Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels

Eric Love

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-246
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-246.html

December 17, 2015

Acknowledgement

This research is partially funded by DARPA Award Number HR0011-12-2-0016, the
Center for Future Architecture Research, a member of STARnet, a Semiconductor
Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab
industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in
this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels

by Eric Love

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Krste Asanović
Research Advisor

12/16/2015

(Date)

* * * * * * *

D A Patterson

Professor David Patterson
Second Reader

12/16/2015

(Date)

Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels

Abstract

Ressort: An Auto-Tuning Framework for Parallel Shuffle Kernels

by

Eric Love

Master of Science in Computer Science

University of California, Berkeley

Krste Asanović, Chair

This thesis presents *Ressort*, an auto-tuning framework for computational patterns that perform any kind of data-dependent data reordering or transformation. These programs, which we call *shuffle kernels*, account for large fractions of the runtime of database workloads and other applications. Hardware-conscious optimizations of shuffle kernels are all alike in that they generally consist of choosing one of many possible ways to decompose a particular kernel into a pipeline of more primitive operations: a sort might consist of a hash-based partitioning followed by an in-cache quicksort, or a join might entail sorting and then merging, for example. Ressort presents a domain-specific language (DSL) that enables the succinct expression of these compositions while also exposing the nested array-style parallelism that they imply, and supplies a compiler to exploit it. It also includes a parallel C-like intermediate representation that assists in the generation of performant shuffle kernel implementations. We present the design and implementation of these two language layers, and evaluate the performance of their output on a variety of hardware targets under different algorithmic requirements.

Contents

Acknowledgments

I would like first and foremost to thank my advisor, Krste Asanović, for supporting this project consistently from the beginning. More recently, Lisa Wu has contributed countless hours of advice, assistance, and guidance for which I am very grateful. Many thanks are due to Scott Beamer, whose tutelage in performance counter interpretation and OpenMP subtleties was invaluable. Finally, I would like to thank Dave Patterson for very generously reviewing this thesis.

This research is partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

Chapter 1

Introduction

Software spends a lot of time moving data from one place to another. Operating systems burn CPU cycles running `memcpy()` and `memset()`, while device drivers DMA swathes of bytes to and fro. Even though optimal implementations of `memcpy()` are tricky to code, due to the cumbersome quirks of packed SIMD instructions and subtleties of memory access alignment, the right thing to do for a given machine is generally surmisable.

Databases, more interestingly, spend their time moving records around in ways that depend on the contents of the records themselves: they sort them, partition them, build hash tables, perform joins, apply filters, and perform all manner of other transformations. Many non-database workloads make use of these algorithms too, so their efficiency is broadly important. However, they are very inefficient when implemented naïvely, and it is often far from obvious what sort of sophistication is needed, or even what the best *kind* of algorithm is.

Worse yet, all kinds of algorithms for these problems are tricky to optimize because their communication patterns cannot be pre-determined. Sorting an array might require moving the first element to the end, or leaving it in place. This pattern is quite different from mathematical kernels like matrix multiplication, where the programmer can reason about which elements combine with which other elements, and can statically order arithmetic operations so as to maximize the amount of inter-element communication that happens through the highest levels of the memory hierarchy.

This type of communication and movement distinguishes *shuffle kernels* from other computational patterns. We introduce and employ the term shuffle kernel to designate any kind of data-dependent data reordering, which is the focus of this study. Moreover, we attempt to enumerate the ways in which random interactions between shuffle kernels' inputs can be constrained and managed to achieve high performance on diverse hardware platforms.

This thesis describes specific shuffle patterns, algorithms, and their design spaces, and proposes a software framework to help programmers uncover the best strategies for any particular circumstance. This framework consists of a compiler for a domain-specific language (DSL) that describes shuffle patterns as compositions of the algorithmic building blocks we've examined.

1.1 Algorithmic Design Space of Shuffle Kernels

Many optimized algorithms have been proposed for each shuffle pattern in order to overcome the mismatch between these patterns' data-dependent data movement and hardware's ability to give peak performance only where there is locality and predictability. Decades ago, this meant minimizing the number of random disk I/Os needed to process a query; now, it means avoiding DRAM traffic, exploiting prefetchers, keeping hardware threads busy, and maximizing instruction- and data-level parallelism within a core.

Both then and now, a few common themes pervade the space of strategies for mapping shuffle kernels efficiently onto the physical reality of hardware:

1. If an algorithm requires too many random accesses to an array, *partition* that array into sets of related values. For example: hash joins send probes all over a large table, resulting in cache misses or random disk accesses, so instead divide up the inputs based on high-order bits of their keys and build separate, smaller hash tables for each bin [31, 38].

2. When multiple processors are available, divide up inputs among them either by:

   a) partitioning the input, as above, and processing partitions independently, (called *partition-then-split*) or

   b) by simply dividing up or *splitting* the original, unordered inputs at will, processing them separately at each node, and then merging the results at the end to restore order (*split-then-merge*)

Figure 1.1 illustrates the two structures of (2) as applied to a sorting algorithm. Each half of the figure actually depicts a *combination* of the parition-then-split and split-then-merge methods, applied in a different order in each case. Both expose parallelism at two levels, and both could themselves be component parts of a still larger operation.

Which strategy is better depends on many aspects of both the target hardware platform and the input data itself. If the data's keys are not uniformly distributed, partitioning won't result in even load balance across processors or cache blocks, but if they are, a large merge might be slow by comparison. If the target machine has more or fewer levels of parallelism (multiple sockets, cores, SIMD ways, etc.), more or fewer levels of parallelism should be exposed. If the memory system of the machine in question has high write bandwidth or can better tolerate random accesses than another, then the efficiency of a higher-fanout partitioning might change the tradeoff too.

One generalization holds across all these cases: *the most efficient implementation of any shuffle pattern on a particular platform will be a recursive composition of partitions, merges, and smaller instances of the target pattern.* However, given the large number of factors on which the optimum choice depends, *it is difficult to predict which combination of strategies is optimal.* Ideally, then, we would like to find the best algorithm empirically, but that would require codes with quite different structures and expressions of parallelism

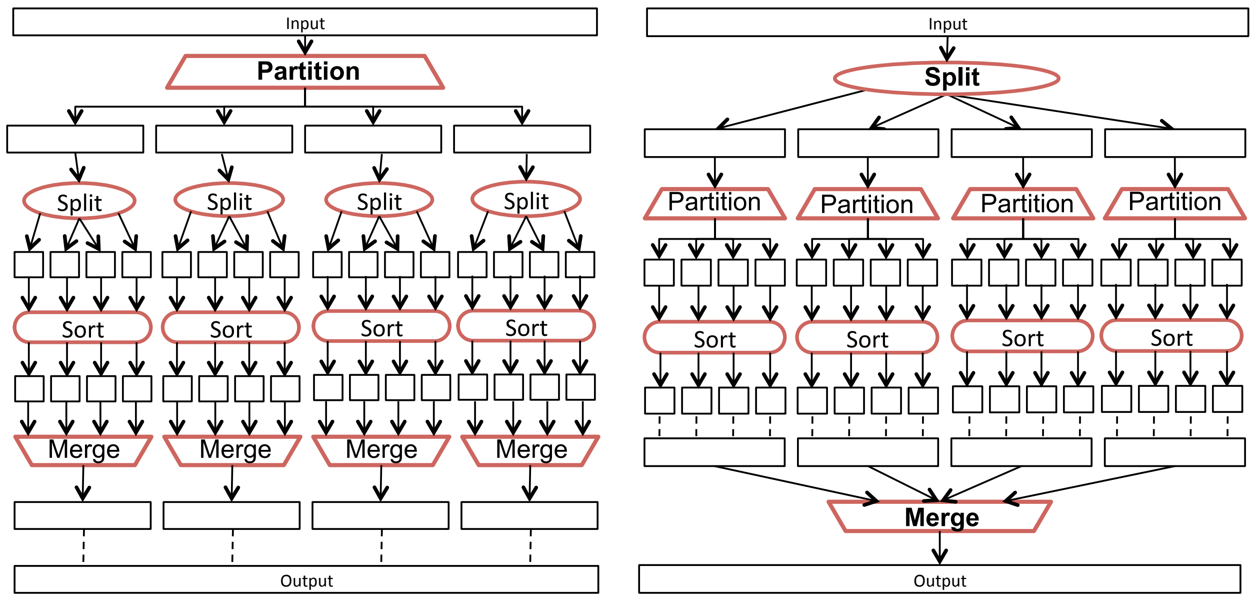


Figure 1.1: **Two Complementary Shuffle Algorithm Patterns** (for sorting in this example): On the left is the partition-then-sort strategy, and the right is the split-sort-merge strategy

for each combination of problem size, input type, algorithm, and machine. This approach is not practical to undertake manually, given the size of the search space and implementation effort required for each design point.

