Automatic Synthesis of Out-of-Core Algorithms

Yannis Klonatos Andres Nötzli Andrej Spielmann Christoph Koch Viktor Kuncak
School of Computer and Communications Sciences, EPFL
{yannis.klonatos, andres.notzli, andrej.spielmann, christoph.koch, viktor.kuncak}@epfl.ch

ABSTRACT

We present a system for the automatic synthesis of efficient algorithms specialized for a particular memory hierarchy and a set of storage devices. The developer provides two independent inputs: 1) an algorithm that ignores memory hierarchy and external storage aspects; and 2) a description of the target memory hierarchy, including its topology and parameters. Our system is able to automatically synthesize memory-hierarchy and storage-device-aware algorithms out of those specifications, for tasks such as joins and sorting. The framework is extensible and allows developers to quickly synthesize custom out-of-core algorithms as new storage technologies become available.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*; D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

Out-of-core algorithms, synthesis, memory hierarchies

1. INTRODUCTION

The design of performance-critical software systems must depend on the hardware on which these systems run. This is particularly true for data-intensive computations. The research literature describes numerous out-of-core algorithms designed and optimized for a variety of hardware and storage device configurations [23, 13, 9, 25, 5, 21, 27, 16, 18]. These are case studies of how understanding memory hierarchies and data locality can drive algorithm design.

To this day, no methodology exists for creating such algorithms. We must rely on significant creative talent to serve our need for such algorithms, which remain publishable as original research contributions.

The introduction of a new storage or memory technology requires the development of new versions of most out-of-core algorithms. This leads to an arms race between the developers of hardware on one hand and of software systems and out-of-core algorithms on the other. Each new development in hardware calls for numerous research contributions on the software side, to update a multitude of algorithms and systems. Given the rapid rate of hardware innovation and the increasing popularity of hardware specialization, the ultimate consequence is a modest software crisis.

To address this challenge, we propose the use of software synthesis to automatically generate specialized out-of-core algorithms. The input is a naive memory hierarchy oblivious algorithm and a description of the target hardware setup and memory hierarchy. We encode fundamental principles of out-of-core algorithm design, many of which aim at the maximization of data locality, as transformation rules. The application of such a rule to the algorithm results in an equivalent algorithm which *may* have better performance on real hardware. By applying transformation rules, we create a navigable search space of equivalent algorithms. To be able to choose an optimal algorithm, we develop a cost estimation procedure based on the given hardware description.

The next example illustrates our approach:

EXAMPLE 1. The simplest way to implement a join algorithm on relations R and S is with two nested for-loops:

for ($x \leftarrow$ R) for ($y \leftarrow$ S) if joinCond(x,y) then [$\langle x,y \rangle$] else []

This program is an intuitive description of the programmer's intention. Let us assume a scenario where the input is stored on a hard disk and the output is not written anywhere (e.g., it is consumed by the CPU). Then, ignoring any buffering of the hard disk and the operating system, this program transfers every tuple of R and S from the hard disk separately. The efficiency of the algorithm can be improved significantly if we reduce the number of disk seeks by accessing the relations in larger contiguous blocks. Also, the semantics of the program does not change if the loops are reordered so that the outer relation is the smaller, but this further reduces the amount of seeking. By expressing such knowledge as transformation rules, we can automatically transform the above program into one that implements these two optimizations:

($\lambda\langle$R, S$\rangle$.for ($xBlock$ [k_1] $\leftarrow$ R)
  for ($yBlock$ [k_2] $\leftarrow$ S) for ($x \leftarrow$ xBlock)
    for ($y \leftarrow yBlock$) if joinCond(x,y) then [$\langle x,y \rangle$] else [])
(if length(R) $\leq$ length(S) then $\langle$R, S$\rangle$ else $\langle$S, R$\rangle$)

When the block-size k1 of $xBlock$ is maximized, this is the canonical Block Nested Loops Join. □

In the above example, we used the following transforma-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$15.00.

tion rule that turns a loop into a buffered scan with block-based transfers of block size k:

for $(x \leftarrow \mathsf{S})\ e \Rightarrow$
  for $(xBlock\ [k] \leftarrow \mathsf{S})$ for $(x \leftarrow xBlock)\ e$

This rule says that the left-hand side program is equivalent to the right-hand side and suggests that, subject to the targeted memory hierarchy, the latter is likely to be more efficient than the former. Indeed, the latter program requires less seeking on the hard disk.

To realize our vision of out-of-core algorithm synthesis, we have addressed the following challenges:

Design of a new language. We have designed a domain-specific language (DSL) called OCAL (Out-of-Core Algorithm Language), described in Section 3. The primary design goals of OCAL are (i) to be expressive enough for a variety of out-of-core algorithms, (ii) to be succinct enough to keep typical algorithms short and the search space for program synthesis manageable, and (iii) to keep syntax and semantics of the language simple to facilitate analysis and program transformation. In order to make it reasonably easy to cost programs and apply transformations, we should avoid constructs such as unrestricted recursion, mutable values, and side effects. As a consequence, we avoid imperative and low-level languages such as C for program representation during synthesis.

OCAL is defined as Monad Calculus on lists [8, 26] with a fold expression. It satisfies the above three design desiderata. (i) It is expressive, extending the power of nested relational algebra by the ability to process collections sequentially and exploiting order, which is central to capturing the essence of most out-of-core algorithms. (ii) Named OCAL function definitions can be used to keep OCAL programs short; these definitions are treated like language extensions in our synthesis system. (iii) OCAL is a simple purely functional language without side-effects in which recursion is confined to fold (and flatMap). Transformations in OCAL can be applied locally due to its functional and algebraic qualities. Finally, since full recursion is excluded and for-comprehensions (functional for-loops as in, say, XQuery and Scala) are straightforward to define in OCAL, even users only familiar with imperative languages can read most OCAL programs without great difficulty. It is relatively easy to map OCAL programs to imperative (C) code.

Cost Estimation and Cost Minimization. We need a systematic cost estimation framework to reason about the efficiency of OCAL programs. This requires an easily computable cost measure for evaluating the performance of each program that we explore. This measure, presented in Section 5, is a function of the algorithm, the memory hierarchy and statistics about the input. One contribution of this paper is the demonstration that in this domain it is possible to efficiently and automatically perform such estimation, and that the estimates are predictive enough to differentiate more efficient from less efficient algorithms on a given memory hierarchy.

There are two orthogonal aspects of cost estimation: structural program transformations and parameter selection. For the former, we use a breadth-first search strategy to explore the space of structurally different programs, and we use constants derived from the given memory hierarchy to build a cost function. To perform the latter, each program is also parameterized with values such as sizes of blocks and buffers. In Example 1, k is one such parameter. We use our cost estimation rules to characterize the running time estimate as a (possibly non-linear) function of those parameters. We have also implemented the non-linear optimization solver described in [19] to tune the values of parameters so as to minimize the cost estimate. We have found this strategy to be computationally feasible and to yield efficient programs for various memory hierarchies.

Development of a program synthesizer. Based on the language, rewrite rules, and costing framework, we have implemented OCAS, the Out-of-Core Algorithm Synthesizer. The input to OCAS consists of two orthogonal items: (1) a naive memory-oblivious algorithm given in OCAL; and (2) the structure and parameters of a memory hierarchy and storage devices. From this input, OCAS automatically derives efficient algorithms that have the same functional behavior as the initial specification algorithm, but whose performance is tuned to the given memory hierarchy. To do so, it uses a library of transformation rules, which we discuss in Section 6. OCAS then generates C code out of the optimized algorithm, using an OCAL-to-C code generator. Our approach also necessitates a technique, presented in Section 4, for describing memory hierarchies, such that device properties can be expressed sufficiently abstractly.

Because OCAS operates automatically, it is possible to deploy it even in environments where the system configuration changes dynamically, such as cloud infrastructures. OCAS can be used at installation time to adapt a piece of data management software to a computer, or at deployment time via just-in-time-compilation to make best use of fresh information on the availability of system resources.

Most importantly, the design of OCAS provides the so far missing methodology for designing efficient out-of-core algorithms, and even automatizes algorithm creation. Developers may need to make use of the extensibility of OCAS to adapt to unforeseen developments, but there is no need to "reinvent the wheel"; The basic machinery of OCAS will remain unchanged.

Providing an extensible architecture. Extensibility is an important property of the design of OCAS. Developers should be able to easily adapt OCAS as new hardware platforms become available and new algorithms are proposed. The library of program transformation rules of OCAS can be extended to implement new ways of using data locality considerations to create better algorithms. Furthermore, we can create named definitions in OCAL that can subsequently be used like new language operations. For each such definition, we can extend OCAS by matching code generator and cost function plugins to allow the synthesizer to make use of a particularly efficient implementation of that new language feature. Thus, definitions (in conjunction with code generator and cost function extensions) do *not* increase the expressiveness of the language but the efficiency of the algorithms created.

