomputation in which the maximum footprint of live data allocated fits in the allocation cache, the user need not worry about any costs for any temporary memory. In a block matrix multiply on $n \times n$ matrices, for example, once $kn^2 \leq M$ for some small constant k, the only cost that needs be considered is the cost of reading the input and evicting the output. This is the case even though the multiply will allocate a total of $\Theta(n^3)$ space. It is also important that the partitioning of locations into blocks is not decided until locations are evicted from the allocation cache, which ensures that only live data is ever migrated to memory. If blocking were to be decided on allocation, for example, then by the time the objects are evicted most of the objects in a block may no longer be live. This would break the bounds we give in Section 6.

The cost semantics accounts for the allocation of stack frames in order to account for the space required to manage the control state of evaluation. This is particularly important in the case that no allocation is associated with the creation of a frame, for then there is no possibility to amortize the space required for the frame against the allocated object. Note that the semantics only models the space taken by the frames in the allocation cache and the cost of evicting them. It does not model any costs associated with reloading them into the read cache. As described in the next section, in a lower level model this can be amortized against the cost of evicting the frames in the first place.

It is important that the stack frames be heap allocated. A crucial invariant we require is that all live data in the allocation cache can be determined solely through the caches. If we had a separate stack cache it could allow for the eviction of a stack frame that references data in the allocation cache, breaking the required invariant. There are other techniques to handle this problem but we found that allocating the stack frames in the heap is the easiest.

Our model is non-deterministic in the choice of what block is evicted from the read cache in the case of a read miss. In our provable implementation bounds we show that if there is a (non-deterministic) execution that gives certain cache complexity then we can guarantee those bounds on the ideal cache model (within constant factors). When analyzing an algorithm this allows one to consider any policy for eviction. This is possible because the ideal cache makes the optimal decisions and will therefore be at least as good as the policy the user assumes. The justification for the ideal cache model is given in Section 2.

4. Abstract Machine

The abstract cost of a computation assigned by the evaluation dynamics given in the preceding section is validated in two stages. First, in this section we define an abstract machine with an explicit control stack, and show that the evaluation dynamics accurately predicts the behavior of the abstract machine with respect to both the outcome and the cost of the computation. Second, in Section 5 we show how to implement the basic operations of the abstract machine with only a small overhead. Taken together these two arguments demonstrate that the evaluation semantics provides an accurate model of the cache complexity of a program when implemented as described in these two steps.

The abstract machine takes the form of a labeled transition system between states of two different forms:

1. Evaluation state: $\sigma \mathbin{@} k \triangleright e$, where $k \in \mathrm{dom}(\sigma)$ is a stack pointer, and $\mathrm{locs}(e) \subseteq \mathrm{dom}(\sigma)$, stating that e is to be evaluated on stack k relative to store σ.
2. Return state: $\sigma \mathbin{@} k \triangleleft l$, where $k, l \in \mathrm{dom}(\sigma)$, stating that small value l is to be returned to stack k relative to store σ.

The control stack is represented by a stack location, k, that refers to a linked list of frames, either the empty stack, written $\bullet$, or a frame together with another stack location, written $f;k$. The label on a transition is either 0, 1, or 2, and specifies the amount of work to be charged for that transition.

The rules given in Figure 3 define the abstract machine. Their overall form is standard (see, for example, Chapter 27 of [13]), with the main differences being that allocation and reading of values is made explicit, just as in the evaluation dynamics, and that the stack is explicitly represented as a linked data structure in the store. The multistep transition judgment $s \overset{n}{\longmapsto}{}^{*} s'$ means that there is a finite, possibly empty, sequence of transitions from s to s' whose labels sum to n.

THEOREM 4.1 (Correctness of Evaluation Dynamics). *Let σ_0 be an initial store, let e_0 be a closed expression such that locs$(e_0) \subseteq$ dom(σ_0). Let the abstract machine be equipped with one additional block in the read cache, and let k_0 be a reserved stack location not used in the evaluation dynamics. If*

$$\sigma_0 \mathbin{@} e_0 \Downarrow^{n}_{\emptyset} \sigma \mathbin{@} l,$$

then there is an evaluation

$$\sigma_0[k_0 \mapsto \bullet] \mathbin{@} k_0 \triangleright e_0 \overset{m}{\longmapsto}{}^{*} \sigma'[k_0 \mapsto \bullet] \mathbin{@} k_0 \triangleleft l'$$

such that

1. *the results are isomorphic, $\sigma \mathbin{@} l \cong \sigma' \mathbin{@} l'$, and*
2. *the cost m is at most $3n$.*

The relation $\sigma \mathbin{@} l \cong \sigma' \mathbin{@} l'$ states that the reachable graph from l in σ is isomorphic to the reachable graph from l' in σ'.