1.2 Auto-Tuning: High- and Low-level Tuning Spaces

The solution to this quandary is to recognize that the compositions of algorithms described above form a kind of language whose grammar defines how simpler algorithms can be combined to form more complex ones. If this language were sufficiently well-defined, then a compiler could be written to ease the burden of implementing each interesting combination. An automated tool could be written to generate expressions in that this language, and heuristics could be applied to prune the search space effectively.

This solution is the classic technique of *auto-tuning*, first proposed as a means of finding optimal cache blocking structures for DGEMM and other linear algebra kernels [7, 42], but subsequently applied to a wider variety of algorithms [34, 3]. *The purpose of the present study is to propose just such method for optimizing shuffle kernels.*

The algorithmic design space of shuffle kernels represents one level at which to apply auto-tuning methods—which we call the *high-level tuning space*—but some of the individual components out of which these algorithms are built have many possible concrete realizations of their own: they may support different code schedulings, parallelizations, loop unrollings,

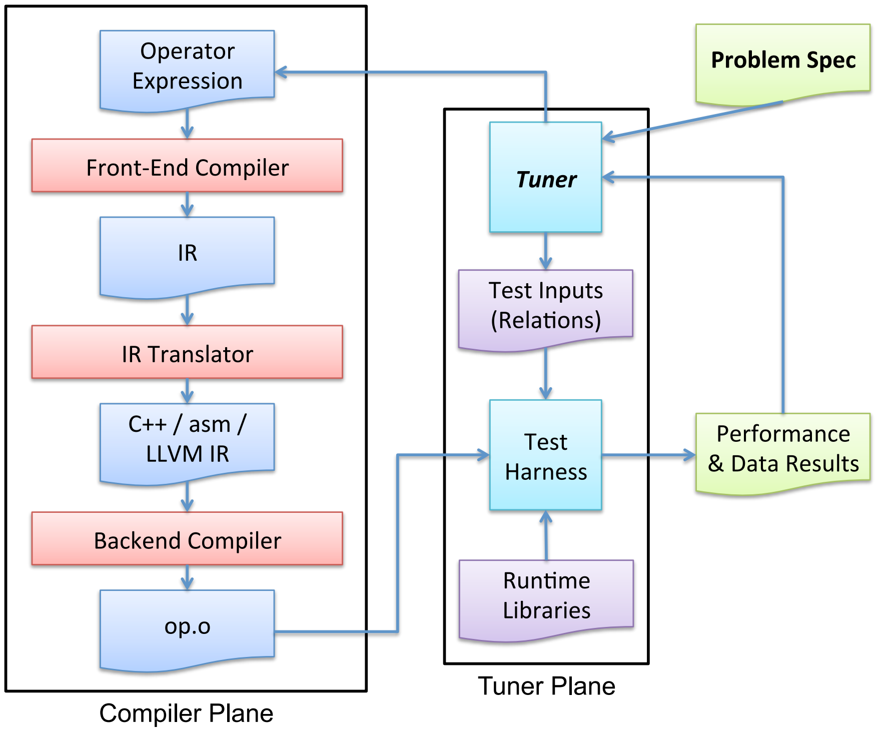


Figure 1.2: Ressort Compilation and Testing Loop

and so on. We call these parameters the *low-level tuning space.* To obtain real efficiency, tuning must be applied at both levels.

1.3 A Scala-based Framework for Shuffle Kernel Code Synthesis

This thesis presents *Ressort*[1], an auto-tuning framework for shuffle algorithms that spans both the high- and low-level design spaces. Figure 1.2 shows its organization. It comprises both a *compiler plane*, which generates code for different structural decompositions of shuffle algorithms, and a *tuner plane*, which generates these decompositions, and measures the performance of the compiler's output. Within the compiler plane, two domain-specific languages support tuning across both levels of design space. The *Ressort Operator Functional Language (ROFL)* expresses the algorithmic design space of Section 1.1, while the Ressort *intermediate representation IRep* supports parameterized code generation for each primitive algorithm implemented by ROFL.

[1] The name *Ressort* is intended to be read either in English as /ɹəsoɹt/ or in French as /ʁəsɔʁ/ ("spring, resilience")

Ressort's goal is to enable rapid evaluation of many different structural variants of shuffle algorithm implementations. While it does not yet interface with any external machine learning-based optimization or search tool, it does grant the programmer a means of quickly and concisely expressing any desired kernel decomposition, and of generating fast parallel code to implement that decomposition. The contributions of this paper are therefore:

- ROFL: a new DSL for describing shuffle patterns in a way that exposes their inherent parallelism
- A compiler for the ROFL language that emits parallel OpenMP/C++ code
- IRep, a C-like intermediate language that facilitates low-level tuning and code transformations inside the ROFL compiler
- An architectural study that characterizes the performance tradeoffs of ROFL primitive operations, and more complex algorithms built from them, on several diverse hardware platforms

The rest of this thesis is structured as follows: Chapter 2 reviews the fundamental algorithms out of which shuffle kernels are assembled, introduces a taxonomy of algorithms built on top of them, and discusses prior work on optimizing them. Chapter 3 presents the ROFL language as a Scala-embedded DSL, and shows how it can be used to construct performant shuffle algorithms for a variety of purposes. Chapter 4 examines the compiler we have built for the ROFL language, while Appendix C describes how it implements each primitive algorithm individually. Chapter 5 presents the IRep intermediate language, explaining its design decisions and its translation to C++. Finally, Chapters 6 and 7 analyze the performance of the ROFL compiler's output for sorting and partitioning on several different platforms.

Chapter 2

Background

This chapter surveys the space of shuffle kernels, reviews the various algorithms proposed for their implementation, and discusses previous work on optimizing their performance through hardware and software means.

2.1 Fundamental Shuffle Kernels

In this section we review the most fundamental building blocks of shuffle kernels, and discuss their implementation, parallelization, qualitative performance characteristics, and memory overheads. Our discussion follows the taxonomy of algorithmic primitives given in Figure 2.1. It is these units which Chapter 3 explicitly composes to form complex, performant operators, but we simply note that any of the partitioning algorithms (green) can serve as inputs to a sort algorithm (red), and both of those may appear in a complex join (purple).

Notation

The descriptions of shuffle kernels throughout the rest of this section all assume a *relation* consisting of an array of *records* is to be reordered or manipulated in some way. Here the term record refers to some bytes of data in an arbitrary format of which some bits are designated as the *key*. Most shuffle kernels concern only the key bits of each record, which determine where that record is to be relocated to or how it generally is to be treated. Within a given relation, all records are assumed to have the same data type, and have the same subset of bits used as the key.

