Cache Efficient Functional Algorithms

Guy E. Blelloch
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15232
guyb@cs.cmu.edu

Robert Harper
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15232
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 fast memory (cache) of size M, and an unbounded slow 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.

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 bounds imply that purely functional programs based on lists and trees with no special attention to any details of memory layout can be 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.

1. INTRODUCTION

Today's computers exhibit 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 drive (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.

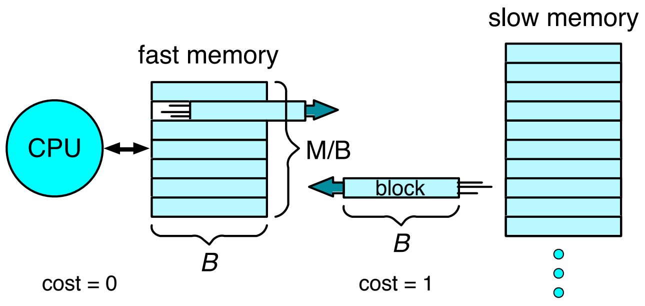

Figure 1: The *I/O model* [2] assumes a memory hierarchy comprising a fast memory of size M and slow memory of unbounded size. Both memories are partitioned into blocks of a fixed size B of consecutive memory locations. The CPU and fast memory are treated as a standard Random Access Machine (RAM)—the CPU accesses individual words, not blocks. Additional instructions allow one to move any block between the fast and slow memory. 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. The *ideal-cache machine model* (ICMM) [11] is similar except that the program only addresses the slow memory, and an "ideal" cache decides which blocks to cache in the fast memory using an optimal replacement strategy. Accessing the fast memory (cache) is again regarded as cost-free and transferring between slow memory and fast memory has unit cost. The costs in the two models are equivalent within a constant factor, and in both models the two levels of memory can either represent main memory and disk, or cache and main memory.

Figure 1 describes the the widely used I/O [2] and ideal-cache [11] machine models designed for this purpose. The models are based on two parameters, the memory size M and the block size B, which are considered variables for the sake of analysis and therefore show up in asymptotic bounds. To design efficient algorithms for these models, it is important to consider both temporal locality, so as to take advantage of the limited size slow memory, and spatial locality, because memory is transferred in contiguous blocks. In this paper we will use terminology from the ideal-cache machine

```
fun mergeSort([]) = []
  | mergeSort([$a$]) = [a]
  | mergeSort($A$) =
let
  val $(L, H)$ = split($A$)
in
  merge(mergeSort($L$),mergeSort($H$))
end
```

Figure 2: Recursive `mergeSort`. If the input sequence is empty or a singleton the algorithm returns the same value, otherwise it splits the input in two approximately equal sized sequences, `L` and `H`, recursively sorts each sequence, and merges the result.

model (**ICMM**), including referring to the fast memory as **cache**, the slow memory as **main memory**, block transfers as **cache misses**, and the cost as the **cache cost**.

Algorithms that do well in these models are often referred to as cache or I/O efficient. The theory of cache efficient algorithms is now well developed (see, for example, the surveys [17, 3, 23, 6, 19, 12]). These models do indeed express more accurately the cost of algorithms on real machines than does the standard RAM model, for example. For example, the models properly indicate that a blocked or hierarchical matrix-matrix multiply has cost

$$\text{mat. mult. cache cost} \quad = \quad \Theta\left(\frac{n^3}{B\sqrt{M}}\right). \qquad (1)$$

This is much more efficient than the naïve triply nested loop, which has cost $\Theta(n^3/B)$. Furthermore, although the models only consider two levels of a memory hierarchy, cache efficient algorithms that do not use the parameters B or M in the code are simultaneously efficient across all levels of the memory hierarchy. Algorithms designed in this way are called *cache oblivious* [11].