Theorem 4.1 states that the outcome of a computation on the abstract machine is the same, up to choice of locations, as the outcome of the same computation according to the evaluation dynamics. Moreover, the total cost of the machine execution (measured in accordance with the I/O model described earlier) is at most a small constant factor larger than the cost assigned by the evaluation dynamics. The content of the theorem amounts to a proof that the space required by the control stack in a computation may be managed so as not to interfere with space usage of the computation itself.

The correctness proof may be decomposed into three major components. The first obligation is to relate the outcome of the evaluation dynamics to that of the abstract machine, disregarding, for the moment, the cost. The required correspondence is proved

$$\frac{}{\sigma \mathbin{@} k \triangleright l \overset{0}{\longmapsto} \sigma \mathbin{@} k \triangleleft l} \tag{6a}$$

$$\frac{\sigma \mathbin{@} \mathtt{z} \uparrow^{n}_{\{k\}} \sigma' \mathbin{@} l}{\sigma \mathbin{@} k \triangleright \mathtt{z} \overset{n}{\longmapsto} \sigma' \mathbin{@} k \triangleleft l} \tag{6b}$$

$$\frac{\sigma \mathbin{@} \mathtt{s}(-);k \uparrow^{n}_{\mathrm{locs}(e')} \sigma' \mathbin{@} k'}{\sigma \mathbin{@} k \triangleright \mathtt{s}(e') \overset{n}{\longmapsto} \sigma' \mathbin{@} k' \triangleright e'} \tag{6c}$$

$$\frac{\sigma \mathbin{@} k \downarrow^{n} \sigma' \mathbin{@} \mathtt{s}(-);k' \quad \sigma' \mathbin{@} \mathtt{s}(l) \uparrow^{n'}_{\{k'\}} \sigma'' \mathbin{@} l'}{\sigma \mathbin{@} k \triangleleft l \overset{n+n'}{\longrightarrow} \sigma'' \mathbin{@} k' \triangleleft l'} \tag{6d}$$

$$\frac{\sigma \mathbin{@} \mathtt{ifz}(-; e_2; x.e_3);k \uparrow^{n}_{\mathrm{locs}(e_1)} \sigma' \mathbin{@} k'}{\sigma \mathbin{@} k \triangleright \mathtt{ifz}(e_1; e_2; x.e_3) \overset{n}{\longmapsto} \sigma' \mathbin{@} k' \triangleright e_1} \tag{6e}$$

$$\frac{\sigma \mathbin{@} k \downarrow^{n_1} \sigma' \mathbin{@} \mathtt{ifz}(-; e_2; x.e_3);k' \quad \sigma' \mathbin{@} l \downarrow^{n_2} \sigma'' \mathbin{@} \mathtt{z}}{\sigma \mathbin{@} k \triangleleft l \overset{n_1+n_2}{\longrightarrow} \sigma'' \mathbin{@} k' \triangleright e_2} \tag{6f}$$

$$\frac{\sigma \mathbin{@} k \downarrow^{n_1} \sigma' \mathbin{@} \mathtt{ifz}(-; e_2; x.e_3);k' \quad \sigma' \mathbin{@} l \downarrow^{n_2} \sigma'' \mathbin{@} \mathtt{s}(l')}{\sigma \mathbin{@} k \triangleleft l \overset{n_1+n_2}{\longrightarrow} \sigma'' \mathbin{@} k' \triangleright [l'/x]e_3} \tag{6g}$$

$$\frac{\sigma \mathbin{@} \mathtt{fun}(x, y.e) \uparrow^{n}_{\{k\}} \sigma' \mathbin{@} l'}{\sigma \mathbin{@} k \triangleright \mathtt{fun}(x, y.e) \overset{n}{\longmapsto} \sigma' \mathbin{@} k \triangleleft l'} \tag{6h}$$

$$\frac{\sigma \mathbin{@} \mathtt{app}(-; e_2);k \uparrow^{n}_{\mathrm{locs}(e_1)} \sigma_1 \mathbin{@} k_1}{\sigma \mathbin{@} k \triangleright \mathtt{app}(e_1; e_2) \overset{n}{\longmapsto} \sigma_1 \mathbin{@} k_1 \triangleright e_1} \tag{6i}$$