The following variables and abbreviations are assumed throughout the rest of this discussion:

- R, R_1, R_2, etc. represent *relations* used as input to some shuffle algorithm
- $|R|$ is the number of records contained in relation R
- $R[n]$ is the nth record of relation R

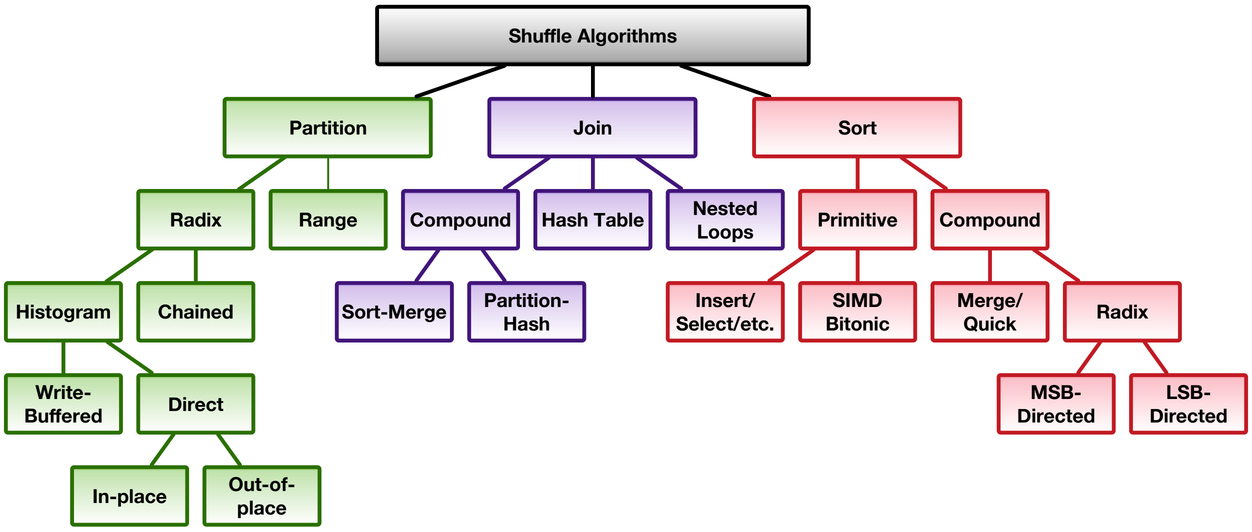


Figure 2.1: **A Taxonomy of Shuffle Algorithms**: Partitioning (green), sorting (red), and joins (purple) form the three main categories of shuffle algorithms. The leaf nodes of the partitioning branch can be fed into some members of the sort branch, and building blocks from both of these categories can be used to perform a join.

- $K(R[n])$ is the *key* of record $R[n]$
- b is a number of *radix bits* used in a radix sort or partition
- T is the number of *threads* used in a parallel algorithm

Partitioning

Partitioning is arguably the most important shuffle kernel. It divides input records into separate *buckets* or *partitions* based on the value of some bits of their keys. This kernel is a preliminary step in some possible implementations of all shuffle patterns, including sorting, joins, hash table building, and many others.

Types of Partitioning

Radix- vs. Range-Based Partitioning can be either range-based or radix-based. *Range-based* partitioning uses a sorted array of *splitter values* or *delimiters*, and assigns records to buckets defined by the range of keys between any two consecutive splitter values. *Radix-based* partitioning simply examines a fixed subset of each key's bits to assign its bucket. The number of bits examined is called the *radix*.

Histograms vs. Chaining In both radix- and range-based partitioning it is necessary to determine how much memory is reserved for each output partition. This reservation cannot be known *a priori*, so either (1.) partitions must be allowed to expand dynamically by adding new blocks of records to a linked list or other expandable data structure, or (2.) the input relation must be read twice: once to determine the sizes of each partition, and once more to actually move the records. The former is called *chaining* because it chains together blocks of records for each partition, while the latter is called *histogram-based* because it constructs a histogram that counts partitions' sizes.

We presently consider only histogram-based radix partitioning. Although range-based partitioning is an important component of many algorithms, we defer its investigation to future work. We also postpone consideration of chaining because it complicates code generation.

Parallelization

Partitioning is inherently parallelizable. Since no inter-record comparisons are needed, all inputs are read and examined independently from each other. Synchronization is only needed to ensure threads output records to disjoint regions of each bucket. In a histogram-based method, this can be done two ways: the *shared histogram* approach requires an efficient atomic increment so that threads can update histogram counters safely and independently, while a *shared nothing* approach creates a separate histogram for each thread, and determines per-thread partition sizes during the initial counting phase. Shared histograms do not scale well, and few hardware platforms support suitably efficient atomics, so we consider only shared-nothing algorithms.

Algorithm 1 on the next page gives a complete definition of the shared-nothing parallel histogram-based radix partition algorithm employed all throughout this study. It consists of four main steps:

1. **Histogram counting** (lines 2 to 10): Build a histogram array for radix r of size $2^r \times T$ and count the number of records in each partition for each thread

2. **Prefix sum reduction** (lines 11 to 18): Do a *prefix sum* operation to transform the per-partition counts into *offsets* at which each of the per-thread partitions start

3. **Record Movement** (lines 19 to 28): Read through all the input records a second time and write them to their corresponding partitions at the offsets given by the histogram

4. **Histogram Merger** (lines 29 to 33): Merge together the per-thread histograms into one unified view to return

Computational Characteristics of Partitioning

Partitioning is a computationally inexpensive kernel. It requires only a few tens of instructions per record on most architectures, and has completely regular control flow. Its performance is constrained mainly by memory bandwidth and, at high radices, scattered writes

Algorithm 1 Parallel Radix Partitioning

```
1: procedure RadixPart($R$, msb, lsb, $T$)
2:     $r \leftarrow$ msb $-$ lsb
3:     Hist$[0..T][0..2^r] \leftarrow 0$
4:     for $t < T$ do                                ▷ Build a Histogram
5:         for $i < |R|/T$ do
6:             rec $\leftarrow R[i + t|R|/T]$
7:             part $\leftarrow K$(rec)[msb : lsb]
8:             Hist$[t]$[part] $\leftarrow$ Hist$[t]$[part] $+ 1$
9:         end for
10:    end for
11:    $a_1, a_2 \leftarrow 0$   ▷ Perform a prefix sum reduction across all threads' private histograms
12:    for $p < 2^r$ do
13:        for $t < T$ do
14:            $a_1 \leftarrow$ Hist$[t][p]$
15:            Hist$[t][p] \leftarrow a_2$
16:            $a_2 \leftarrow a_1$
17:        end for
18:    end for
19:    Output$[0..|R|] \leftarrow \emptyset$          ▷ Distribute records to their corresponding partitions
20:    for $t < T$ do
21:        for $i < |R|/T$ do
22:            rec $\leftarrow R[i + t|R|/T]$
23:            part $\leftarrow K$(rec)[msb : lsb]
24:            offset $\leftarrow$ Hist$[t]$[part]
25:            Output[offset] $\leftarrow rec$
26:            Hist$[t]$[part] $\leftarrow$ offset $+ 1$
27:        end for
28:    end for
29:    Hist$'[0..2^r] \leftarrow 0$                    ▷ Merge together per-thread histograms
30:    for $1 < i < 2^r$ do
31:        Hist$'[i] \leftarrow$ Hist$[i-1]$
32:    end for
33:    Hist$'[i] \leftarrow 0$ return (Output, Hist')
34: end procedure
```

distributed randomly across a possibly very large output buffer. Section D.1 describes this access pattern in more detail, and presents a simple analytic model for the amount of DRAM traffic generated by partitioning.

The write pattern exhibited by the record distribution stage of the partitioning algorithm can be thought to consist of 2^rT *independent streams*: within each thread's share of each partition, the next record is always written to the next sequential address, but these streams advance independently of each other, and the next stream to be used is determined randomly by the key of each input record.

Since for large relations these streams will be distributed evenly across a large virtual address space, the TLB working set size will exceed the hardware's capacity and TLB misses will impact performance unless large pages are employed. High-radix partitioning's random write pattern also increases last-level cache (LLC) misses by lowering the likelihood that a given cache line in the output buffer, when read in from DRAM on a write miss, will be written to again before it is conflict-evicted by another stream that maps to the same set. For this reason, it is sometimes optimal to allocate an *output buffer* that contains one cache line worth of records for each stream in a contiguous array [24]. Only when a line in this buffer is completely filled will it then be written to the main output. Another way to mitigate the impact of high-fanout is to perform partitioning in multiple passes, where each pass uses a radix that is a fraction of the final desired one. This compromise requires scanning the input relations multiple times, but avoids to problems of high fanout.

Sorting

That sorting is fundamental not just within shuffle kernels but among all algorithms is attested to by the long list of methods commonly taught to undergraduates: quicksort, merge sort, heap sort, insertion sort, radix sort, and so on [14]. In the space of shuffle algorithms, sorting's importance is even greater, since it is a building block of many other operations one might wish to implement. It is the first step of a sort-merge join, a component of set intersection, and in the case of recursive algorithms, one sort's output is an input to another.

The space of all possible sorting algorithms is much too vast to review here; instead, we merely make note of those most relevant to Ressort's currently supported library of shuffle kernel primitives, and those it is likely to support soon.