As an example of an algorithm analyzed in the models consider recursive mergeSort (see Figure 2). If the inputs of size n and output are in arrays, then the `merge` can take advantage of spatial locality and has cache cost $O(n/B)$ (see Figure 3). The `split` can be done within the same, or better bounds. For the recursive mergeSort once the recursion reaches a level where the computation fits into the cache, all subcalls are free, taking advantage of temporal locality. Figure 4 illustrates the merge sort recursion tree and analyzes the total cost, which for an input of size n is:

$$\text{merge sort cache cost} \quad = \quad \Theta\left(\left(\frac{n}{B}\right)\log_2\left(\frac{n}{M}\right)\right) \qquad (2)$$

This is a reasonable cost and a lot better than assuming every memory access requires a cache miss. However, this is not optimal for the model. Later, we will discuss a multiway merge sort that is optimal. It has cost

$$\text{optimal sort cache cost} \quad = \quad \Theta\left(\left(\frac{n}{B}\right)\log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right) \qquad (3)$$

This optimal cost can be significantly less than the cost for mergeSort. For example if $n \le M^k$ and $M \ge B^2$, which are reasonable assumptions in many situations, then the cost reduces to $O(nk/B)$. All the fastest disk sorts indeed use some variant of a multiway merge sort or another optimal cache efficient algorithm based on sample sorting [21].

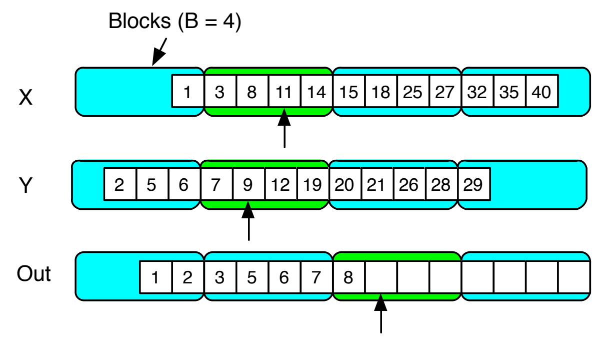

Figure 3: Merging two sorted arrays. The fingers are moved from left to right along the two inputs and the output. At each point the lesser value at the input fingers is copied to the output, and that finger, along with the output finger, are incremented. During the merge, each block of the inputs only has to be loaded into the fast memory once, and each block of the output only needs to be written to slow memory once. Therefore, for two arrays of size n the total cache cost of the merge is $O(n/B)$.

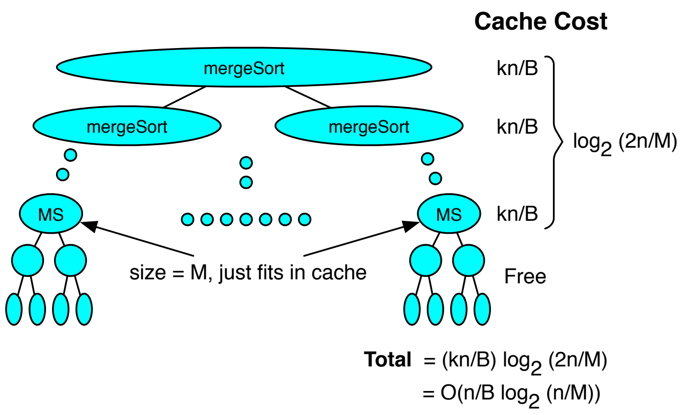

Figure 4: The cost of mergeSort in the I/O or ideal-cache model. At the i^{th} level from the top there are 2^i calls to mergeSort, each of size $n_i = n/2^i$. Each call has cost kn_i/B for some constant k. The total cost of each level is therefore about $2^i k\left(\frac{n}{2^i}\right)/B = kn/B$. When a call fits into the fast memory, then the rest of the sort is cost-free. Because a merge requires about $2n$ space (for the input and the output), this will occur when $M \approx 2(\frac{n}{2^i})$. Therefore, there are $\log_2(2n/M)$ levels that have non-zero cost, each with cost kn/B.

Careful Layout and Careful Allocation.

Although the analysis for mergeSort given above seems relatively straightforward, we have picked a simple example and left out several important details. With regards to memory layout, ordering a sorted sequence contiguously in memory is pretty straightforward, but this is not the case for many other data structures. Should a matrix, for example, be laid out in row-major order, in column-major order, or perhaps in some hierarchical order such as z-ordering? These different orders can make a big difference. What

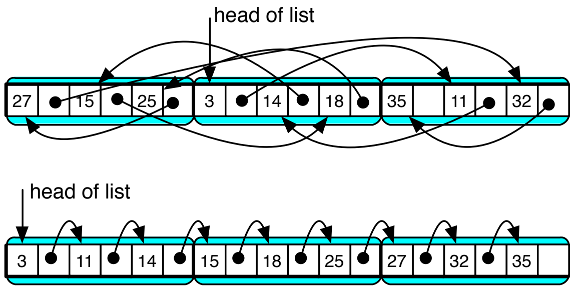

Figure 5: Two lists in memory. The cost of traversing the lists is very different depending on how the lists are laid out. In the figure, the top one, which is randomly laid out, will require $O(n)$ steps to traverse, whereas the bottom only requires $O(n/B)$.

about pointer-based structures such as linked-lists or trees? As Figure 5 indicates, the cost of traversing a linked list will depend on how the lists are laid out in memory.