We conclude with an experimental evaluation of our system. We use OCAS to derive C code for algorithms such as Block Nested Loops Join, GRACE Hash Join and the External Merge-Sort in their canonical textbook forms starting from a naive specifications of joins and sorting. We also present examples of algorithms specialized for memory hierarchies that are not yet found in textbooks, such as a join

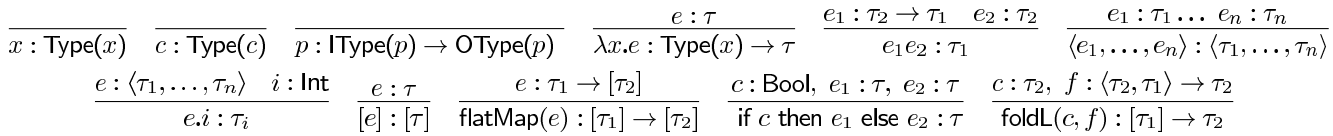

Figure 1: The type system of OCAL.

algorithm for flash drives. We present these case studies and their evaluation in Section 7.

2. RELATED WORK

There is very little work on automating out-of-core algorithm design. In contrast, a great amount of work has been done on manually developing specialized out-of-core algorithms for various tasks and memory hierarchies. In our work, we often refer to canonical algorithms for certain database management tasks. Their descriptions can be found in the standard text books like [23]. More recently, effort has been expended on designing algorithms for flash memory [21, 18, 5], the intricate memory hierarchies of graphics cards [9, 13, 25, 27], and multi-level memory hierarchies [16]. These papers demonstrate that the state of the art in developing out-of-core algorithms is to manually carry out ad-hoc effort; one cannot yet rely on automation or a clear design methodology.

The idea of automatic program transformation is present in [17]. However, this work does not take into account the characteristics of the architecture as OCAS does. This makes the approach limited as new architectures become available. Furthermore, the authors focus only on for-loops while OCAS proposes the use of a new language that (i) supports a wider set of constructs and (ii) is extensible to allow for easy addition of new definitions. Synthesis appears in domains other than data management as well. For instance, hardware-specific synthesis of linear transforms and other mathematical functions is the aim of SPIRAL [22].

In the Sequoia project [24], a general-purpose C-like language is presented that has *explicit* knowledge of the topology of the machine, and allows writing programs that efficiently utilize the hierarchy and the available parallelism. The Sequoia system does not perform software synthesis, so the programmers must specify the out-of-core algorithms themselves. However, it still handles other aspects of out-of-core algorithms like our tool, such as parameter selection.

OCAS performs algebraic manipulations to obtain more efficient equivalent programs; this idea can also be found in work on functional programming [20, 7]. Recursion schemas like folding [12] also play an important role in our work, especially for our sorting algorithms.

For static analysis of the running costs of functional programs, we draw inspiration from [15, 14, 10, 11]. COSTA [4] is a general-purpose cost estimation system for Java bytecode. This makes it applicable to languages more powerful than the one described in this paper. However, for the same reason, COSTA often fails to deduce bounds as tight as those of our system when working with a restricted custom-designed language. In particular, for the Merge-Sort algorithm that we use in one of our examples, we could not bring COSTA to estimate the asymptotically correct cost bound of $O(n \log n)$.

3. THE OUT-OF-CORE DSL

In this section we present OCAL (Out-of-Core Algorithm Language) which OCAS uses to represent data processing algorithms. The design of OCAL provides enough expressive power to describe commonly used algorithms ranging from traditional relational algebra operators, such as selection, projection and joins, to additional aspects of data processing such as sorting. At the same time, OCAL also allows easy application of transformation rules and costing as it will be discussed in more detail in Section 5 and Section 6.

The base language. OCAL extends Monad Calculus on lists [8, 26] with a fold expression. Consequently, the proposed language is more expressive than nested relational calculus. Starting from a totally ordered set D of atomic values that includes integers, booleans and strings, values are built inductively from D using list and tuple construction as formalized by the following grammar:

$$\tau ::= \mathsf{D} \mid \langle \tau, \ldots, \tau \rangle \mid [\tau]$$

The typing rules of the language are presented in Figure 1 where $e, e_1, \ldots, e_n$ range over expressions, x over variables, c over primitive constants and $\tau, \tau_1, \ldots, \tau_n$ over types. Each value x is assigned a $\mathsf{Type}(x)$ and, similarly, constants c have a type $\mathsf{Type}(c)$. Functions in OCAL are of type $\tau_1 \to \tau_2$ where τ_1 and τ_2 are value types. As an example, the type of a join operator for two binary relations on D is:

$$\langle [\langle \mathsf{D}, \mathsf{D} \rangle], [\langle \mathsf{D}, \mathsf{D} \rangle] \rangle \to [\langle \mathsf{D}, \mathsf{D}, \mathsf{D}, \mathsf{D} \rangle]$$

In the same figure, p ranges over primitive functions including boolean connectives ($\wedge$, $\vee$, $\neg$); equality of values of various types and comparison of basic data types D ($==$, $\leq$, $\geq$); a list union operator $\sqcup$, and further functions on tuples of values of D that only require a constant amount of memory (e.g., arithmetic operations). IType and OType are the input and output types of p, respectively.

The addition of a fold expression to Monad Calculus adds the ability to express sequential computation, which is essential for data processing algorithms including sorting. Folding from the left – foldL(c,f) – encodes a restrictive recursion pattern, an iterative application of the binary function f to elements of an input list. When using an infix operator $\oplus$, foldL is defined as follows:

$$\mathsf{foldL}(c,\oplus)([v_1,v_2,\ldots,v_n]) = (\cdots((c \oplus v_1) \oplus v_2) \oplus \cdots \oplus v_n)$$

Extensibility. Developers also have the ability to provide additional *definitions*, expressed in terms of the base language. Figure 2 presents schemes of definitions, where we use symbol _ as a placeholder for an unused function argument. We make the following observations regarding these definitions.

The head and tail constructs are used to extract elements from a list. They are undefined when the list is empty.

The functional for loop returns a value of a list type, which is the concatenation of list-typed values computed by its

body at each iteration. This is similar to the for loop in XQuery and to flatMap/ext in other languages [3, 8]. The parameter k concerns blocking and is explained in detail in Section 6. Whenever omitted, its value is assumed to be 1.

The treeFold construct generates a tree-shaped bracketing for the applications of a function f which takes k arguments. For example, for a ternary f we have:

treeFold[3](c,f)([v_1,v_2,...,v_6]) = $f(f(v_1,v_2,v_3),f(v_4,v_5,v_6),c)$

This construct is used to represent divide and conquer strategies, as found in e.g. Merge-Sort. It uses a queue to store the initial elements and the intermediate results.

The unfoldR function iterates over a tuple of n lists simultaneously. In every iteration the n-ary function f is applied, which computes part of the output and removes at most one element from the beginning of each list. The computation terminates when all lists are empty. The result of each iteration is appended to the intermediate result from the previous iteration starting with an empty list. We can use unfoldR to express the merging of two sorted lists as unfoldR(mrg) and the zipping of n lists as unfoldR(z).

The partition function groups a set of tuples by their first elements.

For a given fixed k, the funcPow[k](f) definition scheme yields a nonrecursive definition to obtain a 2^k-ary function using multiple applications of a binary function f.