$$\frac{\left\{\begin{array}{c}\sigma \mathbin{@} k \downarrow^{n_1} \sigma_1 \mathbin{@} \mathtt{app}(-; e_2);k_1 \\ \sigma_1 \mathbin{@} \mathtt{app}(l_1; -);k_1 \uparrow^{n_2}_{\mathrm{locs}(e_2)} \sigma_2 \mathbin{@} k_2\end{array}\right\}}{\sigma \mathbin{@} k \triangleleft l_1 \overset{n_1+n_2}{\longrightarrow} \sigma_2 \mathbin{@} k_2 \triangleright e_2} \tag{6j}$$

$$\frac{\sigma \mathbin{@} k \downarrow^{n_1} \sigma_1 \mathbin{@} \mathtt{app}(l_1; -);k_2 \quad \sigma_1 \mathbin{@} l_1 \downarrow^{n_2} \sigma_2 \mathbin{@} \mathtt{fun}(x, y.e)}{\sigma \mathbin{@} k \triangleleft l_2 \overset{n_1+n_2}{\longrightarrow} \sigma_2 \mathbin{@} k_2 \triangleright [l_1, l_2/x, y]e} \tag{6k}$$

Figure 3. Abstract Machine

by induction on the derivation of evaluation judgment. Specifically, we prove that if $\sigma \mathbin{@} e \Downarrow^{n}_{R} \sigma' \mathbin{@} l$, then for any stack pointer k,

$$\sigma \mathbin{@} k \triangleright e \mapsto^{*} \sigma \mathbin{@} k \triangleleft l.$$

The proof proceeds along standard lines, as described, for example, in Chapter 27 of the second author's textbook [13]. The same choice of locations may be made in the machine derivation as were made in the evaluation derivation, because the sequence of value allocations is precisely the same in both forms of dynamics.

The next step of the proof is to show that the abstract machine performs the same sequence of value reads in the same order as specified by the evaluation dynamics. This may be proved along with the correspondence described in the preceding paragraph. The argument relies on two important properties of the evaluation dynamics:

1. Any read of a value location is either a read of a location in the initial store, or a location that was allocated earlier in the evaluation.
2. Stack frames are allocated, but never read, in order to ensure that eviction of blocks from the nursery occurs in exactly the order imposed by the stack-based abstract machine.

A deterministic nursery eviction policy is required to ensure that the memory reads correspond exactly between the evaluation dynamics and the abstract machine. We can assume whatever policy is used the the dynamic semantics is also used by the abstract machine.

It remains to show that the stack reads employed by the abstract machine do not impose an asymptotically significant cost beyond what is predicted by the evaluation dynamics. Without special provision, access to the control stack would interfere with the allocation of data in the read cache, invalidating the cost given to the computation by the evaluation dynamics. To avoid this we make use of a dedicated read cache block in the abstract machine, which we will call the *stack cache block*, and explicitly manage this cache block as follows. Whenever a stack location is read from main memory, its neighborhood is loaded into the stack cache block, evicting the block that currently occupies it. We will argue that the cost of loading the stack cache can be amortized across the execution sequence, even if the same block is loaded into the stack cache more than once, a possibility that will be detailed shortly. The validity of the argument depends on two special properties of the run-time stack, namely that each allocated frame is read exactly once in a complete computation, and that the preceding stack frame is always older than the current one. Given such an amortization, it is then clear that the overall cost of execution on the abstract machine is bounded by a small constant factor of the cost ascribed to it by the evaluation dynamics, establishing the theorem.

To complete the proof, we describe the amortization of the cost of stack management in more detail. As an invariant we put a "dollar" on every memory block that contains a stack frame, except if it is the youngest such block and resides in the stack cache block, in which case it has no "money" associated with it. When the abstract machine evicts a block containing a stack frame from the allocation cache we spend three dollars—one for the eviction itself, one to put a dollar on the evicted block, and one to put a dollar on the block that is in the cache stack block. This third dollar might be needed to maintain our invariant since that block, if there is one, will no longer be the youngest memory block containing a stack frame. Now when the abstract machine loads a block into the stack cache block from memory we spend its dollar for the load. All blocks with older frames have a dollar on them already by the invariant, so the invariant is maintained. In summary we spend 3 block transfers (worst case) per block that is evicted from the allocation cache.

We finally note that there is no need to explicitly maintain the stack cache block in the abstract machine semantics since we are assuming an "ideal cache". Therefore as long as the cache has an extra block available, then the cache policy will do at least as well as the one we described.