Two Kinds of Sorts

Sort algorithms are traditionally categorized as either *comparison-based* or not: insertion sort and merge sort compare pairs of values and reorder them accordingly, whereas radix sort never explicitly compares anything with anything else.

In the context of Ressort, we consider two other broad classes of sorting algorithms: (1.) *leaf-node* sorts, or *primitive sorts*, and (2.) *composite* sorting algorithms. The former class includes those algorithms which most efficiently process small (order ten or so) input elements, and so might serve as building blocks to algorithms in the latter category, which employ some strategy to break up their inputs into smaller chunks, and then eventually invoke a primitive algorithm when these chunks are sufficiently small. Thus, we would say that the primitive algorithms reside at the "leaf nodes" of those composite algorithms' call graphs.

Leaf-node Sorts

Among the more primitive kinds of sorts are *insertion sort*, which has asymptotically quadratic runtime but requires fewer operations at small problem sizes than do more complex methods. We include it as a candidate primitive algorithm in our study for just this reason, and discuss its computational properties elsewhere (Section C.4 and D.2). It is often used at the lower levels of recursive algorithms like merge sort, and the size at which to switch over to insertion sort is a tuning parameter of such algorithms.

Sort networks, such as bitonic or Batcher sort [6], offer an alternative strategy for small input sizes. These exploit the fact that some algorithms such as merge sort can be made to have a fixed set of comparisons that does not depend on the values of the input data, and so all these comparisons can be spelled out in advance of execution, either as explicitly parallel hardware, or as static software instructions. Though they require more total work than other algorithms, they are valuable for their parallelizability. Recent work has shown such algorithms to be amenable to implementation with SIMD instructions [24, 36].

Composite Sorts

Composite sorting algorithms are those that chain two or more primitive algorithms together. We further divide composite sorts into two categories: *split-merge* and *partition-sort* algorithms.

Split-merge sorts are those that first break up their inputs in a data-independent manner, sort all segments independently, and then merge them together somehow. Classical merge sort is in this category: implementations might sort chunks of ten records with insertion sort, and then merge them together. Another example would be cache-blocked radix sort, wherein inputs are first evenly divided into cache-sized blocks, each of which is then sorted with radix sort independently before being merged with the other sorted blocks. *Partition-sort* algorithms, on the other hand, apply a data-dependent partitioning operation (Section 2.1) to their inputs, and then sort the resulting partitions without the need for a final merge step. Quicksort is a partition-sort, because it first range-partitions its input by comparing all keys to a splitter value, and then independently sorts the resulting sets.

A special case of partition-sort algorithms is radix sort, which radix-partitions its input repeatedly until all bits of the input keys have been used. Radix sort (described more formally by the program listing in 3.4) has itself two variants. In LSB sort, records are partitioned first using the least significant bits and then re-partitioned using successively more higher-order bits. The partitioning must be *stable* in this case, meaning that records sent to the same partition appear in the same order with respect to each other in the output as they did in the input. By contrast, MSB sort begins partitioning at the most significant bits, and subsequently sorts each resulting partition in parallel. The partitioning operator need not be stable.

One additional—and orthogonal—dimension along which composite algorithms may be categorized is whether they are *dynamically-structured* or *statically-structured*. Dynamically structured algorithms follow a divide-and-conquer strategy and recursively assemble sorted sub-problems until the entirety of the input has been processed. They can react to situations where one sub-problem is too large (resulting in load imbalance) by dynamically recursing once more to split it it up. Statically-structured algorithms determine the number and order of processing of sub-problems in advance.

Merge sort is always implemented as a dynamically-structured algorithm, but doesn't have to be: insertion sort-sized sub-arrays could all be sorted at once, and then a fixed-depth merge tree could be imposed on top of them. Radix sort is statically-structured by definition, but radix-partitioned radix sort could be implemented dynamically to continue partitioning some buckets that, due to key skew, are too large to fit in the intended level of cache. Quicksort is a dynamically-structured partition-sort algorithm, and cannot be made static, since the recursive call tree's structure is data-dependent, unlike merge sort's, which depends only on the size of the input. Sort networks are always statically-structured and cannot be made dynamic.

Computational Characteristics of Sort Algorithms

In general, split-merge algorithms differ from partition-sort algorithms in that they trade off increased computational intensity for better memory hierarchy interaction. Partition-sort algorithms usually incur the wasted memory bandwidth of random accesses when the inputs are large and the radix is high, but execute few dynamic instructions per record and provide ample parallelism. Split-merge algorithms start off naturally cache-blocked, and only ever need to make sequential memory accesses, but they require asymptotically more work ($O(n \log n)$ vs. $O(n)$) and have less parallelism in the final merge stages.

Dynamically-structured algorithms like merge sort can be naturally *cache-oblivious* [18], as they maintain cache-sized working sets without needing any explicit blocking. They are less sensitive to input skew and achieve better load balance, but not all desirable algorithms can be dynamically structured.

For small input sizes, which can be processed entirely in cache, comparison sorts stress core microarchitecture with data-dependent branches and stores to the same address with different data on the taken and not-taken branch paths. Non-comparison sorts do not have this problem. Some comparison sorts can be implemented with predicated or conditional move instructions to avoid incurring load-store ordering misspeculations.

Joins

Though we do not investigate them in the present study, *joins* are some of the most common operations in analytic database systems, and account for more than half of all execution time in many queries [19]. They are almost always implemented with composite algorithms of the kind described above, so we review their design space here briefly. Formally, a join specifies the computation implied in Definition 1:

Definition 1 *A* join $R \bowtie S$ *of an* outer relation R *onto an* inner relation S *is the set of all pairs of records* $(r_i, s_j) \in R \times S$ *such that* r_i *and* s_j *have keys that match. If the condition for a match is that the keys are equal, it is called an* equijoin*.*

The simplest way to find all pairs of tuples that meet Definition 1 would be to compare each record in the outer relation to each record of the inner relation and output those pairs whose keys match. This method is known as *nested loops join (NLJ)* and requires $O(N^2)$ comparisons for two relations of approximately equal length N, so different methods are required for all but the very smallest relations.

The most common form of join for larger input sizes is the *hash join*, which is so called because it entails building a *hash table* of the inner relation, and then *probing* it with records from the outer relation. The hash join reduces the asymptotic complexity to $O(N)$, but is still sub-optimal on real hardware. In fact, it shares many of the same bottlenecks as partitioning: if the number of entries in the hash table is too high, then random probes to a large hash table will frequently miss in cache [38]; if it is too small, then a large constant will be hidden in the asymptotic bound because collisions will result in more comparisons on every probe.

As a result, two main categories of improved join algorithms have emerged: *partitioned hash joins* and *sort-merge joins*. In the case of the former, both input relations are first radix- or range-partitioned with the same partitioning function, and then much smaller hash tables are built from the partitions of the inner relation, each one of which is probed only with records from its corresponding partition in the outer relation. In the case of a sort-merge join, both relations are first sorted, and then scanned sequentially in tandem. Whenever the key of a record in the inner relation is found to be greater than that of the currently-examined record in the outer, the scan pointer in the outer is advanced by one. By maintaining the invariant that records examined in the inner relation always have keys less than or equal to the scan pointer in the outer relation, it is guaranteed that all possible matches will be found.

The ensemble of techniques proposed in these last two methods shows the relevance of our compositional approach to the problem of joins. A partitioned hash join composes a partition operator with a hash table build and a probe (really nested loops join), the latter two stages of which may proceed in parallel for each of the initial partitions. In a sort-merge join, any of the compound sort techniques presented above—including those which themselves utilize partitioning—may serve as the input to the join.

2.2 Related Work

We briefly review some of the literature associated with the main areas of work related to Ressort: in-memory databases, shuffle algorithm optimization and hardware acceleration, software auto-tuning, and parallel programming language systems.

Parallel, In-Memory Database Systems and Query Compilation

Ressort's very existence is predicated on the emergence of memory-resident, data analytics-oriented column store database systems. Since nearly a decade ago, researchers have focused their attention on the technical challenges that have resulted from the desire to run databases entirely out of main memory, which fundamentally altered the nature of performance bottlenecks. Instead of minimizing the number and frequency of disk I/Os, engineers now needed to make more efficient use of CPU and cache hierarchy resources. Additionally, the compute-intensive and mainly read-only queries typical of this new era motivated a shift away from optimizing transaction throughput and towards accelerating large aggregation operations on vast tracts of data.