More subtle than memory layout is memory allocation. Touching unused memory after allocation will cause a cache miss. Managing the pool of free memory properly is therefore very important. Consider, again, the mergeSort example. In our discussion we assumed that once $2n_i \leq M$ the problem fits in fast memory and therefore is free. However, if we used a memory allocator to allocate the temporary array needed for the merge the locations would likely be fresh and accessing them would cause a cache miss. The cost would therefore not be free, and in fact because levels near the leaves of the recursion tree do many small allocations, the cost could be significantly larger near the leaves. To avoid this, programmers need to manage their own memory. They could, for example, preallocate a temporary array and then pass it down to the recursive calls as an extra argument, therefore making sure that all temporary space is being reused. For more involved algorithms, allocation can be much more complicated.

In summary, designing and programming cache efficient algorithms for these models requires a careful layout of data in the flat memory and careful management of space in that memory.

Our Goals.

The goal of our work is to be able to take advantage of the ideal-cache model, and hence machines that match the model, without requiring careful layout of data in memory and without requiring managing memory allocation by hand. In particular we want to work with recursive data structures (pointer structures) such as lists or trees. Furthermore, we want to work with a fully automated memory manager with garbage collection and no (explicit) specifications of where data is placed in memory. These goals might seem impossible, but we show that they can be achieved, at least for certain cases. We show this in the context of a purely functional (side-effect free) language. The formulation given here is slightly simplified compared to the associated conference paper [5] for the sake of brevity and clarity; we refer the reader to that paper for a fuller explanation.

As an example of what we are trying to achieve consider the `merge` algorithm shown in Figure 6. Some properties to notice about the algorithm are that it is based on linked-lists

```
 1  datatype List = Cons of (int * List)
 2                 | Nil
 3
 4  fun merge(A, B) =
 5    case (A, B) of
 6        (Nil, B) ⇒ B
 7      | (A, Nil) ⇒ A
 8      | (Cons(a, At), Cons(b, Bt)) ⇒
 9            if (a < b)
10            then Cons(!a, merge(At, B))
11            else Cons(!b, merge(A, Bt))
```

Figure 6: Code for `merge` of two lists. If list is empty, the code compares the two heads of the list and recurs on the tail of the list with the smaller element, and on the whole list with the larger element. When the recursive call finishes a new list element is created with the smaller element as the head and the result of the recursive call as the tail. The `!` is discussed in section 3.

instead of arrays. Furthermore, because we are working in the functional setting, every list element is created fresh (in the `Cons` operations in Lines 10 and 11.) Finally there is no specification of where things are allocated and no explicit memory management. Given these properties it might seem that this code could be terrible for cache efficiency because accessing each element could be very expensive, as indicated in Figure 5. Furthermore, each fresh allocation could cause a cache miss. Also there is an implied stack in the recursion and it is not clear how this affects the cost.

We show, however, that with an appropriate implementation the code is indeed cache efficient, and when used in a mergeSort gives the same bound as given in Equation 2. We also describe an algorithm that matches the optimal sorting bounds given in Equation 3. The approach is not limited to lists, it also works with trees. For example Figure 11 defines a representation of matrices recursively divided into a tree and a matrix multiplication based on it. The cost bounds for this multiplication match the bounds for matrix multiplication for the ideal-cache model (Equation 1) using arrays and careful programmer layout.

Cost Models and Provable Implementation Bounds.

One way to achieve our goals would be to analyze each algorithm written in high-level code with respect to a particular compilation strategy and specific runtime memory allocation scheme. This would be extremely tedious, time consuming, not portable, and error prone. It would not lead to a practical way to analyze algorithms.