5. Provable Implementation

In this section we describe an implementation of the abstract machine given in Section 4 in the ideal cache model with the same asymptotic cost. The efficiency proof for the implementation takes account of two issues that are treated abstractly in the evaluation semantics and in the abstract machine. The main issue is how to implement the allocation judgment defined in Figure 2. Rules 5a and 5b make reference to liveness of the data, and evict a block consisting of the oldest B live objects in the nursery. To ensure that the predicted costs are realized in practice we must argue that these conditions can be met by an implementation. The second issue is that we must account for the size of the stored objects (values and frames) that may appear in a computation, and account for the cost of handling these objects in an implementation.

Define the *size* of a machine state $\sigma_0 \mathbin{@} k_0 \triangleright e_0$ be the sum of the size of e_0 and the size of any function in σ_0. This may be thought of as the size of the program, including any λ-abstractions that may be present in the initial store.

THEOREM 5.1. *Fix an initial state $\sigma_0 \mathbin{@} k_0 \triangleright e_0$ of size s_0, and consider a complete computation*

$$\sigma_0 \mathbin{@} k_0 \triangleright e_0 \stackrel{m}{\longmapsto} \sigma \mathbin{@} k_0 \triangleleft l$$

with s_1 objects in the final store, σ. This computation be simulated in the ideal cache model with cache complexity $c \times m$ for some constant c, provided that words are of size at least $d \log(\max(s_0, s_1))$ for some $d > 0$ and that the cache has at least $(4M + B) \times s_0$ words. (The constants c and d are independent of σ_0, k_0 and e_0.)

Theorem 5.1 states that the implementation asymptotically realizes the work attributed to the computation by the evaluation semantics, and hence validates the algorithm analysis performed using that semantics.

The requirement on the word size in Theorem 5.1 ensures that all objects are addressable by a word-sized pointer, and accounts for the sizes of the objects themselves in storage. (A closure can be as large as the initial program.) The requirement on the cache size in Theorem 5.1 ensures that we may implement the abstract memory hierarchy with no more than a small constant factor of overhead in a manner that we now describe. (The $B \times s_0$ additional words account for the stack cache described in Section 4; it remains to discuss the implementation of allocation.)

The allocation judgment defined in Figure 2 relies on an assessment of the live size of the nursery, and on the eviction of blocks from the nursery to ensure that the nursery contains no more than M live objects. As we note earlier, the liveness of data in the nursery may be assessed without reference to the main memory; the liveness computation takes place entirely within the cache. Rather than assess liveness, possibly evicting a block, on each allocation, we instead amortize these costs across multiple allocations according to the following strategy. We reserve $2M \times s_0$ words of cache memory for the allocation area to accommodate at least $2M$ objects. Objects are allocated by maintaining a pointer into the nursery area, incrementing it on each allocation until $2M$ objects have been allocated, at which point the nursery space is exhausted. When this occurs, we perform a compacting garbage collection that preserves the allocation order of objects, simultaneously evicting as many blocks as necessary to obtain a live size of M objects in the nursery. After compaction, allocation continues as before until the nursery is again exhausted.

As long as there is sufficient space, allocation takes constant time. When a garbage collection is required, the cost may be attributed to the allocations of the live data in the nursery, so that in an amortized sense garbage collection is cost-free [3]. It is easy to see that no object is evicted to main memory using this implementation that would not have been evicted in the abstract sense. However, the evictions will, in general, happen later than predicted by the semantics. As a result, fewer objects may be live at the time of eviction, and so fewer blocks overall may be moved to main memory. As a result of this compression effect, two locations that were neighbors in the evaluation semantics may be in two different blocks in the implementation. To account for this, two blocks must be loaded to ensure that neighboring objects in the semantics are loaded into the read cache together. Thus we require $2M \times s_0$ words of cache in the ideal cache model to account for the M objects in the read cache.

With regards to the eviction policy from the read cache we note that an ideal cache will always choose an optimal policy (furtherst in the future). It will therefore do at least as well as any policy assumed by the abstract machine.

This completes the proof of the implementation bound stated in Theorem 5.1.

6. I/O Efficient Algorithms

We now describe algorithms analyzed in the model and prove bounds on their cache complexity. In particular we will consider mergeSort, k-way mergeSort and a recursive block matrix-matrix multiply. We will show that the k-way mergeSort and the matrix-matrix multiply analyzed in our model match the best bounds for the ideal-cache. Furthermore the implementations are completely natural functional programs using lists and trees instead of arrays.