C-Store [40, 26] was one of the pioneering database architectures developed in response to these trends. It proposed organizing relations in *columns* rather than rows. This layout arranges each table as several separate arrays in memory, each one of which contains a single attribute for all records. This facilitated the parallelization of databases across clusters of nodes, allowing a single node to contain all entries of a particular column and, potentially, to examine them all at once without intervening communication. Other modern column-stores, such as MonetDB [12, 47, 48], have exploited the columnar organization to find intra-node parallelism. Whereas traditional query execution engines relied on a "tuple-at-a-time" approach that interprets entire queries for a single record before processing another, columnar storage enabled *vectorized*[1]processing, or the evaluation of a single operator across a block of records at a time. This eliminated interpretive overheads and exposed instruction-level parallelism (ILP) to the compiler and to superscalar architectures.

Other researchers have also sought to improve ILP and general computational efficiency in analytic workloads. Some did so by compiling queries to native machine code instead of interpreting them dynamically [39, 32]. Meanwhile, Shatdal et al. [38] worried already in 1994 about how best to utilize on-chip caches during query execution. The concerns they raised reverberate throughout all the algorithmic optimizations described below and throughout the rest of this thesis.

[1]This should not be confused with vectorization in the vector architecture sense, which is its own area of investigation [22].

Shuffle Algorithms for Modern Architectures

Joins

Many modern shuffle kernel optimizations build upon early join work from database systems in the 1980s. Several papers from that decade described how partitioning could be used to reduce disk seeks in hash joins [16, 25], as well as to parallelize them.

Subsequent research has re-evaluated these strategies in the context of modern in-memory databases. In 2009, Kim et al. [24] revisited the question of whether hash joins or sort-merge joins were faster on current hardware. They ultimately concluded that hash joins (based on the shared-nothing parallel partition method of Algorithm 1) would yield better performance for the time being, but would fall behind as the compute capacity of CPUs with wide SIMD increased faster than memory bandwidth. Albutiu et al. [1] also proposed a merge sort-based join algorithm, Massively-Parallel Sort-Merge (MPSM), that specifically targeted NUMA machines. It works by assigning each socket to sort its local NUMA segment of both input relations independently, and then performing a massive merge-join in which each processor accesses all other sockets' NUMA domains, but does so *sequentially* in order to benefit from prefetching.

Blanas, Li, and Patel then argued [8] that, actually, an even simpler partition algorithm based on atomic updates to a shared histogram table was more efficient. Later work by Balkesen et al. [3, 4] contradicted the claims made by both Kim and Blanas: first, they argued that sort-merge would only outperform radix-hash if relations were very, very large, and second, they determined that the purported performance advantage of the shared histogram approach arose from the use of pre-sorted data in the inner relation, which eliminated access contention [5].

Joins do have inherent data parallelism, and Rich Martin's 1996 technical report [29] showed how to vectorize them for Cray-style architectures. His algorithm derives from Zagha and Blelloch's vectorized radix sort [45]. The latter work proposed a vectorized partitioning technique (*loop raking*) that is *stable*, meaning that records assigned to the same partition appear in the same order as they did in the original input relation. LSB radix sort requires this property for correctness.

Partitioning

More recent work by Polychroniou and Ross [33] thoroughly examined a wide variety of radix- and range-partitioning algorithms, contributing to and enumerating an already vast taxonomy of techniques. In particular, they supply *in-place* and *out-of-place* variants of both Algorithm 1 and the shared histogram approach presented in [9]. They also explained how to exploit SIMD instructions to improve range partitioning performance for small and large numbers of delimiter keys. They demonstrated and explained the advantage of range partitioning over a radix-based approach when used as an input to subsequent steps of a shuffle kernel, which is that skewed key distributions are less problematic, and that a careful choice of delimiters can ensure consistently cache-sized partitions at the output.

Sorting

Satish et al. made the same argument in [37] and [28] as in their previous join paper [24] that radix sort currently outperforms merge sort, but that the balance will tilt in favor of "bandwidth-oblivious" merge sort as SIMD widths become wider and key sizes increase to accommodate ever larger in-memory relations. They implemented the then fastest radix sort using an output buffer-based (Section 2.1) partition operation, and measured a sorting throughput of 250M keys/second on a 4-core, 3.2GHz Intel Core i7 machine using 32 bit keys. They claimed that because radix sort is memory bandwidth-bound, its performance will not improve on future hardware.

Therefore, they also implemented a merge sort algorithm using a SIMD sorting network that achieves roughly comparable throughput to radix sort. Because this method is currently compute-bound, and because its bandwidth requirements are much lower than those of radix sort, its performance will dominate on platforms with greater per-core compute resources. Their parallel merge sort implementation builds on a previous algorithm for GPUs [36], and parallelizes the merger of large lists by finding splitter values, as did [17].

In the same paper as their partition study, Polychroniou and Ross also examined how to build faster radix and merge sort implementations on top of their new partitioning primitives. They use range partitioning to divide keys evenly across NUMA domains, and then apply MSB-, LSB-, and comparison-based sorts within NUMA partitions. They found that the best algorithmic decomposition depended on relation size, key size, and skew, even while the hardware platform remained constant. We feel this dependency, combined with their expansion of an already large taxonomy of operators, motivates the design of Ressort. As different systems offer different balances of compute to memory bandwidth, cache hierarchy designs, and degrees of data parallelism, the choice of optimal algorithm becomes even less intuitive. An efficient means of generating parallel code for many distinct design points will ultimately provide the best means of realizing optimal performance on new platforms.

Hardware Support

Other researchers have sought to improve shuffle kernel performance with the support of specialized hardware ranging from general-purpose data parallel processors to fixed-function accelerators. Kaldewey et al. presented efficient hash join algorithms for GPUs [23]. Jiong He, Shuhao Zhang, and others explored the mapping of database systems onto CPU/GPU hybrid platforms [21, 46]. Meanwhile, Timothy Hayes et al. proposed a vector architecture specifically targeted at improving hash join and sort performance [19]. They also presented a new vector sort algorithm and supporting instruction set additions designed to avoid duplicating histograms once per vector element during partitioning [20].

Other researchers have proposed fixed-function accelerators for particular shuffle kernels. Lisa Wu et al. designed the HARP [43] accelerator to speed up range partitioning, and then integrated it into their Q100 query processing architecture [44]. The latter's pipelines offered an assortment of dedicated units for different shuffle kernel primitives such as filtering, partitioning, joining, and aggregation, and provided a dataflow programming model for scheduling query execution on top of them.

Auto-Tuning

Origins: Linear Algebra

The idea of using an automated script to empirically choose an algorithm's optimal implementation on a particular platform arose in the scientific computing community. "Portable, Hi-Performance ANSI C" (PHiPAC) by Bilmes et al. [7] applied this method to matrix multiplication and other numerical kernels in the mid-1990's. Their software searched the space of possible cache-blockings and register panelings for DGEMM, outputting highly stylized C code for which even the least reliable of compilers could generate performant assembly. Nearly two decades later, improvements in alias analysis have rendered many of these defensive coding strategies irrelevant, but the need to automate the search for the best of many structural code variants remains strong. Subsequent work by Whaley and Dongarra, "Automatically Tuned Linear Algebra Software" (ATLAS) was used by several numerics libraries. More recently, the OSKI [41] extended this approach to sparse matrices, and has been widely deployed.

Auto-Tuning in Other Domains

The auto-tuning methodology has also been leveraged in several domains outside of linear algebra. The Halide [35] project defined a DSL for describing stencil operations on neighboring pixels in graphics kernels, and supplied a compiler to produce and evaluate the performance of different operator pipeline schedules. Meanwhile, the SPIRAL project's [34] approach to optimizing FFTs inspired Ressort's treatment of shuffle algorithms. It exploits the fact that there are many possible mathematical decompositions of the FFT kernel, and introduces an algebra to express different points in this space. Part of their auto-tuning process involves sampling various such formulae. Ressort's ROFL language is thus akin to their Signal Processing Language (SPL).