Generating C code from OCAL. As we mentioned earlier, OCAS generates C code out of programs written in OCAL by translating each expression to an appropriate sequence of C statements. C is the target language since it is widely used in database systems development. By default, OCAS expands definitions and generates code for each individual expression of the base language. In order to increase efficiency, developers can overwrite the default code generators for expressions and definitions using generator plugins. OCAS contains efficient generator plugins for all definitions in Figure 2. For instance, our partition definition as shown in Figure 2 has $O(n^2)$ complexity, even though there exists a linear implementation with the same semantics. By providing a code generator plugin for this construct, the linear implementation can be used. Similarly, the definitions of the head and length functions have linear time complexity, even though there exist suitable implementations for constant time execution. Finally, because the inner function of unfoldR can only access the head of the lists and the output is produced sequentially, we can transfer *blocks* of elements at once, as we present in Section 6.

```
head : [τ] → τ
  := λl.foldL(⟨true, 0⟩, λ⟨a, x⟩.if a.1 then ⟨false, x⟩ else a)(l).2

tail : [τ] → τ
  := λl.foldL(⟨true, []⟩, λ⟨a, x⟩.
       if a.1 then ⟨false, []⟩ else ⟨false, a.2 ⊔ [x]⟩)(l).2

length : [τ] → Int
  := foldL(0, λ⟨sum, _⟩.sum + 1)

avg : [D] → D
  := (λx.(x.1/x.2))(foldL(⟨0, 0⟩, λ⟨a, x⟩.⟨a.1 + x, a.2 + 1⟩))

for (x [k] ← R) e : [τ1] → [τ2]
  := foldL(⟨[], []⟩, λ⟨a, x⟩.if length(a.1) − 1 == k then
        ⟨[], a.2 ⊔ f(a.1 ⊔ x)⟩ else ⟨a.1 ⊔ x, a.2⟩)

treeFold[k](c, f) : [τ1] → [τ2]
  := λseed.foldL(⟨[], seed⟩, λ⟨a, _⟩.
     if length(a.2) == 1 ∧ a.1 == [] then a
     else if length(a.1) == k then ⟨[], a.2 ⊔ f(a.1)⟩
     else if tail(a.2) != [] then ⟨a.1 ⊔ head(a.2), tail(a.2)⟩
     else ⟨a.1 ⊔ head(a.2), [c]⟩)(seed ⊔ seed)

unfoldR(f) : ⟨[τ1], ..., [τn]⟩ → [τr]
  := λseed.(foldL(⟨[], seed⟩, λ⟨a, _⟩.
     if a.2 == ⟨[], ..., []⟩ then ⟨a.1, ⟨[], ..., []⟩⟩
     else ⟨a.1 ⊔ f(a.2).1, f(a.2).2⟩))(seed.1 ⊔ ... ⊔ seed.n)

mrg : ⟨[τ], [τ]⟩ → ⟨[τ], ⟨[τ], [τ]⟩⟩
  := λ⟨l1, l2⟩.
     if length(l1) == 0 ∧ length(l2) == 0 then ⟨[], ⟨[], []⟩⟩
      else if length(l1) == 0 then ⟨[head(l2)], ⟨[], tail(l2)⟩⟩
      else if length(l2) == 0 then ⟨[head(l1)], ⟨tail(l1), []⟩⟩
      else if head(l1) < head(l2) then ⟨[head(l1)], ⟨tail(l1), l2⟩⟩
      else ⟨[head(l2)], ⟨l1, tail(l2)⟩⟩

z : ⟨[τ1], ..., [τn]⟩ → ⟨[⟨τ1, ..., τn⟩], ⟨[τ1], ..., [τn]⟩⟩
  := λ⟨l1, ..., ln⟩.
       ⟨[⟨head(l1), ..., head(ln)⟩], ⟨tail(l1), ..., tail(ln)⟩⟩

partition : [⟨τ1, ..., τn⟩] → [τ1, [⟨τ2, ..., τn⟩]]
  := foldL([], λ⟨ps, x⟩.
    (λnps.if nps.1 then nps else ps ⊔ ⟨x.1, [x.2]⟩)(
       foldL(⟨false, []⟩, λ⟨nps, xs⟩.if xs.1 == x.1
        then ⟨true, nps ⊔ [xs ⊔ [x.2]]⟩ else ⟨false, nps ⊔ [xs]⟩)(ps)))

funcPow[1](f) : ⟨τ1, τ2⟩ → τ3
  := f
funcPow[k + 1](f) : ⟨τ1, ..., τ2^k⟩ → τr
  := λ⟨a.1, ..., a.2^(k+1)⟩.
   f(funcPow[k](f)(a.1, ..., a.(2^k)),
      funcPow[k](f)(a.(2^k + 1)), ..., a.2^(k+1)))
```

Figure 2: Examples of definitions

4. MEMORY AND STORAGE MODEL

Automated transformations in OCAS are driven by a model of the memory hierarchy. For this purpose, the developer must specify a tree-shaped hierarchy where every node represents a hardware component able to store data and an edge represents the ability to transfer data between two nodes. In Example 1, the hierarchy consists of a main memory node at the root with a single child node representing a hard disk.

Every node is attributed a set of properties that provide information about its characteristics. This is merely an abstract description of each node's characteristics, since precise modeling of the architectural and physical attributes of nodes is beyond the scope of this work. Examples of such properties are presented in Figure 3.

Our model makes three assumptions. First, events between distinct hierarchy levels do not interfere with each other (we assume DMA transfers). Second, there exists a single processing unit which executes all computation and can only access data that is stored at the root node of the tree. Third, we assume synchronous I/O and that the hardware properties, such as the throughput and seek time of hard disks, remain constant.

For a program, the location of the input data, as well as the output node, must both be specified. If the output node is not set, we assume that the output is consumed by the CPU. Each data value resides in a node. In order to perform computation on those values, they must be transferred to the root node. Thus, for a given program, OCAS has to

Size. The size of the device. This property must be set for all nodes.

Pagesize. The data at this node must be accessed by pages of this size. If it is possible to address every byte individually then pagesize $= 1$.

Maximum length of a write sequence (maxSeqW). The maximum amount of data that it is possible to write in a sequence, using a single I/O request. For flash drives this is equal to the erase block size.

Maximum length of a read sequence (maxSeqR). The maximum amount of data that it is possible to read in a sequence, using a single I/O request.

Edge properties: Weights of InitCom[m1 $\rightarrow$ m2] and UnitTr[m1 $\rightarrow$ m2] cost events.

Figure 3: Examples of properties of the hierarchy

infer transfers for the set of values that have to be accessible by the processing unit throughout the execution of the program. In Example 1, values x and y have to be transferred to RAM before performing the join.

Moving a data value v from one hierarchy level to another induces *costs*. We leave the specifics of cost computation of OCAL expressions for Section 5, but we note here that the final cost depends on the paths used for data transfers. The act of transferring data concerns not only the input and the output but intermediate results as well.

In order to model the cost of moving data along an edge in the memory hierarchy, each edge has two cost metrics associated with it. Using different costs for different edges enables more accurate cost estimation. First, we consider the cost of initiating a transfer between the two hierarchy nodes (InitCom event). If either of the nodes is a hard disk, this corresponds to a seek in our model. Similarly, in order to transfer data to a flash drive, a block has to be erased before data can be written. The second metric is the cost of transferring a unit of data between the two hierarchy levels (UnitTr event). If the developer chooses to ignore certain cost events, he can set their value to zero. This allows our system to, for example, ignore the cost of InitCom for RAM when considering I/O intensive workloads. Both costs can be collected either from the device specifications or using standard tools like e.g. Seeker [2] for hard disk seeks. We follow this approach in our evaluation.

Because we model memory hierarchies as trees whose leaves are storage devices and whose root is the fastest level of the hierarchy, we cannot model, say, general parallel computation. We are currently working on a way of modeling hardware deeply with the goal of ultimately being able to automatically infer program transformation rules and cost functions from the hardware description.

5. AUTOMATED COST ESTIMATION

Sufficiently accurate cost estimation of programs is essential because it is used by OCAS to compare programs in terms of efficiency. In the domain of out-of-core algorithms we are mainly interested in costs introduced by moving data around in the memory hierarchy. Thus, we currently neglect the actual computation cost of a program in our system. Instead, we opt for modeling only the two aspects of data transfers that we introduced in Section 4: initiating the transfer and actually transferring the requested data.

This section provides a stepwise description of the communication cost computation. First, we describe *how* to compute the result size of each OCAL expression. This is needed since the input typically represents structured data, and thus we need to estimate not only the total size, but also the sizes of nested components that may be separately used in subcomputations. Then, we describe *when* data transfers are introduced in our cost model. Finally, we analyze how the cost estimator separately computes two aspects of data transfers in order to provide the final cost formula and we discuss the extensibility of the costing.

Note that the costing of a program in OCAS does not require to actually run the program. This is important, since actual execution may be very costly. This aspect enables our methodology to be used to compare a large number of programs efficiently, which is essential when exploring variations of a program by applying transformations. We discuss transformation rules in greater detail in Section 6 and in this section we focus on how to cost one single program.

Figure 4 provides a guiding example for this section. It shows the different steps carried out by the cost estimation engine for every expression of a block nested loop join that reads two relations R and S from HDD into RAM, joins them there and finally writes the result back to HDD.