Preliminaries

Before describing these algorithms we discuss some general issues and techniques that will be important in the analysis. For simplicity the semantics described in Section 3 only defines natural numbers. In the discussion in this section we assume the semantics have been augmented with some basic types including base types that fit in a machine word (machine integers, floats, and Boolean), sum types, product types, and recursive types made from products and sums. As with the RAM and I/O models, we assume that for an input of size n that machine words can store $O(\log n)$ bits. Hence a machine integer will be bounded by n^k for some constant k.

Since sorting (and matrix multiply) can be defined as higher order polymorphic functions we need to be careful about the size of the element type and cache complexity of the element function when analyzing overall cache complexity. For this purpose we define the notion of a *hereditarily finite* (HF) values. At the base any value of basic types that fit in words are HF. Inductively any sums or products of HF values are HF. A value of function type is HF iff for every HF argument of the domain type the function yields a HF argument of the range type, using only constant space in the process. In sorting we assume the elements themselves and the comparison function are hereditarily finite. In matrix multiply we assume the elements, addition, and multiplication functions are hereditarily finite.

It is important that the data structures that are traversed by our algorithms are laid out in an order that makes accessing them efficient. In our model the only way to control the layout of data structures is to allocate them in the desired order. We could try to define the notion of a list being in a good order in terms of how the list is represented in memory. This is cumbersome and low level. We could also try to define it with respect to the specific code that allocated the data. Again this is cumbersome. Instead we define it directly in terms of the cache complexity of traversing the structure. By traversing we mean going through the structure in a specific order and touching (reading) all data in the structure. Since types such as trees might have many traversal orders, the definition is with respect to a particular order (e.g. pre-order). For this purpose we define the notion of "compact".

DEFINITION 6.1. *A data structure of size n is* compact *with respect to a given traversal order if traversing it in that order has cache complexity $O(n/B)$ in our cost semantics with $M \geq kB$ for some constant k.*

We can now argue that certain code will generate data structures that are compact with respect to a particular order.

To keep data structures compact not only do the top level links need to be accessed in approximately the order they were allocated, but anything that is touched by the algorithm during traversal also needs to be accessed in a similar order. For example, if we are sorting a list it is important that in addition to the "cons cells", the keys are allocated in the list order (or, as we will argue shortly, reverse order is fine). To ensure that the keys are allocated in the appropriate order they need to be copied whenever placing them in a new list. This copy needs to be a deep copy that copies all components. If the keys are machine words then in practice these might be inlined into the cons cells anyway by a compiler—in fact optimizing compilers such as MLton can even inline product and sum types. However, to ensure that objects are copied we will use the copy operation described in Section 3, writing `!a` to indicate copying of `a`.

```
fun traverseL [] = []
  | traverseL h::R  = let val _ = touch h
                      in traverseL R end

fun traverseR [] = []
  | traverseR h::R  = let val R' = traverseR R
                          val _ = touch h
                      in R' end

fun map f [] = []
  | map f h::R  = f(h)::map R

fun map f [] = []
  | map f h::R  = let val R' = map R
                  in f(h)::R' end
```

Figure 4. Traversing a list in two orders, and examples of `map` that use each of the orders.

```
fun mergeSort less [] = []
  | mergeSort less [a] = [a]
  | mergeSort less A =
let
  fun merge(A,B) =
    case (A,B) of
       ([],B) => B
     | (A,[]) => A
     | (Ah::At, Bh::Bt) =>
         if (less(Ah,Bh))
         then !Ah::merge(At,B)
         else !Bh::merge(A,Bt)
  val (L,H) = split A
in
  merge(mergeSort(L),mergeSort(H))
end
```

Figure 5. Code for `mergeSort`.

Although it might seem that there is only one "canonical" way to traverse a list, there are actually two. In the first all the elements of the list are visited on the way down the recursion, and in the second the elements are visited on the way back up. The order in which the elements are visited is reversed. The two versions are illustrated by the code in Figure 4. Fortunately if an algorithm is compact for one traversal it is compact for the other. This is because to be compact under either traversal requires that adjacent elements are allocated in the same block (neighborhoods in memory), and hence will be efficient in both directions. The fact that the orders are effectively equivalent is important since it means the model is quite robust relative to programming styles. For example the two implementations of the `map` function shown in Figure 4 will both be efficient if the list is compact with respect to either traversal. Furthermore the output is compact with respect to either traversal. This is true even though in the first case all elements are allocated first and then all cons cells, while in the second case they are interleaved.