Parallel Programming Languages and DSLs

Ressort's ROFL programming language is also very much indebted to NESL [11]. The latter work presented in the early 1990's a language for expressing portable parallel algorithms as parallel operations on nested arrays. This facilitated compilation of NESL code to exploit hardware ranging from SIMD units accessed with vector instructions up to cluster machines programmed via MPI. The shuffle patterns targeted by Ressort map very naturally onto this paradigm, and we hope eventually to cover a similar breadth of scales.

The Ressort compiler's C++ backend leverages the parallel OpenMP [15] extensions for C and their accompanying language runtime. The OpenMP framework permits the expression of thread- and data-level parallelism by *parallel regions*. When a C++ code block is tagged with OpenMP's `omp parallel` pragma, all threads spawned by the runtime will execute the block concurrently. More importantly, a parallel pragma applied to a for loop indicates that there are no dependencies between successive loop iterations, and causes the runtime to divide the iterations among threads.

2.3 Summary

The three main flavors of shuffle kernels—partitions, sorts, and joins—have each been the subject of virgorous investigation, and the simpler implementations of each have been deployed in more complex versions of the others. At the same time, architectural and microarchitectural differences between platorms, as well as differences in the representation, size, and key distribution of relations, can all change the tradeoffs that make one strategy more viable than another. We consider these two facts as motivation for the design of a single programming language that compactly expresses shuffle algorithms and enables algorithmic design space exploration on current and future architectures.

Chapter 3

The Ressort Operator Functional Language (ROFL)

As Chapter 1 argues, the most performant implementations of shuffle algorithms are usually recursive compositions of more fundamental shuffle operations. Since the best strategy to choose at each level of composition is non-obvious, as are the optimal number of levels and the parameters to each level itself, the only way to achieve peak performance on any system is to experimentally evaluate many different points in this space. It is the goal of this project to enable just such a design space exploration. We present the *Ressort Operator Functional Language (ROFL)*, which allows the compact expression of shuffle algorithms. It exposes the nested-array type of parallelism induced by partitioning or slicing arrays of records, but moves the burden of exploiting it from the programmer to the compiler. By representing shuffle kernel compositions at a higher level than fully spelled-out C++ code, it allows the compiler to transform, reorder, parallelize, and fuse together these operations in ways that would not otherwise be possible.

This chapter describes the ROFL language in more detail. We implement ROFL as a domain-specific language (DSL) embedded in Scala. Scala's rich syntax supports adding layers of syntactic sugar to construct ROFL abstract syntax trees (ASTs) directly and without the need for a ROFL-specific parser. Moreover, by hosting it in the same language as its compiler, we allow for the expression of ROFL *program generators* that look themselves like ROFL primitive operations but in fact assemble and return more sophisticated compositions thereof. Instances of this are shown later in the chapter. For now, we introduce the ROFL language with some simple examples.

3.1 A Simple ROFL Program

ROFL allows a programmer or auto-tuner (hereafter *the user*) to compose *operators*, which transform arrays of records into reordered, and reorganized arrays of the same or other records. Operators are really functions, in the mathematical or functional programming sense, but we use the term operator to avoid confusion with Scala or C++ functions. The output of the Ressort compiler is a C++ function that implements the semantics of its original ROFL operator on any input arrays passed to it.

The following is the 'Hello World' of ROFL codes:

```
1 val A: Operator = OuterRel // Defined by ROFL to represent an input array
2 val myOp: Operator = Flatten(InsertionSort(Split(A, Length(A)/Const(10))))
```

These two lines of Scala declare a ROFL operator, `myOp`:

- Line (1) defines the `A` symbol to reference a ROFL language primitive `OuterRel`, which is itself defined to be the first input array passed as an argument to the C++ function that results from compiling this program.
- Line (2) actually specifies an operator to be compiled. Though it may look like a nested function call, it is in fact a declaration of a ROFL AST that Ressort will process later.

Compiling this two-line ROFL program results in the OpenMP-based parallel C++ code shown in Figure 3.1. More specifically, it generates the `InsSrt_SplitPar_Rel()` function, which implements the insertion sort operation in parallel across the `Length(OuterRel)/10` segments of an input array of records, `OuterRel`. The auto-generated code is unsurprisingly obtuse, but its structure is still straightforward:

- Lines (3-7): Allocate a new output buffer
- Lines (10-15): Invoke threads to handle segments in parallel
- Lines (33-50): Implement insertion sort with two nested loops

3.2 Kinds of ROFL Operators

A ROFL operator is a combination of one or more *ROFL primitives*, or operations for which the compiler knows how to generate code directly. A simple example is `InsertionSort()`, which denotes the result of applying the insertion sort algorithm to all the records in its input array.

Most ROFL primitives expect as input a flat array of records. Such operators are called *canonical operators*. Whenever a canonical operator receives a nested array as input, an *implicit* `map()` *operation* occurs, and the operator is applied to all sub-arrays independently. By design, these semantics resemble those of `map()` in functional programming languages, which applies a supplied function to each element of an array independently.

The example in Figure 3.1 describes a computation, or modification of an input array; other primitives imply no computation, but instead specify a *change in the structure* of their

```
struct _arr_urec__UInt32_UInt32* InsSrt_SplitPar_Rel(struct _arr_urec__UInt32_UInt32* OuterRel)
{
  struct _arr_urec__UInt32_UInt32* r_OuterRel_ref_copy_ptr_0;
  r_OuterRel_ref_copy_ptr_0 = new struct _arr_urec__UInt32_UInt32;
  r_OuterRel_ref_copy_ptr_0->len = (OuterRel->len)+((0)*((OuterRel->len)/(10)));
  r_OuterRel_ref_copy_ptr_0->items =
      new struct Rec_uint32_t_uint32_t_[r_OuterRel_ref_copy_ptr_0->len];
  const size_t r_nslice_9 = (OuterRel->len)/(10);
  const size_t _CTMP1_r_OuterRel_offset_5_max = r_nslice_9;
  #pragma omp parallel
  {
    #pragma omp for
    for( size_t r_OuterRel_offset_5 = 0;
         r_OuterRel_offset_5 < _CTMP1_r_OuterRel_offset_5_max;
         r_OuterRel_offset_5 = r_OuterRel_offset_5+(1))
    {
      const size_t r_nslice_9 = (OuterRel->len)/(10);
      size_t r_slen_6, r_winsz_7, r_boff_8;
      const size_t r_nslice_13 = (OuterRel->len)/(10);
      size_t r_slen_10, r_boff_12;
      r_slen_6 = ((OuterRel->len)<r_nslice_9) ?
          (((OuterRel->len)>(0)) ? (1) : (0)) :
          (((OuterRel->len)+(r_nslice_9-(1)))/r_nslice_9);
      r_boff_8 = r_OuterRel_offset_5*r_slen_6;
      r_winsz_7 = ((r_boff_8+r_slen_6)<(OuterRel->len)) ?
          r_slen_6 : ((OuterRel->len)-r_boff_8);
      r_slen_10 = (((OuterRel->len)+((0)*((OuterRel->len)/(10))))<r_nslice_13) ?
          ((((OuterRel->len)+((0)*((OuterRel->len)/(10))))>(0)) ? (1) : (0)) :
          ((((OuterRel->len)+((0)*((OuterRel->len)/(10))))+(r_nslice_13-(1)))/r_nslice_13);
      r_boff_12 = r_OuterRel_offset_5*r_slen_10;
      const size_t r_isort_numEntries_20 = r_winsz_7;
      if(r_isort_numEntries_20>(0)) {
        const size_t _CTMP2_r_isort_i_14_max = r_winsz_7;
        for( size_t r_isort_i_14 = 0;
             r_isort_i_14<_CTMP2_r_isort_i_14_max;
             r_isort_i_14 = r_isort_i_14+(1))
        {
          struct Rec_uint32_t_uint32_t_ r_isort_tmp_17 =
              (OuterRel->items)[r_boff_8+r_isort_i_14];
          size_t r_isort_j_15 = r_isort_i_14;
          while( (r_isort_j_15>(0)) &&
              (((r_OuterRel_ref_copy_ptr_0->items)[r_boff_12+(r_isort_j_15-(1))]).field0 >
              r_isort_tmp_17.field0))
          {
            (r_OuterRel_ref_copy_ptr_0->items)[r_boff_12+r_isort_j_15] =
                (r_OuterRel_ref_copy_ptr_0->items)[r_boff_12+(r_isort_j_15-(1))];
            r_isort_j_15 = r_isort_j_15-(1);
          }
          (r_OuterRel_ref_copy_ptr_0->items)[r_boff_12+r_isort_j_15] = r_isort_tmp_17;
        }
      } else {
        if(r_isort_numEntries_20>(0)) {
          (r_OuterRel_ref_copy_ptr_0->items)[r_boff_12] = (OuterRel->items)[r_boff_8];
        }
      }
    }
  }
  return r_OuterRel_ref_copy_ptr_0;
}
```

Figure 3.1: **Parallel C++ Code generated for ROFL Program in Section 3.1** (with some manual linebreaks and reformatting to fit this page).