5.1 Estimating the result size of expressions

Given that OCAL programs are compositions of expressions, and that each expression may increase the amount of output, we must estimate the result size of every expression in OCAL. To do that, we introduce the notion of *annotated types*, which annotate lists types with cardinalities. The corresponding grammar is:

$$\alpha ::= [\alpha]^x \mid \langle \alpha_1, \ldots, \alpha_n \rangle \mid c$$

An annotated type α is either a list of form $[\alpha]^x$ where x is the cardinality, a tuple of annotated types or a constant size c. This notation allows us to represent the size of values while retaining their structure. It is worth mentioning that the length of a list is not restricted to integer constants but can be described by an arithmetic expression containing variables. As an example of an annotated type, $\langle [[1]^y]^x, [\langle 1, 1 \rangle]^z \rangle$ represents a tuple composed of a list of lists and a list of tuples. By using variables we can express the result size as a function of the input sizes and other parameters without having to recompute the cost of a program every time the size of its inputs or other parameters change.

By using annotated types, the result size of expressions can be then estimated as shown in Figure 5. In what follows, we sometimes write $x \cdot [b]^y$ to denote $[b]^{x \cdot y}$. The recursive function R defines the result size as an annotated type for an expression in a context Γ, which is a set that maps symbols to annotated types. This context is extended every time new symbols are referenced. In order to turn the estimate of a result size into a single arithmetic expression, we define the function size which turns an annotated type to an integer-valued arithmetic expression representing the size of the annotated type in bytes. In addition, we define card and elem to extract information about lists. Since function definitions do not produce any results until they are applied to a value, in our costing we assume that all of them are matched with corresponding function applications. The cost of the flatMap construct is the same as that of for with k set to 1.

Observe that we perform *worst-case* analysis of the result size of each expression. For instance, for nested lists, we take the maximum of the lengths of the inner lists. This design choice may lead to overestimation of result sizes, e.g. in the

Expression	Context	Result size	UnitTr $m_{\mathsf{HDD}} \to m_{\mathsf{RAM}}$	UnitTr $m_{\mathsf{RAM}} \to m_{\mathsf{HDD}}$	InitCom $m_{\mathsf{HDD}} \to m_{\mathsf{RAM}}$	InitCom $m_{\mathsf{RAM}} \to m_{\mathsf{HDD}}$
for (xB $[k_1] \leftarrow$ R)	$\Gamma_1 = \mathsf{R} \mapsto [1]^x, \mathsf{S} \mapsto [1]^y$	$[\langle 1,1\rangle]^{x \cdot y}$	$x + \frac{x}{k_1} y$	$2xy$	$x/k_1 + \frac{xy}{k_1 k_2}$	$2xy/k_o$
for (yB $[k_2] \leftarrow$ S)	$\Gamma_2 = \Gamma_1 \cup xB \mapsto [1]^{k_1}$	$[\langle 1,1\rangle]^{k_1 \cdot y}$	y	$2k_1 y$	y/k_2	$2k_1 y/k_o$
for ($x \leftarrow$ xB)	$\Gamma_3 = \Gamma_2 \cup yB \mapsto [1]^{k_2}$	$[\langle 1,1\rangle]^{k_1 \cdot k_2}$	0	$2k_1 k_2$	0	$2k_1 k_2/k_o$
for ($y \leftarrow yB$)	$\Gamma_4 = \Gamma_3 \cup x \mapsto 1$	$[\langle 1,1\rangle]^{k_2}$	0	$2k_2$	0	$2k_2/k_o$
if joinCond(x,y)	$\Gamma_5 = \Gamma_4 \cup y \mapsto 1$	$[\langle 1,1\rangle]^1$	0	0	0	0
then $[\langle x,y\rangle]$	Γ_5	$[\langle 1,1\rangle]^1$	0	0	0	0
else []	Γ_5	0	0	0	0	0

Figure 4: Costing of an example of two unary relations R and S of type [Int]. The hierarchy has two nodes, an HDD and a RAM (root node), and we assume that the size of Int is 1.

$$\mathsf{card}([\alpha]^x) := x \qquad \mathsf{elem}([\alpha]^x) := \alpha \qquad \mathsf{size}([\alpha]^x) := x \cdot \mathsf{size}(\alpha) \qquad \mathsf{size}(\langle \alpha_1, \ldots, \alpha_n \rangle) := \mathsf{size}(\alpha_1) + \ldots + \mathsf{size}(\alpha_n)$$

$$\mathsf{size}(c) := c \qquad \mathsf{R}(\Gamma, x) := \Gamma(x) \qquad \mathsf{R}(\Gamma, [e]) := [\mathsf{R}(\Gamma, e)]^1 \qquad \mathsf{R}(\Gamma, \mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) := \mathsf{max}(\mathsf{R}(\Gamma, e_1), \mathsf{R}(\Gamma, e_2))$$

$$\mathsf{R}(\Gamma, c) := \mathsf{sizeof}(c) \qquad \mathsf{R}(\Gamma, e.i) := \mathsf{R}(\Gamma, e).i \qquad \mathsf{R}(\Gamma, e_1 \sqcup e_2) := \mathsf{R}(\Gamma, e_1) + \mathsf{R}(\Gamma, e_2) \qquad \mathsf{R}(\Gamma, \langle e_1, \ldots, e_n \rangle) := \langle \mathsf{R}(\Gamma, e_1), \ldots, \mathsf{R}(\Gamma, e_n) \rangle$$

$$\mathsf{R}(\Gamma, \mathsf{for}(x\ [k] \leftarrow e_1)\ e_2) := \tfrac{\mathsf{card}(\mathsf{R}(\Gamma, e_1))}{k} \cdot \mathsf{R}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_1))]^k\}, e_2)) \qquad \mathsf{R}(\Gamma, (\lambda x.e_1)(e_2)) := \mathsf{R}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, e_2)\}, e_1)$$

$$\mathsf{R}(\Gamma, \mathsf{foldL}(c, \lambda\langle a, x\rangle.e_1)(e_2)) := \mathsf{R}(\Gamma, c) + \mathsf{card}(\mathsf{R}(\Gamma, e_2))\Big(\mathsf{R}(\Gamma \cup \{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)\Big)$$

Figure 5: Data size estimation rules for every expression of OCAL.

case of if-then-else, the branch that gives the largest result may not be the one that will actually be taken during execution. However, as we show in our evaluation, even with this overestimation, OCAS can still differentiate more efficient programs from less efficient ones, with respect to the given memory hierarchy. Finally, we also allow the programmer to annotate any expression with a custom result size estimate. This may be needed since the static rules that OCAS uses for data size estimation may not capture specific algorithm semantics. A fold which produces a very small output in its last iteration, but very large outputs in all others is one example where these annotations allow programmers to explicitly express the intention of their algorithm.

5.2 Determining data transfer occurrences

OCAS models data transfers implicitly: whenever the execution context is extended with a new value, we account for an appropriate amount of transfers for this value. After modeling the transfer, this value is then considered to be in its new location. Furthermore, as soon as a value is not in the context anymore, it does not consume space for the hierarchy level it belonged to. By implicitly modeling data transfers, we enable separation between a program and its execution environment (memory hierarchy). This alleviates the need for programmers to annotate where intermediate values are stored throughout the execution of a program.

We use the following notation and semantics for data transfers. First, values are transferred from a hierarchy node m_s, where they originally reside, to a memory node m_d. In order to simplify costing of a program, we assume that data transfers happen only between adjacent memory nodes (m_s is directly connected to m_d in the tree). Furthermore, if m_d is the root node, then an expression will be executed to process the fetched data, thus producing an output written at a node m_o, which has to be a child of m_d, possibly different from m_s (because otherwise no transfers are needed).

Finally, data transfers between adjacent hierarchy levels are constrained by the physical size of the participating nodes. Given that modeling replacement algorithms at each level of the memory hierarchy is a very complicated task, we choose a simpler solution. We use dedicated space for input (b_{in}) and output (b_{out}) buffers at each level, per value, so that their combined size does not exceed the size of the specific level. These buffers determine the amount of transfers necessary to process each value and will be utilized by the transformation rules presented in Section 6. Furthermore, when the output buffer is filled, it is completely evicted to the output memory level. Choosing good values for input and output buffer sizes is a critical aspect of designing high performance out-of-core algorithms. It is also a non-trivial task for developers, since choosing locally optimal solutions at each node may not give a globally optimal solution for the whole hierarchy. Thus, the automation that our system provides in that respect is very helpful to developers.

5.3 Estimating cost events

The core of cost computation concerns estimating the cost of InitCom and UnitTr transfer events occurring between adjacent nodes. OCAS estimates these events independently. Figure 6 shows how our system counts the amount of data transferred for various kinds of functions. For function definitions (along with their applications), the size of the argument determines how many bytes are transferred from m_s to m_d. The size of the result tells how many bytes are written out. In addition, as flatMap executes its inner function for every element, we have to multiply the number of the elements caused by this function by the length of the list. For foldL the situation is very similar but we also have to take into account that c has to be transferred to m_d as well.[1] Every expression other than function application basically just aggregates the number of events of its subexpressions. Calculating the amount of InitCom events is similar to computing the amount of data transferred. The total cost is then found if we add up the two separate costs, which gives a sin-

[1]The presented cost function is simplified and adapted to our examples. The general cost function is more complicated because OCAL is able to express algorithms with super-exponential running time.