Instead we use a cost semantics that directly defines costs in the programming language so that algorithms can be analyzed without concern of implementation details (such as how the garbage collector works). We refer to our language, equipped with our cost semantics, as **ICFP**, for *Ideal-Cache Functional Programming* language. The goal is then to relate the costs in **ICFP** to costs in the **ICMM**. We do this by describing a simulation of **ICFP** on the **ICMM** and proving bounds on the relative costs based on this simulation. In particular, we show that a computation running with cost Q, with parameters M and B, in **ICFP** can be simulated

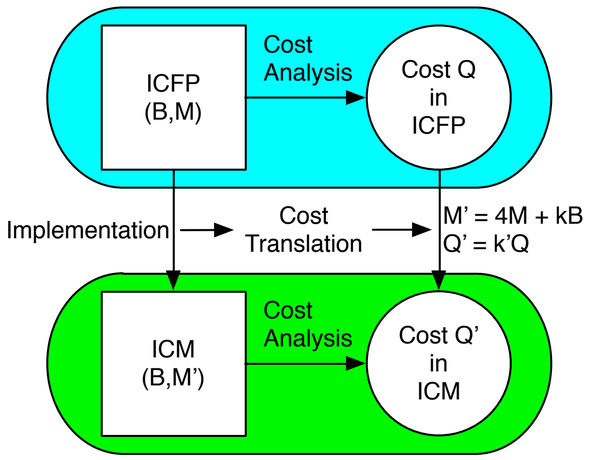

Figure 7: Cost semantics for ICFP and its mapping to the ideal cache model (ICMM). The values k and k' are constants.

on the **ICMM** with $M' = O(M)$ locations of fast memory, blocksize B, and cost $O(Q)$. The framework is illustrated in Figure 7.

Allocation Order and Spatial Locality.

Because functional languages abstract away from data layout decisions, we use temporal locality (proximity of allocation time) as a proxy for spatial locality (proximity of allocation in memory). To do this, the memory model for **ICFP** consists of two fast memories: a nursery (for new allocations) and a read cache. The nursery is time ordered to ensure that items are laid out contiguously when written to main (slow) memory. For merging, for example, the output list elements will be placed one after the other in main memory. To formalize this idea we introduce the notion of a data structure being compact. Roughly speaking a data structure of size n is compact if it can be traversed in the model in $O(n/B)$ cache cost. The output of merge is compact by construction. Furthermore, if both inputs lists to merge are compact, we show that merge has cache cost $O(n/B)$. This notion of compactness generalizes to other data structures, such as trees.

Another feature of the allocation cache of **ICFP** is that it only contains live data. This is important because in some algorithms, such as matrix multiply, the rate at which temporary memory is generated is asymptotically larger than the rate at which output data is generated. Therefore, if all counted it could fill up the cache, whereas we want to make sure that temporary data slots are reused. Also, we want to ensure that temporary memory is not transferred to the slow memory because such transfer would cause extra traffic. Moreover, the temporary data could end up between output data elements, causing them to be fragmented across blocks. In our provable implementation we do not require a precise tracking of live data, but instead use a generational collector with a nursery of size $2M$. The simulation then keeps all the $2M$ locations in the fast memory so that all allocation and collection (within the new generation) is fast.

Related Work.

The general idea of using high-level cost models based on a cost semantics along with a provable efficient implementation has previously been used in the context of parallel cost models [4, 13, 22, 15]. 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 ([10, 24, 14, 7] and many references in [16]), 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. However, they 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. COST SEMANTICS

In general a *cost semantics* for a language defines both *how to execute* a program and the *cost* of its execution. In functional languages execution consists of a deterministic strategy for calculating the value of an expression by a process of simplification, or *reduction*, similar to what one learns in elementary algebra. But rather than working with the real numbers, programs in a functional language calculate with integers, with inductive data structures, such as lists and trees, and with functions that act on such values, including other functions. The theoretical foundation for functional languages is Church's λ-calculus [8, 9].

The cost of a computation in a functional language may be defined in various ways, including familiar measures such as time complexity, defined as the number of primitive reduction steps required to evaluate an expression, and space complexity, defined as the number of basic data objects allocated during evaluation. Such measures are defined in terms of the constructs of the language itself, rather than in terms of its implementation on a machine model such as a RAM or Turing machine. Algorithm analysis takes place entirely at the language level. The role of the implementation is to ensure that the abstract costs can be realized on a concrete machine with stated bounds.

Cache Cost.