Sorting

We now consider analyzing sorting in our cost model. We first consider mergeSort. We assume a vanilla purely functional version of mergeSort on lists as shown in Figure 5. We will not cover the definition of `split` since it is similar to merge. Note the only difference from a standard mergeSort is the use of the copy (`!`)

before the `Ah` and `Bh`. As discussed earlier this is important to ensure the result of the merge is compact. We note that for a list to be compact all its elements must be constant size. The interesting aspect of the code is that the cache complexity of this code in our model and hence when mapped onto the ideal cache using the implementation in section 5 matches the bounds for the array version discussed in Section 2.

To analyze the mergeSort we first analyze the merge.

THEOREM 6.2. *For a HF function* `less`*, and compact lists* `A` *and* `B`*, the evaluation of* `(merge less A B)` *starting with any cache state will have cache complexity* $O\left(\frac{n}{B}\right)$ *(*$n = |A| + |B|$*) and will return a compact list as a result.*

Proof. We consider the cache complexity of going down the recursion and then coming back up. Since A and B are both compact we need only put aside a constant number of cache blocks to traverse each one (by definition). Recall that in the cost model we have a nursery ν that maintains both live allocated values and place holders for stack frames in the order they are created. In `merge` nothing is allocated from one recursive call to the next (the cons cells are created on the way back up the recursion) so only the stack frames are placed in the nursery. After M recursive calls the nursery will fill and blocks will have to be flushed to the memory μ (rule 5b in Figure 2). The merge will invoke at most $O(n/B)$ such flushes since only n frames are created. On the way back up the recursion we will generate the cons cells for the list and copy each of the keys (using the `!`). Note that copying the keys is important so that the result remains compact. The cons cells and copies of the keys will be interleaved in the allocation order in the nursery and flushed to memory once the nursery fills. Once again these will be flushed in blocks of size B and hence there will be at most $O(n/B)$ such flushes. Furthermore the resulting list will be compact since adjacent elements of the list will be in the same block (neighborhood). □

We now consider mergeSort as a whole.

THEOREM 6.3. *For a HF function* `less`*, and compact list* `A`*, the evaluation of* `(mergeSort less A)` *starting with any cache state will have cache complexity* $O\left(\frac{n}{B}\log\frac{n}{M}\right)$ *(*$n = |A|$*) and will return a compact list as a result.*

Proof. As with the array version, we consider the two cases when the input fits in cache and when it does not. The mergeSort routine never requires more than $O(n)$ live allocated data. Therefore when $kn \leq M$ for some (small) constant k all allocated data fits in the nursery. Furthermore since the input list is compact, for $k'n \leq M$ the input fits in the read cache (for some constant k'). Therefore the cache complexity for mergeSort is at most the time to flush $O(n)$ items out of the allocation cache that it might have contained at the start, and to load the read cache with the input. This cache complexity is bounded by $O(n/B)$. When the input does not fit in cache we have to pay for the merge as analyzed above plus the recursive calls. This gives the same recurrence as for the array version (Section 2, equation 1) and hence solves to the claimed result. □

We now consider a k-way mergeSort using lists and a tree based heap for the k-way merging. The code is shown in Figure 6. The sort partitions the list into k parts, sorts each part recursively, builds a priority queue (PQ) out of the resulting parts, and pulls keys one by one out of the PQ adding them to the output list. The only slightly tricky part is maintaining the priority queue. The idea is each of the sorted lists is placed at a leaf. When pulling elements from the root of the PQ, the value is removed from the root and the two children are joined, which recursively pulls the value from the child with the smaller root. We don't show the code for `partition`, which simply partitions a list into k equal length parts (within 1). Although the code is somewhat involved the analysis of the cache complexity is relatively simple since most of the data allocated for the tree-based priority queue becomes unreachable before it needs to be flushed to memory.

```
datatype 'a pq = Leaf of 'a list
               | Node of 'a * 'a pq * 'a pq

fun kWayMergeSort _ _ [] = []
  | kWayMergeSort _ _ [a] = [a]
  | kWayMergeSort less k L =
let
  fun getMin Leaf [] = NONE
    | getMin Leaf (a::_) = SOME(a)
    | getMin Node (a,_,_) = SOME(a)

  fun join(L,R) =
    case (getMin(L),getMin(R)) of
        (NONE,_) => R
      | (_,NONE) => L
      | (la,ra) =>
          if less(la,ra)
          then Node(la,delMin(L),R)
          else Node(lb,L,delMin(R))

  and delMin (Leaf (_::R)) = Leaf(R)
    | delMin (Node (_,L,R)) = join(L,R)

  fun merge H =
    case getMin(H) of
       NONE => []
     | SOME(a) => let val r = merge(delMin(H))
                  in !a::r
                  end

  fun buildPQ [a] = Leaf(a)
      buildPQ A =
        let val (L,H) = partition 2 A
        in  join(buildPQ(L),buildPQ(R))
        end

  val LL = partition k L
  val SL = map (kWayMergeSort less k) LL
  val HL = buildPQ SL
in
  merge HL
end
```

Figure 6. K-way Merge Sort.