Expression e	Cost of InitCom events $\mathsf{C}(\Gamma, e)$	Cost of UnitTr events $\mathsf{T}(\Gamma, e)$
$(\lambda x.e_1)(e_2)$	$\mathsf{C}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) + \mathsf{C}(\Gamma, e_2)$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{InitCom}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{InitCom}[m_d \to m_o]$	$\mathsf{T}(\Gamma \cup \{x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) + \mathsf{T}(\Gamma, e_2)$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$
$(\mathsf{for}(x\ [\mathrm{k}] \leftarrow e_2)\ e_1)$	$\frac{\mathsf{card}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{C}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_2))]^k\}, e_1)$ $+\mathsf{C}(\Gamma, e_2) + \frac{\mathsf{size}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{InitCom}[m_s \to m_d]$ $+\frac{\mathsf{size}(\mathsf{R}(\Gamma, e))}{b_{out}}\ \mathsf{InitCom}[m_d \to m_o]$	$\frac{\mathsf{card}(\mathsf{R}(\Gamma, e_2))}{k}\mathsf{T}(\Gamma \cup \{x \mapsto [\mathsf{R}(\Gamma, \mathsf{elem}(e_2))]^k\}, e_1)$ $+\mathsf{T}(\Gamma, e_2) + \mathsf{size}(\mathsf{R}(\Gamma, e_2))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$
$(\mathsf{foldL}(c, \lambda\langle a, x\rangle.e_1))(e_2)$	$\sum_{i=0}^{\mathsf{card}(\mathsf{R}(\Gamma, e_2))-1} \mathsf{C}\Big(\Gamma \cup \{a \mapsto i \cdot (\mathsf{R}(\Gamma \cup$ $\{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)),$ $x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1\Big) + \mathsf{C}(\Gamma, e_2)$ $+(\mathsf{size}(\mathsf{R}(\Gamma, c)) + \mathsf{size}(\mathsf{R}(\Gamma, e_2)))\ \mathsf{InitCom}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{InitCom}[m_d \to m_o]$	$\sum_{i=0}^{\mathsf{card}(\mathsf{R}(\Gamma, e_2))-1} \mathsf{T}\Big(\Gamma \cup \{a \mapsto i \cdot (\mathsf{R}(\Gamma \cup$ $\{a \mapsto \mathsf{R}(\Gamma, c), x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1) - \mathsf{R}(\Gamma, c)),$ $x \mapsto \mathsf{R}(\Gamma, \mathsf{elem}(e_2))\}, e_1\Big)$ $+\mathsf{T}(\Gamma, e_2) + (\mathsf{size}(\mathsf{R}(\Gamma, c))$ $+\mathsf{size}(\mathsf{R}(\Gamma, e_2)))\ \mathsf{UnitTr}[m_s \to m_d]$ $+\mathsf{size}(\mathsf{R}(\Gamma, e))\ \mathsf{UnitTr}[m_d \to m_o]$

Figure 6: Rules for computing the cost of InitCom and UnitTr events.

gle expression depicting the cost of a program as a function of various parameters like block and input sizes.

We end this section with two remarks. First, our system also allows the developer to define custom costs for definitions by extending the mechanisms for counting events and estimating result sizes with special cases. This feature allows to specify tighter bounds for special cases using the developer's expertise. If the developer does not specify a cost formula for his definitions, OCAS extends each definition and costs its inner expressions in order to get a cost estimation metric. Second, observe that when the target memory hierarchy changes, the costing formulas are changed accordingly, based on the above analysis. This may make a different set of transformation rules applicable and may, as a result, generate a different program as output. In the next section, we discuss the transformation rules in more detail, and show how they can generate a better program based on the cost formulas we presented here.

6. TRANSFORMATION RULES

In the previous section we discussed how to estimate the cost of a program based on the amount of data transfers it performs. In this section, we discuss the set of rules that transform a given program to another one with equivalent functionality that may have better performance with respect to a given memory hierarchy. Since it is rarely possible to determine analytically whether the application of a rule results in a performance improvement, we opt for cost-based optimization rather than using a deterministic recipe for obtaining efficient programs. OCAS exhaustively searches the space of equivalent programs, estimates the cost of each and then selects one with the best performance. For example, consider the following sequence of equivalent join algorithms between relations R and S of type list of tuples:

```
for (x ← R) for (y ← S) if joinCond(x,y) then [⟨x,y⟩] else []
⇓ [rule apply−block applied twice, for R and S respectively]
for (xBlock [k1] ← R) for (x ← xBlock) for (yBlock [k2] ← S)
  for (y ← yBlock) if joinCond(x,y) then [⟨x,y⟩] else []
⇓[rules swap−iter and seq−ac]
for (xBlock [k1] ← R) for[HDD⇝RAM] (yBlock [k2]← S)
  for (x ← xBlock) for (y ← yBlock)
    if joinCond(x,y) then [⟨x,y⟩] else []
⇓[rule order−inputs]
(λ⟨R, S⟩.for (xBlock [k1] ← R) for[HDD⇝RAM] (yBlock [k2]←S)
  for (x ← xBlock) for (y ← yBlock)
    if joinCond(x,y) then [⟨x,y⟩] else [])
(if length(R) ≤ length(S) then ⟨R, S⟩ else ⟨S, R⟩)
```

The first program is a naive implementation of a Nested Loops Join algorithm that issues a disk read every time it accesses a tuple from either of the two relations. The final program is a Block Nested Loops Join that uses the smaller relation in the outer loop. Both programs discard the output. Every step in the derivation is annotated with a transformation rule presented in Subsection 6.2.

6.1 From Principles to Transformation Rules

We have identified three main principles that drive the transformations performed by our system:

Data locality and block-based transfers. One heuristic that OCAS uses is to fetch the largest possible block of data to the processing unit *at once*. The justification is that, unless the input contains data that is never looked at by the algorithm, every data element has to be eventually fetched. Thus, if fetching is performed in larger chunks, then the number of InitCom events, which represent disk seeking and the costly erasure on flash drives, decreases. The transformation rule that applies this optimization is called apply−block.

The order in which individual data elements are accessed can also be changed by rules swap−iter and order−inputs. This is a class of optimizations whose effectiveness depends on the interaction of several levels of the memory hierarchy, rather than the properties of each individual level. Therefore, there does not exist a generally valid characterization of the cases when these optimizations are effective, other than suggesting that the performance after the application of the rule should improve.

Sequential versus random access. Some devices perform significantly better if data on them is accessed sequentially rather than in random order. A notable example are hard disks, but also writing sequentially to flash drives is more efficient because an erased block can be filled before

another one has to be erased somewhere else. Pre-fetching data by blocks into a level that does not have a performance penalty for random access can improve performance in programs where random access is confined to happen within blocks. Blocking is introduced through the apply−block rule.

Minimizing the number of passes through the input. Some programs require accessing every element of the input several times to compute the result. There are abundant examples of this behavior in data management systems: join algorithms may have to consider all pairs of members from two relations, comparison-based sorting algorithms have a theoretical bound of at least $\log n$ accesses to each input element, etc. OCAS uses two techniques based on this principle: Partitioning data by hash, represented by the rule hash−part; and Divide-And-Conquer, represented by the rule inc−branching. Both rules have the effect that the individual input elements need to be accessed fewer times, in fact only two times in the case of hash−part, but in a more random order. Therefore there is a trade off between the total amount of data transferred and the amount of seeking that this rule introduces.

In addition to the previous ideas, another class of optimization rules target improving the asymptotic computational complexity of the algorithm, such as the fldL−to−trfld rule. However, we do not make this kind of rules a priority of this paper, because they are rather independent of the usage of the memory hierarchy and they form a broad enough research topic on their own. Application of functional-style transformations to improve the asymptotic complexity of programs has been studied in e.g. [7] and [6], although not in the context of *automatic* synthesis of programs.