The cost measure of interest in the present paper is the *cache cost*, which is derived from considering a combination of the time and space behavior of a program. The cache cost is derived by specifying when data objects are allocated and retrieved, and as with the **ICMM** is parameterized by two constants, B and M, governing the cache behavior. To be more precise, expressions are evaluated relative to an abstract *store*, σ, which comprises a *main memory*, μ, a *read cache*, ρ, and a *nursery*, or *allocation area*, ν. The structure of the store is depicted in Figure 8. The *main memory* maps abstract locations to atomic data objects, and is of unbounded capacity. Atomic data objects are considered to occupy one unit of storage. Objects in main memory are aggregated into *blocks* of size B in a manner to be described shortly. The blocking of data objects does not change during execution; once an object is assigned to a block, it remains part of that block for the rest of the computation.

The *read cache* is a fixed-size mapping from locations to

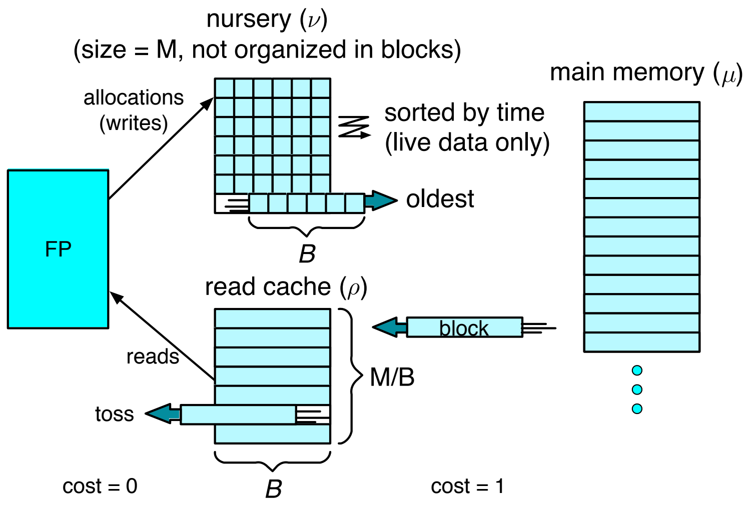

Figure 8: The memory model for the ICFP cost semantics. It is similar to the ideal cache model, but has two fast memories of size M: a *nursery* for newly allocated data, and a *read cache* for data that has migrated to main memory and was read again by the program. The main memory and read cache are organized into blocks, as in the ideal-cache model, but the nursery is not blocked. Rather, it consists of (live) data objects linearly ordered by the time at which they were allocated. The semantics defines when a piece of data is live, but conceptually it is just when it is reachable from the executing program. If an allocation overflows the nursery, the oldest B live objects are flushed from the cache into main memory as a block to make space for the new object. If one of these words is later read, the entire block is moved into the read cache, possibly displacing another block in the process. Displaced blocks can be discarded, because the only writes in a functional language occur at allocation, and never during subsequent execution.

data objects containing at most M objects. As usual, the read cache represents a partial view of main memory. Objects are loaded into the read cache in blocks of size B as determined by the aggregation of objects in main memory. Objects are evicted from the read cache by simply overwriting them; in a functional language data objects cannot be mutated, and hence need never be written back to main memory. Blocks are evicted according to the (uncomputable) *Ideal Cache Model (*ICMM*)*.

The *allocation area*, or *nursery*, is fixed-sized mapping from locations to data objects also containing at most M objects. Locations in the nursery are not blocked, but they are linearly ordered according to the time at which they are allocated. We say that one data object is *older* than another if the location of the one is ordered prior to that of the other. A location in the nursery is *live* if it occurs within the program being evaluated [18]. All data objects are allocated in the nursery. When its capacity of M objects is exceeded, the oldest B objects are formed into a block that is migrated to main memory, determining once and for all the block to which these objects belong. This policy ensures that *temporal locality implies spatial locality*, which means that objects that are allocated nearby in time will be situated nearby in space (that is, occupy the same block). In particular, when an object is loaded into the read cache, its neighbors in its block will be loaded along with it. These will be the objects that were allocated at around the same time.

The role of a cost semantics is to enable reasoning about the cache cost of algorithms without dropping down to the implementation level. To support proof, the semantics must be specified in a mathematically rigorous manner, which we now illustrate for ICFP, a simple eager variant of Plotkin's language PCF [20] for higher-order computation over natural numbers. (It is straightforward to extend ICFP to account for a richer variety of data structures, such as lists or trees, and for hardware-oriented concepts such as words and floating point numbers, which we assume in our examples.)

Cost Semantics.