THEOREM 6.4. *For a HF function* `less`*, and compact list* `A`*, the evaluation of* `(kWayMergeSort less k A)` *starting with any cache state and with an appropriate* k *(*$\approx M/B$*) will have cache complexity* $O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ *(*$n = |A|$*) and will return a compact list as a result.*

Proof. As usual once the input size is less than n/c for some constant, the whole problem fits in cache and we just pay to load the input and write the output, which will have $O(n/B)$ cache complexity if the input and output are compact. When the problem size does not fit in cache we note that with $k = \frac{1}{c}M/B$ for some constant c we can fit the head of each recursively solved list in the read cache, assuming each is compact. Therefore traversing all lists will use $O(n/B)$ cache complexity. Furthermore the size of the priority queue is proportional to k so the live part easily fits within the allocation cache. We have to be careful, however, since the

```
datatype 'a M = Leaf of 'a
              | Node of 'a M * 'a M * 'a M * 'a M

fun mmult + * (Leaf a) (Leaf b) = Leaf(*(a,b))
  | mmult + * (Node(a1,a2,a3,a4)) (Node(b1,b2,b3,b4)) =
let
  fun madd (Leaf a) (Leaf b) = Leaf(+(a,b))
    | madd (Node(a1,a2,a3,a4)) (Node(b1,b2,b3,b4))
      Node(madd(a1,b1),madd(a2,b2),
           madd(a3,b3),madd(a4,b4))

  val mm = mmult + *
in
  Node(madd(mm(a1,b1),mm(a2,b3)),
       madd(mm(a1,b2),mm(a2,b4)),
       madd(mm(a3,b1),mm(a4,b3)),
       madd(mm(a3,b2),mm(a4,b4)))
end
```

Figure 7. Matrix Multiply.

allocation cache is shared with the stack frames, which could eject some of the data allocated by the PQ. But since at any given time much less than half of the allocation cache (only $k = O(M/B)$ of it) is used by the PQ, we can charge all such ejections against the ejected cache frames (we charge for every cache frame). We can also charge reading them back into read cache against the cache frames.

Going down the recursion of the merge therefore requires $O(n/B)$ cache complexity to account for loading the k recursively solved lists, and ejecting the n cache frames. Coming back up the recursion again requires $O(n/B)$ cache complexity for ejecting the list and the copied keys. The resulting list is compact for list traversal since it is allocated in list traversal order (tail of the list first). This gives us the recurrence in equation 2 from Section 2 which solves to the desired result. □

Matrix Multiply

Our final example is matrix multiply. The code is shown in figure 7 (we have left out checks for matching sizes). This is a block recursive matrix multiply with the matrix laid out in a tree. It is therefore an interesting example of a tree data structure. We define compactness with respect to a preorder traversal of this tree. We therefore say the matrix is compact if traversing in this order can be done with cache complexity $O(n^2/B)$ for an $n \times n$ matrix (n^2 leaves). We note that if we generate a matrix in a preorder traversal allocating the leaves along the way, the resulting array will be compact.

THEOREM 6.5. *For HF functions* `*` *and* `+`*, and compact* $n \times n$ *matrices* `A` *and* `B`*, the evaluation of* `(mmult + * A B)` *starting with any cache state will have cache complexity* $O\left(\frac{n^3}{B\sqrt{M}}\right)$ *and will return a compact matrix as a result.*

Proof. Matrix addition has cache complexity $O(n^2/B)$ and generates a compact result since we traverse the two input matrices in preorder traversal and we generate the output in the same order. Since the live data is never larger than $O(n^2)$ the problem will fit in cache for $n^2 \leq M/c$ for some constant c. Once it fits in cache the cost is $O(n^2/B)$ needed to the load the input matrices and write out the result. When it does not fit in cache we have to do 8 recursive calls and four calls to matrix addition. This gives the recurrence.

$$Q(n) = \begin{cases} 8Q(\frac{n}{2}) + O(\frac{n^2}{B}) & n^2 > M/c \\ O(\frac{n^2}{B}) & \text{otherwise} \end{cases}$$