6.2 List of Transformation Rules

We write our rules as $e_1 \Rightarrow e_2$ where e_1 and e_2 are OCAL expressions. This means that whenever a part of a program matches e_1 then this part is equivalent to and can be replaced by e_2, leading to a new program. Most rules come with additional conditions on e_1 that determine when the rule can be applied. These conditions pose a challenge: some are undecidable in the general case or deciding them is too computation-intensive. In such cases, we implement a conservative estimation procedure that returns no false positives by deciding a stronger but simpler condition. This approach may lead our tool to fail to notice opportunities when a rule could be correctly applied but it never allows it to apply a rule in a non-valid context. We now describe the motivation of each rule, the conditions under which it can be applied and some examples of usage.

We show in the experimental section that this set of rules already covers a rich collection of programs. There are other principles that OCAS does not yet deal with. However, our tool can be easily extended by such principles in the form of new transformation rules that follow the same pattern as the ones presented in this section.

Increasing the Block Size (apply−block). The fold and flatMap constructs, as specified in Section 3, iterate over the elements of a list one by one, as they appear, in a sequential fashion. However, OCAS provides the for construct which allows iterating over blocks of elements, instead of one by one. Using the blocked for in place of the more granular counterpart is the aim of the following transformation rule:

for (x [1] ← R) [1] e ⇒
  for ($xBlock$ [k_1] ← R) for (x ← $xBlock$) [k_2] e

This rule can be applied both when fetching data towards the processing unit as well as to the data that is written as a result of evaluating expression e. To use it, we introduce the new annotation [k_2] (in the place shown) for buffering the output. In general, apply−block increases the amount of data read or written in a single I/O request, from the default single element to blocks of size k_1 and k_2, respectively. The value of k_2 is limited by the space and the maxSeqW property of the node where the elements are being written to, and k_1 by the maxSeqR property of the source node. The actual values of k_1 and k_2 are determined by the non-linear optimizer that we have implemented based on [19]. In short, for a single loop, a good heuristic is that both k_1 and k_2 should be as big as possible, subject to the aforementioned restrictions. However, if several nested loops over different ranges compete for space at the same node, this trivial heuristic does not work and we use the optimization solver to determine the block sizes.

If R is originally stored at node m_0, is fetched at m_1 and the output is written to node m_2, this rule reduces the number of InitCom$[m_0 \to m_1]$ cost events k_1-fold, and the number of InitCom$[m_1 \to m_2]$ events k_2-fold, as long as $m_0 \neq m_2$. If some of these nodes are hard disks, this rule decreases the number of disk seeks. In general, our system aims to replace every list-iterative construct with block size 1 with as many levels of nested equivalent constructs with larger block size as there are levels in the memory hierarchy. We note that we also use an analogous rule to introduce bigger blocks to our implementation of unfoldR.

Swapping The Order Of Iterative Constructs (swap−iter). Given two for or two flatMap constructs that iterate over two different lists, we can then change the order in which these two constructs are applied, as follows:

for (x_1 [k_{11}] ← $range_1$) [k_{12}] for (x_2 [k_{21}] ← $range_2$) [k_{22}] e ⇒
  for (x_2 [k_{21}]← $range_2$) [k_{22}] for (x_1 [k_{11}]← $range_1$) [k_{12}] e

This rule can be applied provided that the value of $range_2$ does not depend on x_1. We also have an analogous rule for loops with a condition:

for (x_1 [k_{11}] ← $range_1$) [k_{12}]
 if c then for (x_2 [k_{21}] ← $range_2$) [k_{22}] e_1 else e_2
⇓
for (x_2 [k_{21}] ← $range_2$) [k_{22}]
 for (x_1 [k_{11}] ← $range_1$) [k_{12}] if c then e_1 else e_2

Ordering Input Lists by Length (order−inputs)

$f \Rightarrow \lambda\langle x_1, x_2\rangle.$
  $f($if length$(x_1) \leq$ length(x_2) then $\langle x_1, x_2\rangle$ else $\langle x_2, x_1\rangle)$
$f \Rightarrow \lambda\langle x_1, x_2\rangle.$
  $f($if length$(x_1) \leq$ length(x_2) then $\langle x_2, x_1\rangle$ else $\langle x_1, x_2\rangle)$

The target of this rule are applications where the input is a tuple of lists whose order does not matter for the calculated result but may matter for efficiency. For instance, a Block Nested Loops join is more efficient if the outer relation is the smaller. These two rules can be applied if f is of type $\langle[\tau_1], [\tau_1]\rangle \to \tau_2$. It is easy to generalize this rule for functions whose input is a tuple of type $\langle[\tau_1], \ldots, [\tau_1]\rangle \to \tau_2$.

Hash Partitioning of Input (hash−part). The following procedure can sometimes improve the performance of an algorithm. Given a tuple of lists, we distribute the elements

of each of the lists into subsets, each containing elements that hash into a particular range. We thus obtain a tuple of lists of lists, where each list of lists represents the set of hash partitions of one of the original lists. These are then zipped together to form a value L of type list of tuples of lists, that has length s and contains all the tuples of corresponding partitions. The original algorithm is then mapped over the tuples. OCAS provides an efficient implementation of the partition definition, which is executed in linear time. The following transformation rule captures this idea:

$$f \Rightarrow \lambda\langle x_1, \ldots, x_k\rangle.(\mathsf{flatMap}(f)\ (\mathsf{zip}(\langle \mathsf{partition}(x_1), \ldots, \mathsf{partition}(x_k)\rangle))))$$

This rule works for any s when f is a function that iterates over a tuple of lists, for example a join. Most importantly, f must be such that when one takes the union of results of f applied to the s partitions, one gets the same result as applying f to the original input lists. This means that we do not care about the order in which the function processes the input elements.

If f is a program that accesses every element of its input more than once, applying this rule has the effect that all of the data is read only twice: once during the partitioning phase and once when applying f to the partitions, provided they are small enough to fit in the node into which f reads the data to from their original location. This is ensured by choosing s to be large enough. This rules is needed for synthesizing hash joins.

Increasing the Branching of treeFold (inc−branching).

$\mathsf{treeFold}[2^k](c, \mathsf{funcPow}[k](f)) \Rightarrow$
$\mathsf{treeFold}[2^{k+1}](c, \mathsf{funcPow}[k+1](f))$

The condition for this rule to work is that f has to be associative. When this rule is applied, the number of applications of the function inside the treeFold decreases but the function becomes more complicated as it accepts more arguments. More precisely, we get approximately $\frac{n}{2^k-1}$ applications of funcPow[k](f) instead of approximately n applications of f, where n is the length of the input list. Also, when initially converting treeFold[2] into treeFold[4], the use of this rule is usually preceded by a use of an auxiliary rule $f \Rightarrow$funcPow[1](f) which applies to any f.

In Section 7, we provide the example of deriving 2^k-way External Merge-Sort. There, the sorting algorithm operates on a list of lists, which necessitates the usage of the unfold definition. In this particular case, it is more efficient to actually execute the above transformation rule as follows:

$\mathsf{treeFold}[2^k](c, \mathsf{unfoldR}(\mathsf{funcPow}[k](f))) \Rightarrow$
$\mathsf{treeFold}[2^{k+1}](c, \mathsf{unfoldR}(\mathsf{funcPow}[k+1](f)))$

A detailed explanation is omitted due to space constraints.

Change of Folding Pattern (foldL−to−trfld).

$\mathsf{foldL}(c,f) \Rightarrow \mathsf{treeFold}[2](c,f)$

This rule works whenever f is associative and c is an identity element for f. The treeFold[2] pattern applies f the same number of times as foldL. However, if the size of the result of f and its computational complexity grow at least linearly with the size of its input, then treeFold[2] achieves better performance by balancing f's input sizes more equally.

Adding a Sequentiality Annotation (seq−ac). To enhance the precision of the cost estimation, we allow an expression to be annotated with a token $[m_1 \rightsquigarrow m_2]$. This notifies the costing engine that for this expression, all data transfers from m_1 to m_2 happen sequentially. This annotation serves only as an indicator for the costing engine and it does not change the semantics and implementation of the program. It can be applied when no other part of the program causes any communication to m_2. A syntactic check provides a sufficient condition.

For example, a for-loop that reads a page of the hard disk to the main memory in every iteration and does not otherwise touch the hard disk is allowed to have this annotation. In this case, instead of counting one such event for every iteration (which is the result of ordinary cost inference), the new InitCom cost is given by: $\max\left(1, \frac{\mathsf{totaltransfers}}{\min(m_1.\mathsf{maxSeqR}, m_2.\mathsf{maxSeqW})}\right)$. Another natural interpretation of this cost function optimization is writing multiple blocks of data to a flash drive after a block, usually much larger, has been erased.