The abstract syntax of ICFP is defined as follows:

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

Here x stands for a variable, which will always stand for a data object allocated in memory. The expressions $\mathtt{z}$ and $\mathtt{s}()$ represent 0 and successor, respectively, and the conditional tests whether e is $\mathtt{z}$ or not, branching to e_0 if so, and branching to e_1 if not, substituting (the location of) the predecessor for x in doing so. Functions, written $\mathtt{fun}(x, y.e)$, bind a variable x standing for the function itself, and a variable y standing for its argument. (The variable x is used to effect recursive calls.) Function application is written $\mathtt{app}(e_1; e_2)$, which applies the function given by the expression e_1 to the argument value given by the expression e_2. The typing rules for ICFP are standard (see Chapter 10 of [15]), and hence are omitted for the sake of concision.

Were the cost of a computation in ICFP just the time complexity, the semantics would be defined by an evaluation relation $e \Downarrow^n v$ stating that the expression e evaluates to the value v using n reduction steps according to a specified outermost strategy. Accounting for the cache cost of a computation is a bit more complicated, because we must account for the memory traffic induced by execution. To do

$$\frac{\left\{\begin{array}{rl} \sigma \,@\, e_1 \Downarrow^{n_1} \sigma_1' \,@\, l_1' & \text{(evaluate function)} \\ \sigma_1' \,@\, l_1' \downarrow^{n_1'} \sigma_2 \,@\, \mathtt{fun}(x, y.e) & \text{(load function value)} \\ \sigma_2 \,@\, e_2 \Downarrow^{n_2} \sigma_2' \,@\, l_2' & \text{(evaluate argument)} \\ \sigma_2' \,@\, [l_1', l_2'/x, y]e \Downarrow^{n} \sigma' \,@\, l & \text{(evaluate function body)} \end{array}\right\}}{\sigma \,@\, \mathtt{app}(e_1; e_2) \Downarrow^{n_1+n_1'+n_2+n_2'} \sigma' \,@\, l'}$$

Figure 9: Simplified semantics of function application. This rule specifies the cost and evaluation of a function application $\mathtt{app}(e_1; e_2)$ of a function e_1 to an argument e_2. similar rules govern the other language constructs. Bear in mind that all values are represented by (abstract) locations in memory. First, the expression e_1 is evaluated, with cost n_1, to obtain the location l_1 of the function being called. That location is read from memory to obtain the function itself, a self-referential λ-abstraction, at cost n_1'. Second, the expression e_2 is evaluated, with cost n_2, to obtain its value l_2. Finally, l_1 and l_2 are substituted into the body of the function, and then evaluated to obtain the result l at cost n. The overall result is l with total cost the sum of the constituent costs. The only non-zero cost arises from allocation of objects and reading from memory; all other operations are considered cost-free, as in the I/O model.

so, we consider an evaluation relation of the form

$$\sigma \,@\, e \Downarrow^{n} \sigma' \,@\, l$$

in which the evaluation of an expression e is performed relative to a store σ, and returns a value represented by a location in a modified store σ'. The modifications to the store reflect the allocation of new objects, their migration to main memory, and their loading into the read cache. The evaluation relation also specifies the cache cost n, which is determined entirely by the movements of blocks to and from main memory. All other operations are assigned zero cost.

The evaluation relation for **ICFP** is defined by *derivation rules* that specify closure conditions on it. Evaluation is defined to be the strongest relation satisfying these conditions. To give a flavor of how this is done, we present in Figure 9 a simplified form of the rule for evaluating a function application, $\mathtt{app}(e_1; e_2)$. The rule has four premises, and one conclusion, separated by the horizontal line, which may be read as implication stating that if the premises are all derivable, then so is the conclusion.

The other constructs of **ICFP** are defined by similar rules, which altogether specify the behavior and cost of the evaluation of any **ICFP** program. The abstract cost semantics underlies the analyses given in Sections 1 and 3. The abstract costs given by the semantics are validated by giving a *provable implementation* [4, 13] that realizes these costs on a concrete machine. In our case the realization is given in terms of the **ICMM**, which is an accepted concrete formulation of a machine with hierarchical memory. By composing the abstract semantics with the provable implementation we obtain end-to-end bounds on the cache complexity of algorithms written in **ICFP** without having to perform the analysis at the level of the concrete machine, but rather at the level of the language in which the algorithms are written.

Provable Implementation.