This solves to $O\left(\frac{n^3}{B\sqrt{M}}\right)$. The output is compact since each of the four calls to `madd` in `mmult` allocate new results in preorder with respect to the submatrices they generate, and the four calls are made in preorder. Therefore the overall matrix returned is allocated in preorder. □

7. Conclusion

The idea of distinguishing the abstract cost semantics of language from its concrete implementation originates with Blelloch and Greiner's work on parallel programming [5, 11]. The chief benefit of their approach is that it provides a useful abstraction to the programmer that accounts for the complexity of a program, while simultaneously providing a guide to the implementor for how to achieve the complexity bound (with stated overhead). This work extends that methodology to account for the I/O complexity of a program in terms of two parameters, the cache block size (measured in objects) and the number of cache blocks. The programmer reasons at the level of the evaluation semantics, the implementor makes use of the provable implementation strategy to realize the predicted complexity. In the present case the essence of the proof is to argue that conventional implementation techniques, which rely on a run-time control stack and copying garbage collection, can be deployed to meet the abstract bounds given by the semantics of a functional language. The separation between the semantics and its implementations allows the programmer to work at the level of the code itself, and avoids having to reason in terms of the details of the compiler and run-time system (or, even worse, to be forced to drop down to a C-like level in which the programmer explicit manages storage allocation for each application).

Using the approach we are able to express algorithms in a standard high-level functional style using recursive data types (lists and trees), analyze them using a model that captures the idea of a fixed size read and allocation stack, but no details of the run time system, and yet match the asymptotic bounds for the ideal cache achieved by designing them using arrays and explicit and careful memory management in the imperative setting. For sorting the bounds are optimal.

One direction for further research is to integrate (deterministic) parallelism with the present work. Based on previous work we expect that the evaluation semantics given here will provide a good foundation for specifying parallel as well as sequential complexity. One complication is that the explicit consideration of storage considerations in the cost model given here would have to take account of the interaction among parallel threads. The amortization arguments would also have to be reconsidered to account for parallelism.

Another direction is suggested by the special treatment of the run-time stack described in Section 4. The stack is, after all, a particular data structure that is used implicitly by each program. This use could be made explicit, in which case it would be useful to understand more generally what properties of it allow for its efficient (in terms of cache complexity) implementation. These might well generalize to other data structures, and we it may be useful to develop a type system to capture these special properties.

Finally, although we are able to generate an optimal cache-aware sorting algorithm it is unclear whether it is possible to generate an optimal cache-oblivious sorting algorithm in our model.

Acknowledgments. This work is partially supported by the National Science Foundation under grant number CCF-1018188, and by Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program.

References

[1] J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] A. W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, 1987.

[4] L. Arge, M. A. Bender, E. D. Demaine, C. E. Leiserson, and K. Mehlhorn, editors. *Cache-Oblivious and Cache-Aware Algorithms, 18.07. - 23.07.2004*, volume 04301 of *Dagstuhl Seminar Proceedings*, 2005. IBFI, Schloss Dagstuhl, Germany.

[5] G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *FPCA*, pages 226–237, 1995.

[6] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In K. L. Clarkson, editor, *SODA*, pages 139–149. ACM/SIAM, 1995. ISBN 0-89871-349-8.

[7] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In S. L. P. Jones and R. E. Jones, editors, *ISMM*, pages 37–48. ACM, 1998. ISBN 1-58113-114-3.

[8] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Commun. ACM*, 31(9):1128–1138, 1988.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298. IEEE Computer Society, 1999.

[10] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry (preliminary version). In *FOCS*, pages 714–723. IEEE Computer Society, 1993.

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

[12] D. Grunwald, B. G. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In R. Cartwright, editor, *PLDI*, pages 177–186. ACM, 1993. ISBN 0-89791-598-4.

[13] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013. (Draft available at `http://www.cs.cmu.edu/~rwh/plbook/book.pdf`.).

[14] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[15] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-00883-7.

[16] J. G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA*, pages 66–77, 1995.

[17] K. Munagala and A. G. Ranade. I/o-complexity of graph algorithms. In R. E. Tarjan and T. Warnow, editors, *SODA*, pages 687–694. ACM/SIAM, 1999. ISBN 0-89871-434-6.

[18] G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[19] M. Rahn, P. Sanders, and J. Singler. Scalable distributed-memory external sorting. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 685–688. IEEE, 2010. ISBN 978-1-4244-5444-0.

[20] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.

[21] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In J. Hook and P. Thiemann, editors, *ICFP*, pages 253–264. ACM, 2008. ISBN 978-1-59593-919-7.

[22] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[23] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *LISP and Functional Programming*, pages 32–42, 1992.