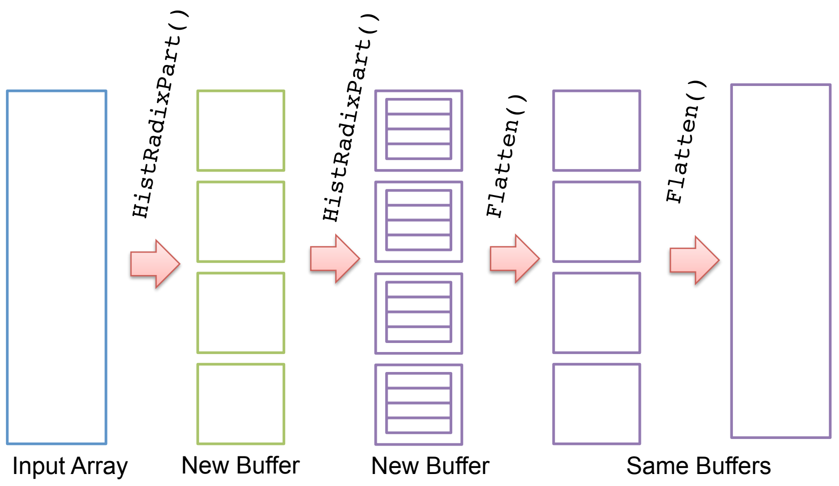


Figure 3.2: Nested array parallelism implied by ROFL operators

inputs. An example of a structural primitive is the previously-seen `Split(A, Nslices)`, which turns an input array `A` of N records into `Nslices` arrays of N/`Nslices` records. The inverse of `Split()` is `Flatten()`, which simply discards the additional structural information added by `Split()`. Primitives like `Split()` are called *nesters* since they add layers of nesting onto array structures, whereas those that remove them are called *flatteners*. A third class of operators, called *nested reductions*, includes those that expect an array of arrays as input, and perform some sort of aggregation across them.

Nearly all ROFL programs involve a combination of nesters and flatteners. They tend to follow the basic structure displayed in Figure 3.2, wherein inputs are subdivided for independent processing, and then recombined in the final output. In this case, the nesters are histogram-based radix partition operators, that both compute and change structure. As Section 3.3 describes, they are not primitives, but compounds comprising several primitives. The coloring of the arrays in Figure 3.2 indicates that they allocate new buffers to contain their output, while the two flatteners operate in-place.

However, that aspect of their semantics is not exposed to the user. ROFL is thus also a functional language in the sense that it leaves memory management at the discretion of the compiler. The code in the insertion sort example does not indicate whether the sorting mutates its input array(s) directly, or whether a new buffer is allocated to contain the output; this ought to depend on whether the original input is ever reused. Partitioning operations, however, involve moving all records to a new, data-dependent location, so they inherently require a new buffer to contain their output [1].

In both cases, Ressort automatically decides what to do. The programmer need never

[1]The new in-place partitioning algorithms of Polychroniou and Ross [33] do not, but as of this writing Ressort does not implement any of these

worry about allocating and freeing buffers, or how many of which type are needed. Even for complex operators, the compiler can statically determine most data structure sizes as a function of the operator's input size, and consolidate all `malloc()`s inside a preamble to the main record-processing loops. While it is true that the different memory overheads imposed by different algorithms can impact overall performance substantially (especially in systems that process multiple queries simultaneously [10]), ROFL still does not directly reveal memory management details because this can simply be "tuned over" as part of the overall performance, and should a particular design point's memory overhead limit its throughput, then it will of course be rejected in favor another. In the case of multi-programmed databases, Ressort can supply a static estimate of a ROFL operator's memory footprint to the query optimizer, which may have more knowledge about the memory and cache footprints of concurrently-scheduled operations.

Lastly, Ressort leaves unspecified the relative order of execution of different parts of a compound operator with respect to each other, and so it is also the compiler's job to schedule computation efficiently when it linearizes the operator DAG (Chapter 4).

3.3 Composing Shuffle Kernels in ROFL

The real purpose of ROFL is to support the expression of more interesting kernels as compositions of primitives. These, in turn, can serve as building blocks for larger operators still, and can even be made themselves to resemble primitives by way of syntactic sugar. This section describes the standard library of ROFL operators, `LibRofl`. It supplies parameterized operator *generators* for a variety of common tasks, including parallelized radix partitioning, and several types of sorting.

Radix Partitioning

Partitioning is a fundamental step in many shuffle kernels, and ROFL supplies a suite of primitives to support it. It is not itself a primitive because, as Algorithm 1 shows, it comprises three separate phases and two loops over the input array. Dividing them lets the compiler schedule them independently[2], but this underlying machinery can effectively be hidden behind syntactic sugar, as shown in the `HistRadixPart()` operator of Figure 3.3.

This operator generator will produce different code depending on the particular choice of radix bits and specified degree of parallelism. Because many algorithms are built on top of it, Section 7.1 evaluates its performance in isolation, and examines the impact of code generation parameters for each of the ROFL primitives that compose it (Section C.2). Later, we show how a simple Scala program can build a radix sort operator out of it.

[2]There is a serial dependency between them, but the compiler can schedule them relative to other operators, that is. A more important reason for the split is actually that they require different kinds of buffer structures to be allocated for them

```scala
// Assembles a ROFL radix partition operator on the chosen
// bit field of the input keys with the indicated degree of parallelism.
def HistRadixPart(base: Operator, lsb: Int, msb: Int, threads: Int): Operator = {
    // Split the input into ‘threads‘ parallel segments
    val splitPar = SplitPar(base, Const(threads))
    MergeHistograms(          // Condense per-thread histograms into one
        MoveRecordsHist(      // Distribute records to their partitions
          splitPar,
          ReduceHistograms(   // Do prefix sum operation
            BuildHistogram(   // Build per-thread histograms in parallel
                splitPar,     // ... parallelism implied by array type!
                msb, lsb),
            multi = true),    // Indicates parallelized partitioning
        multi = true))        // (same as above)
}
```

Figure 3.3: Scala/ROFL Code for Parallel Radix Partitioning

Multi-pass partitioning, which can alleviate the performance degradation of high-fanout on some platforms, can be specified in ROFL simply by applying our `HistRadixPart()` operator to its own output. The only complication is that this results in a nested histogram structure: if the totally resulting radix of a two-pass partitioning is $R = R_1 + R_2$, then the rth partition is accessed via a doubly-indirect lookup first of the R_1 MSBs of r and then the subsequent R_2 bits.[3]

Sorting

ROFL supplies only a few sorting algorithms as primitives; most actual sort operators are assembled as compositions of sort, split, partition, and merge primitives by Scala-based operator generators similar to the one in Figure 3.3. In fact, the only sorting primitive currently implemented in the Ressort compiler is `InsertionSort`, though we imagine adding a SIMD-ized `BitonicSort()` sort network generator in the future, along with support for an opaque `SmallSort()` primitive that maps directly to a hardware sort accelerator on platforms that support it. [4] In this section, however, we examine how to build serial and parallel sort operators out of other ROFL primitives.

[3]A future version of ROFL should include a “merge histograms upward” operator to eliminate the superfluous indirection.