The provable implementation of **ICFP** on the **ICMM** is described in Figure 10. Its main properties, which are important for obtaining end-to-end bounds on cache complexity, are summarized by the following theorem:

THEOREM 2.1. *An evaluation $\sigma \,@\, e \Downarrow^{n} \sigma' \,@\, l$ in* **ICFP** *with abstract cache parameters M and B can be implemented on the* **ICMM** *with concrete cache parameters $4 \times M + k \times B$ and B with cache cost $\mathcal{O}(n)$, for some constant k.*

The proof of the theorem consists of an implementation of **ICFP** on the **ICMM** described in Figure 10, and of showing that reduction step of **ICFP** is simulated on the **ICMM** to within a (small) constant factor. The full details of the proof, along with a tighter statement of the theorem, are given in the associated conference paper.

3. CACHE EFFICIENT ALGORITHMS

We now analyze mergeSort and matrix multiply in **ICFP**, and give bounds for a k-way mergeSort that is optimal. The implementations are completely natural functional programs using lists and trees instead of arrays. In the associated conference paper we describe how one can deal with higher-order functions (such as passing a comparison function to merge), and define the notion of hereditarily finite values for this purpose, but here we only consider first-order usage.

As mentioned in the introduction, the cost of accessing a recursive datatype will depend on how it is laid out in memory, which depends on allocation 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 cost of traversing the structure. By traversing we mean going through the structure in a specific order and touching (reading) all data in the structure. Because types such as trees might have many traversal orders, the definition is with respect to a particular traversal order. For this purpose we define the notion of "compact".

DEFINITION 3.1. *A data structure of size n is* compact *with respect to a given traversal order if traversing it in that order has cache cost $O(n/B)$ 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. We note that if a list is compact with respect to traversal in one direction, it is also compact with respect to the opposite direction.

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 com-

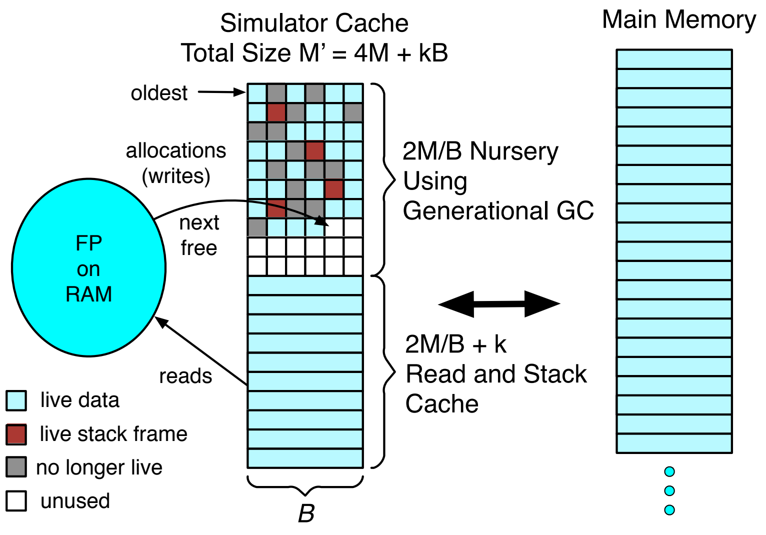

Figure 10: Simulating the ICFP on the ICMM. The simulation requires a cache of size $4 \times M + k \times B$ for some constant k. Allocation and migration is managed using generational copying garbage collection, which requires $2 \times M$ words of cache to manage M live objects. The liveness of objects in the nursery can be assessed without accessing main memory, because in a functional language older objects cannot refer to newer objects. Copying collecton preserves the allocation ordering of objects, as is required for our analyses. Because two objects that are in the same abstract block can be separated into adjacent blocks by the simulation, blocks are loaded two-at-a-time, requiring $2 \times M$ additional words of cache on the ICMM. An additional B words of read cache are required to account for the space required by the control stack using an amortization argument, and a constant number of blocks are required for the allocation of the program itself in memory.

piler. However, to ensure that objects are copied we will use operation `!a` to indicate copying of `a`.

Merge Sort.

We now consider analyzing mergeSort in our cost model, and in particular the code given in Figures 2 and 6. We will not cover the definition of `split` because it is similar to merge. We first analyze the merge.

THEOREM 3.2. *For compact lists* `A` *and* `B`*, the evaluation of* `merge(A,B)` *starting with any cache state will have cache cost* $O\left(\frac{n}{B}\right)$ *($n = |A| + |B|$) and will return a compact list.*

PROOF. We consider the cache cost of going down the recursion and then coming back up. Because 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 (as described in Section 2). The merge will invoke at most $O(n/B)$ such flushes because 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 because adjacent elements of the list will be in the same block. □

We now consider mergeSort as a whole.