Hard disk: size = 1T	pagesize = 4K
Flash drive: size = 512G	maxSeqW = 256K
Cache: size = 3M	pagesize = 512B
InitCom[HDD ↦ RAM] = 15ms	InitCom[RAM ↦ HDD] = 15ms
InitCom[RAM ↦ SSD] = 1.7ms	InitCom[RAM ↦ Cache] = 0.1ms
UnitTr[HDD ↦ RAM] = 1s/30M	UnitTr[RAM ↦ HDD] = 1s/30M
UnitTr[SSD ↦ RAM] = 1s/120M	UnitTr[RAM ↦ SSD] = 1s/120M

Figure 7: Node properties and associated cost units

7. EXPERIMENTAL EVALUATION

In this section we present our experimental platform and evaluate our approach with respect to the following points:

1. The quality of synthesized algorithms. We evaluate this aspect in two ways. First, we manually inspect the generated C code obtained from OCAS and check whether the code matches our expectations. Particularly for disk-based joins and sorting, we check whether we obtain exactly the standard textbook algorithms. Then, we evaluate the performance of the synthesized algorithms by running the generated C code on actual data on a hardware configuration that matches our memory hierarchy description.

2. The accuracy of the predictions compared to the actual execution times of the algorithms. We examine two aspects of this issue, the imprecision of the estimations caused by the fact that the cost formulas do not currently consider CPU costs, and the degree of overestimation caused by the fact that OCAS performs worst-case analysis.

3. The execution time of the synthesizer, given that the search space grows as longer chains of transformation rules are evaluated on larger programs.

7.1 Experimental Platform

Our platform[2] is a Mac OS X machine with an i7-2620M processor, standard 1TB Western Digital hard disk drives and one 500GB Apple SSD TS512C. Input and RAM buffer sizes are reported in bytes, and are specifically chosen for each experiment. The properties of our devices and the cost of unit events are listed in Figure 7. Costs not included are assumed to be zero. OCAS is implemented in Scala, so we use a Java Virtual Machine with 256MB of heap space. The C programs generated by OCAS are compiled using GCC

[2]This is for all experiments other than measuring cache misses.

Program	Spec. [s]	Opt. [s]	Act. [s]	Relation size R	Relation size S	Total buffer size	Search space	Steps	OCAS Runtime [s]
BNL - No writeout	4×10^9	411	545	1G	32M	8M	9287	6	17
BNL with cache - No writeout	4×10^9	445	533	1G	32M	8M	54202	7	370
(GRACE) hash join - No writeout	4×10^9	356	491	1G	32M	8M	28471	7	78.5
BNL writing to HDD	1016144	5058	4704	32K	256M	20K	2566	6	8.2
BNL wr. to other HDD	1016144	1689	2176	32K	256M	20K	7443	6	14.4
BNL writing to flash	561179	307	455	32K	256M	20K	7443	6	12.7
External sorting	1×10^9	157	272	1G	-	260K	130	10	2.9
Set Union	251931	396	499	2G	2G	48K	21	3	0.07
Multiset Union (sorted list)	251931	396	479	2G	2G	48K	21	3	0.06
Multiset Union (value-multiplicity)	251931	396	487	2G	2G	48K	21	3	0.07
Multiset Diff. (sorted list)	126033	266	137	2G	2G	48K	21	3	0.07
Multiset Diff. (value-multiplicity)	126033	266	153	2G	2G	48K	21	3	0.07
Column Store Read 5 cols.	125965	197	196	4G	-	5M	7	3	0.01
Column Store Read 10 cols.	251931	395	382	8G	-	10M	7	3	0.01
Duplicate Removal from a Sorted List	503862	546	882	16G	-	16K	7	3	0.16
Aggregation	125965	136	168	4G	-	32K	7	3	0.25

Table 1: Cost estimates for the naive specification algorithm (Spec) and the synthesized algorithm (Opt), actual running times of the generated C programs for the synthesized algorithms (Act), data sizes, and statistics on synthesis (search space size and depth, and synthesizer running time).

4.2. In what follows, the running time refers to the *actual execution time* of the generated programs.

7.2 Inspection and Performance Evaluation

The aim of this work is to automatically generate algorithms tuned for a particular memory hierarchy. To that end, we do not claim the optimality of the generated algorithms. We have instead manually verified that the generated algorithms are the same as those in textbooks [23]. Table 1 presents results for all of our experiments and it also contains the cost of the naive algorithm the user provides, which assumes one I/O (and one seek) per tuple processed.

Next, we analyze variations of the running BNL Join example and we explain in detail how OCAS automatically derives an External Merge-Sort of $n \cdot \log n$ complexity from a naive specification of an insertion sort of n^2 complexity. The purpose of these examples is to show that OCAS generates optimized algorithms and adapts its cost formulas and generated algorithms when the memory hierarchy changes.

Block Nested Loops (BNL) and Hash Join. All join examples start with the same naive join algorithm of Example 1. We create different algorithms from different memory hierarchy descriptions and parameterizations. Results for the BNL Join with no write-out are shown in the first line of Table 1. This corresponds to the example used so far in the paper.

When a memory hierarchy that consists of a hard disk and the main memory is extended with one level of CPU cache, OCAS generates a version of Block Nested Loops join with additional for loops that make use of the available cache.

The reader can verify that by applying this transformation, which corresponds to loop tiling, the program becomes more cache-friendly. As a result, there is a small performance improvement, as shown in Table 1. We make two observations. First, the underestimation of OCAS is smaller in this experiment, because a previously ignored part of the memory hierarchy is modeled in the cost formulas. Second, using the tool *perf* [1] we measured the number of data cache misses. This number is reduced by 98.2%, compared to the previous example (the non-cache conscious BNL join). However, the execution time does not reflect this significant improvement, since this experiment is I/O bound.[3]

[3]We have acquired the cache miss ratio of this experiment on a Linux server with an Intel Xeon E5-2620 CPU. The relative speedup between the Block Nested Loops join and the cache example remains the same.

Furthermore, by applying the partitioning rule from Section 6, OCAS is capable of transforming the Block Nested Loops Join into a variant of the GRACE hash join. Our experiments show that, as expected, the hash join performs better than the BNL join.

Next, we turn our attention to three examples where the output of the join is written to a device and not discarded.

When the output is written back to the same hard disk that stores the input, sequential reading from the hard disk is no longer possible, because the write operations interfere with the reads. Table 1 shows that the cost formula successfully depicts the considerably increased running time, even though the size of the input relations is significantly smaller compared to the original BNL join with no write-out.

If the memory hierarchy changes so that another hard disk HDD2 stores the output, reading and writing do not interfere with each other, so both can be executed sequentially. By doing so, even though the amount of data transfers remains the same, hard disk seeking is significantly reduced, as indicated by our cost formulas. As a result, Table 1 shows that both the estimated and the actual execution times are reduced by more than 50%. Note that we use the join condition "true" (thus we compute a relational product between the two relations) in the BNL join examples which write their output to a drive. Thus, write cost dominates read cost, which explains why the BNL join writing to a different disk is much slower than the BNL join discarding its output.

Finally, we consider an example with the same memory hierarchy as above, but a flash drive used in place of the second hard disk. In this case, OCAS generates the same program as before. However, both the estimated and the actual execution times are reduced due to the significantly better sequential write speed of SSDs. This is true, even though the factor of the InitCom events changes to depict their different meaning on flash. They do not correspond to seeks, but rather to an erasure occurring before each sequence of write operations, the length of which is given by the maxSeqW property of the flash drive. OCAS estimates better execution time for the example with flash and, thus, it accurately captures this trade-off between sequential writ-

ing and erase operations. The actual execution time of this experiment presents a similar behavior.

External Merge Sort. This example uses the fact that folding merge over a list of singleton lists of integers yields a sorting algorithm, and, thus, demonstrates how our rules for changing the folding patterns can bring us from Insertion Sort to a version of the External Merge-Sort. As a starting point, Insertion Sort can be represented as:

$$\mathsf{foldL}([], \mathsf{unfoldR}(\mathsf{mrg}))(\mathsf{R})$$

where mrg is the supporting function presented in Section 3 and the input is a list R of length x of singleton lists of integers. In this naive version, the elements are transferred one by one from HDD to RAM and back. The cost is:

$$\sum_{j=0}^{x-1}(\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + (j+1)(\mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}]))$$

Our system includes a basic engine for simplifying arithmetic expressions, capable of finding closed forms of some sums, which automatically simplifies the above formula to:

$$x\,\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{x(x+1)}{2}(\mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}])$$

By applying rule fldL−to−trfld, rule inc−branching and finally rule apply−block, we obtain 4-way External Merge-Sort. If we then apply rule inc−branching k − 1 more times, we get to 2^k-way External Merge-Sort, whose code is:

$$\mathsf{treeFold}[2^k]([], \mathsf{unfoldR}(\mathsf{funcPow}[k](\mathsf{mrg})))$$

The cost of running this program is, after simplification:

$$\left\lceil \frac{\lceil \log x \rceil\, x}{\mathsf{k}} \right\rceil (\mathsf{UnitTr}[\mathsf{RAM} \mapsto \mathsf{HDD}] + \mathsf{UnitTr}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{1}{b_{in}}\mathsf{InitCom}[\mathsf{HDD} \mapsto \mathsf{RAM}] + \frac{1}{b_{out}}\mathsf{InitCom}[\mathsf{RAM} \mapsto \mathsf{HDD}])$$

Our non-linear optimization solver determines that this cost is minimal when the all the input blocks and the output block are as large as possible, which means $b_{out} = b_{in} = \frac{m_S \cdot \mathsf{size}}{2^k+1}$, and hence the number of units transferred decreases with k and is proportional to $1/k$, while the amount of seeking *increases* with k and is proportional to $2^k/k$. Choosing the right k is again accomplished using the optimization solver and depends on the ratio between the seek-time and reading speed of the hard disk.