THEOREM 3.3. *For a compact list* `A`*, the evaluation of* `mergeSort(A)` *starting with any cache state will have cache cost given by Equation 2 and will return a compact list.*

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, because the input list is compact, for $k'n \leq M$ the input fits in the read cache (for some constant k'). Therefore, the cache cost 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 cost 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 and hence solves to the claimed result. □

Here we just outline the k-way mergeSort algorithm and state the cost bounds, which are optimal for sorting. The sort is similar to mergeSort but instead of splitting into two parts recursively sorting each, it splits into k parts. A k-way merge is then used to merge the parts. The k-way merge can be implemented using a priority queue. The sort in **ICFP** has the optimal bounds for sorting given by Equation 3.

Matrix Multiply.

```
datatype M = Leaf of int
           | Node of M * M * M * M

fun mmult(Leaf $a$, Leaf $b$) = Leaf($a \times b$)
  | mmult(Node($a_{11}, a_{12}, a_{21}, a_{22}$),
          Node($b_{11}, b_{12}, b_{21}, b_{22}$)) =
  let
    fun madd(Leaf $a$, Leaf $b$) = Leaf($a + b$)
      | madd(Node($a_{11}, a_{12}, a_{21}, a_{22}$),
             Node($b_{11}, b_{12}, b_{21}, b_{22}$)) =
          Node(madd($a_{11}, b_{11}$),madd($a_{12}, b_{12}$),
               madd($a_{21}, b_{21}$),madd($a_{22}, b_{22}$))
  in
    Node(madd(mmult($a_{11}, b_{11}$),mmult($a_{12}, b_{21}$)),
         madd(mmult($a_{11}, b_{12}$),mmult($a_{12}, b_{22}$)),
         madd(mmult($a_{21}, b_{11}$),mmult($a_{22}, b_{21}$)),
         madd(mmult($a_{21}, b_{12}$),mmult($a_{22}, b_{22}$)))
  end
```

Figure 11: Matrix Multiply.

Our final example is matrix multiply. The code is shown in figure 11 (we have left out checks for matching sizes). This is a block recursive matrix multiply with the matrix laid out in a tree. 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 cost $O(n^2/B)$ 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 matrix will be compact. Also, every recursive sub-matrix is itself compact.

THEOREM 3.4. *For compact $n \times n$ matrices `A` and `B`, the evaluation of `mmult(A,B)` starting with any cache state will have cache cost given by Equation 1 and will return a compact matrix as a result.*

PROOF. Matrix addition has cache cost $O(n^2/B)$ and generates a compact result because we traverse the two input matrices in preorder traversal and we generate the output in the same order. Because 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 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 a recurrence,

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

which solves to $O\left(\frac{n^3}{B\sqrt{M}}\right)$. The output is compact because 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. □

4. CONCLUSION

The present work extends the methodology of Blelloch and Greiner [4, 13] to account for the cache complexity of algorithms stated in terms of two parameters, the cache block size and the number of cache blocks. Analyses are carried out in terms of the semantics of **ICFP** given in Section 2, and are transferred to the Ideal Cache Model by a provable implementation, also sketched in Section 2. The essence of the idea is that conventional copying garbage collection can be deployed to achieve cache-efficient algorithms without sacrificing abstraction by resorting to manual allocation and cacheing of data objects. Using this approach, we are able to express algorithms in a high-level functional style, analyze them using a model that captures the idea of a fixed size read and allocation stack, without exposing the implementation details, and still match the asymptotic bounds for the ideal cache model achieved using low-level imperative techniques including explicit memory management. For sorting, the bounds are optimal.

One direction for further research is to integrate (deterministic) parallelism with the present work. Based on previous work [4, 13] 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. 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.

5. 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] 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*. IBFI, Schloss Dagstuhl, Germany, 2005.

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

[5] G. E. Blelloch and R. Harper. Cache and i/o efficent functional algorithms. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 39–50. ACM, 2013.

[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.

[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.

[8] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[9] A. Church. *The calculi of lambda-conversion*. Annals of mathematics studies. Princeton university press, 1941.

[10] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Commun.*

ACM, 31(9):1128–1138, 1988.

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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*. Springer, 2003.

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.