To sum up, the output algorithm of OCAS is always better than the specification algorithm provided by the user, and is adapted to the given memory hierarchy.

Manual inspection of the generated C programs shows that OCAS produces exactly the standard textbook (disk-based) BNL and hash join and external sorting algorithms.

7.3 Accuracy of Cost Formulas

General Overview In Table 1, we present the actual execution times of the generated C code for the optimized algorithms. As we can see, the estimates of OCAS are in general not far from the actual execution time, and in some cases our tool underestimates. This is because, as we explain below, the cost formulas are very simple and they do not completely represent the actual execution properties, especially for CPU dominated workloads. The important point to notice is that, when comparing equivalent algorithms like we did above in the case of BNL join with writing to different devices, the predictions follow the same trend as the actual execution times. Next, we examine two different aspects of accuracy in more detail: the effects of not modeling computation cost and of performing worst-case analysis.

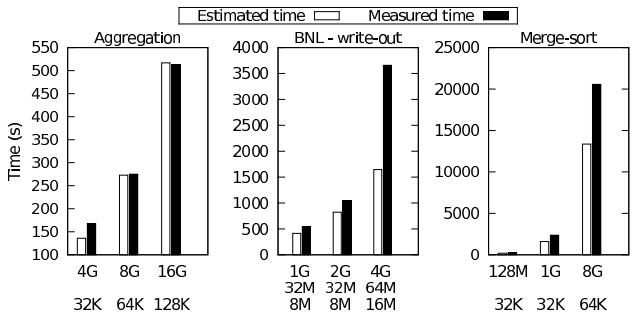

Figure 8: Estimated and actual running times for varying input and buffer sizes. The x-axis label shows the size of the first input relation, the second input relation (if applicable) and buffer size.

Impact of Computation Costs. OCAS does not currently model computation costs. This can cause our system to underestimate, as shown in Table 1. Moreover, underestimation should grow the more CPU intensive a task is. To examine this hypothesis, we run a set of experiments with a variety of different algorithms, input and buffer sizes. The results for these experiments, presented in Figure 8, confirm the initial assumption: For tasks that are not CPU-intensive, such as aggregation, the estimations are very accurate. However, for tasks like joins or sorting, which consume a significant amount of CPU cycles, underestimation grows with the input size. This raises the need of a more precise modeling of the CPU. However, we leave this for future work.

Impact of Worst-Case Analysis. The worst case analysis that OCAS performs can lead to significant overestimation of the output size, resulting in overestimation of the number of write operations. This is important, since this amount proportionally affects the reported estimations. To better understand this behavior, we present three examples in Table 1: one that calculates the union of sets represented as a sorted list of unique values, another that returns the union of multisets represented as a list of value-multiplicity pairs, and finally one that calculates their difference. For the union examples, the estimated output is equal to $\mathsf{length}(L_1)+\mathsf{length}(L_2)$ and for difference it is $\mathsf{length}(L_1)$. The latter follows because in the worst case there is no element that is the same amongst the two relations. The results of Table 1 show that there is overestimation due to predicting more write operations than those actually happening for the difference example. The union algorithm, however, is estimated correctly in both examples.

The join operator has a similar behavior due to selectivity: the higher the selectivity, the closer the estimation is to the actual running time. As Table 1 shows, with the selectivity of 100%, which corresponds to a relational product, the predictions become very accurate.

7.4 Running Time of OCAS

Table 1 presents the time required for OCAS to generate the optimized algorithms. As we can see, our tool is practical, since its execution time is small for all examples. We observe that the size of the search space depends on the number of steps needed for the derivation, the complexity of the input program, and the memory model used in the experiment. As expected, the search space is growing roughly exponentially with the number of transformation steps and the execution time is linked to the size of the search space. However, it is not dependent on the input size because OCAS uses cost-based optimization, which does not need to execute the programs in order to estimate their cost.

8. CONCLUSIONS

In this paper we describe OCAS, a code synthesizer that, given a memory hierarchy oblivious algorithm, automatically generates an efficient out-of-core version by exploiting characteristics of the provided memory hierarchy. OCAS applies transformation rules to a program, and exhaustively searches the space of generated equivalent programs to locate the one with the best performance metric. This metric is an estimation of the data transfers occurring at execution time. Our preliminary results show that OCAS adapts the generated algorithms to changes in the memory hierarchy and that it produces optimized versions of algorithms quickly. Its estimations are accurate when I/O cost dominates CPU cost. Otherwise, the underestimation increases proportionally with the CPU costs. This does not affect the correctness of the approach, as OCAS is able to always differentiate between more efficient and less efficient algorithms. Finally, OCAS efficiently performs estimation of parameters like buffer sizes, a task which is non-trivial for developers.

Acknowledgments

This work was supported by ERC grants 279804 and 306484.

9. REFERENCES

[1] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[2] Seeker: a utility to measure disk performance. http://www.linuxinsight.com/how_fast_is_your_disk.html.

[3] Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/.

[4] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *J. Automated Reasoning*, 46(2):161–203, 2011.

[5] P. Andreou, O. Spanos, D. Zeinalipour-Yazti, G. Samaras, and P. K. Chrysanthis. FSort: External sorting on flash-based sensor devices. In *Data Management for Sensor Networks*, 2009.

[6] L. Augusteijn. Sorting morphisms. In *3rd Int'l Summer School on Advanced Functional Programming, volume 1608 of LNCS*, 1998.

[7] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, April 1989.

[8] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Proceedings of 4th International Conference on Database Theory (ICDT)*, pages 140–154. Springer-Verlag, 1992.

[9] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *Proc. 16th European Symp. Algorithms*, 2008.

[10] W. Chin and S. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3), September 2001.

[11] N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proc. POPL*, January 2008.

[12] J. Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, page chapter 3. Palgrave, 2003.

[13] N. K. Govindaraju, R. Kumar, and D. Manochas. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proc. SIGMOD*, 2006.

[14] M. Hoffmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. POPL*, 2003.

[15] S. Jost, K. Hammond, H. Loidl, and M. Hoffmann. Static determination of quantitative resource usage for higher-order programs. In *Proc. POPL*, 2010.

[16] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. SIGMOD*, 2010.

[17] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C.-C. Lam, and J. Ramanujam. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. In *IPDPS*, page 34, 2004.

[18] Y. Liu, Z. He, Y. P. Chen, and T. Nguyen. External sorting on flash memory via natural page run generation. *The Computer Journal*, 2011.

[19] G. Liuzzi, S. Lucidi, and M. Sciandrone. Sequential penalty derivative-free methods for nonlinear constrained optimization. *SIAM Journal on Optimization*, 20(5):2614–2635, 2010.

[20] E. Meijer, M. Fokkinga, and R. Patterson. Programming with bananas, lenses, envelopes and barbed wire. In *FPCA*, 1991.

[21] H. Park and K. Shim. Fast: Flash-aware external sorting for mobile database systems. *J. Systems and Software*, 82(8):1298–1312, 2009.

[22] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[23] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd ed.* McGraw-Hill, 2002.

[24] M. Ren, M. Houston, J.-Y. Park, W. Dally, and A. Aiken. A tuning framework for software-managed memory hierarchies. In *Proc. PACT*, October 2008.

[25] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel and Distributed Computing*, 68(10):1381–1388, October 2008.

[26] V. Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Proc. ICALP*, pages 60–75, 1991.

[27] X. Ye, D. Fan, W. Lin, N. Yuan, and P. Ienne. High performance comparison-based sorting algorithm on many-core GPUs. In *Proc. IPDPS*, 2010.