[4]These could conceivably be calls into vendor-supplied libraries in the case of hardware accelerators, or a Ressort-supplied code generator, in the case of bitonic sort, that resembles those described in Appendix C.

```scala
// Makes a ROFL operator to perform LSB radix sort on 32-bit keys
// by recursively applying partition and flatten operators.
def LsbRadixSort(base: Operator, threads: Int, radix: Int): Operator = {
    // First, handle enough LSBs to make the remaining # of bits
    // a clean multiple of the radix
    val rem = 32 % radix
    val first = if (rem != 0) {
      HistRadixPart(base, lsb = 0, msb = rem, threads = config.threads)
    } else {
      base
    }
    // Recursively apply the radix partition operator and flatten
    def part(base: Operator, lsb: Int): Operator = {
      if (lsb < 32)
        part(radixPartOp(base, lsb, lsb+radix-1, threads), lsb+radix)
      else
        base
    }
    part(first, rem)
}
```

Figure 3.4: Scala/ROFL Code for Parallel Radix Sort: See Figure 3.3 for a definition of `HistRadixPart()`

Radix Sort

The `HistRadixPart()` operator generator can be leveraged to implement an LSB radix sort algorithm. For 32-bit keys, radix-sort with radix R merely involves partitioning the input data $\lceil 32/R \rceil$ times, with the radix bits at each stage moving from the LSBs to the MSBs of the key field. In this case, a `Flatten()` must be applied to the result of each partitioning, since the subsequent partition operations need to treat their inputs as a single, flat array. Figure 3.4 demonstrates how to do exactly this for any arbitrary $R < 32$ using the parallel `HistRadixPart()` operator.

Radix-Partitioned Sorts

An alternative strategy to the radix sort described above would be to first partition the input records based on their *MSBs* and then sort each of the resulting partitions in parallel, or at least in-cache.

If LSB radix sort is chosen as the per-partition sorting method, then the code to implement the full sort is really just a wrapper around `LsbRadixSort()` from the previous section that applies another `HistRadixPart()` to the input *before* applying the `RadixSort()` generator's code, but *without* inserting a `Flatten()` operation in between. Of course, for optimal

```
def RadixPartInsertionSort(
        input: Operator, bits: Int, threads: Int): Operator = {
    Flatten(
        InsertionSort(
            HistRadixPart(input, msb=31, lsb=31-bits+1, threads)))
}
```

Figure 3.5: Radix-Partitioned Insertion Sort: Radix partitioning allows insertion sort to be applied in parallel across partitions.

efficiency, the `LsbRadixSort()` code must be modified to avoid redundant partitioning based on the MSBs, which will be the same for all records within a given partition. ‘LibRofl‘’s actual implementation of these operators does so, and can handle keys with an arbitrary number of bits.

The user may also want to set the number of threads requested by `LsbRadixSort()` to one, thereby letting the number of partitions determine the degree of available parallelism, and giving control over load balancing to the OpenMP runtime. Or the user may exploit the initial partitioning to obtain LLC-sized partitions, and then sort one partition at a time, but in parallel out of the shared cache. Exploring this design space requires changing just a few lines of Scala/ROFL.

Future Work: Cache-Blocked Split-Merge Sorts

The structural converse of partitioned sorts is the split-merge sort, of which the familiar merge sort is one example. The algorithms in this family start by *dividing* their inputs (e.g. with the `Split()` primitive) into per-thread or per-cache block sub-arrays, sorting the sub-arrays, and then merging those with a merge operator.

Figure 3.6 gives an example from this class of sort algorithms: the cache-blocked radix-merge sort. The `CacheBlockedRadixSort()` function returns a ROFL operator that divides its input into cache-size blocks, applies a parallel radix sort to each LLC-sized cache block in sequence, and then invokes a merge network primitive. We do not evaluate this particular operator in Chapter 7, however, as the current version of Ressort does not yet include a merge operator.

3.4 Summary of ROFL Operators

Table 3.1 presents a more detailed overview of the various operators that exist in ROFL, as well as some that are not yet implemented by the compiler but are nonetheless important parts of the overall algorithmic space targeted by ROFL. In the table, each operator is named along with the number and types of its arguments and the type of its output. Types are specified using a shorthand notation where $Arr[T]$ designates an array of elements of type

```
def CacheBlockedRadixSort(
      input:     Operator,
      blockSize: Int,
      radix:     Int,
      threads:   Int): Operator = {
   MergeSlices(      // Builds a merge tree on top of the sub-arrays
      RadixSort(     // Could easily replace with another low-level primitive
         SplitSeq(  // Don't allow parallel sub-arrays
            OuterRel,
            slices = Length(input) / Const(blockSize)),
         radix,
         threads))
}
```

Figure 3.6: Cached Blocked Partition and Merge Sort

T, *or* an array of arrays of type T, up to an arbitrary degree of nesting. As a base type, T simply refers to any kind of record, unless otherwise specified (see $Arr[Index]$ in some of the histogram operators, for example). All operator types are specified as *functions*, since a ROFL program is actually a specification of how an input array should be transformed into an output.

Table 3.1: Summary of ROFL Operators and their Types

Primitive : Type	
Description	**Status**
`OuterRel` : $Arr[T] \to Arr[T]$	
Refers to the first argument (record array) passed to the compiled function	Implemented
`SplitPar(o, N, [padding])` : $Arr[T] \to Arr[Arr[T]]$ `SplitSeq(o, N, [padding])` : $Arr[T] \to Arr[Arr[T]]$	
Divides the input array into `N` arrays of |`o`|/`N` elements. If `padding` is specified, then as many elements of padding are added between each pair of slices. `SplitPar` allows parallel processing of sub-arrays, while `SplitSeq` does not.	Implemented
`Chunk(o, N, [padding])` : $Arr[T] \to Arr[Arr[T]]$	
Divides the input into |`o`|/`N` arrays of length `N`. If `padding` is specified, then as many elements of padding are added between each pair of slices.	NOT implemented
`InitHistogram(o, msb, lsb, [partitionPadding])` : $Arr[T] \to Arr[Index]$	
Initializes a radix r = (`msb` − `lsb` + 1) histogram with 2^r entries. If `padding` is specified, this will produce as many elements of padding *between partitions* in the resulting buffer once records have been moved.	All but `partitionPadding`
`BuildHistogram(o, hist)` : $Arr[T] \times Arr[Index] \to Arr[Index]$	
Reads records in `o` and increments counters in `hist` accordingly.	Implemented
`ReduceHistograms(hist, [multi])` : $Arr[Index] \to Arr[Index]$	

Performs a prefix sum reduction on the counters of `hist`. If `multi` is set, then the histogram is interpreted as a multi-threaded histogram of type $Arr[Arr[T]]$, and interpreted appropriately (see Algorithm 1).	Implemented
`MoveRecordsHist(o, hist, [padding])` : $Arr[T] \times Arr[Index] \rightarrow Arr[T] \times Arr[Index]$	
Moves records of the input relation into their assigned partitions according to the offsets contained in the prefix-summed histogram. If `padding` was set in `InitHistogram`, then it must be set to the same value here.	Implemented
`RestoreHistograms`**(o, [multi])** : $Arr[T] \times Arr[Index] \rightarrow Arr[T] \times Arr[Index]$	
Restores all counters in `hist` to their state before the application of `MoveRecordsHist`, which will have incremented each partition's counter by the number of records it contains. If `multi` is set, then the multi-threaded histogram case is handled correctly. Note that `o` should be an operator containing `MoveRecsHist`, and its type should be a combination of histogram and buffer.	Implemented
`Compact(o)` : $Arr[Arr[T]] \rightarrow Arr[Arr[T]]$	
Produces a new output array where padding in the topmost layer of `o` has been removed.	Implemented
`Flatten(o)` : $Arr[Arr[T]] \rightarrow Arr[T]$	
Removes the outermost layer of array nesting from `o`.	Implemented
`MergeSlices(o, [radix=2])` : $Arr[Arr[T]] \rightarrow Arr[T]$	
Merges all the (sorted) sub-arrays of `o` together into one sorted array. We plan to generate merge trees of arbitrary radix when we implement this operator, meaning that if `o` has M sub-arrays, then a tree of depth $\log M / \log \texttt{radix}$ will be emitted.	NOT implemented.
`Merge(o1, o2)` : $Arr[T] \times Arr[T] \rightarrow Arr[T]$	
Merges the sorted array of `o1` with that of `o2`.	NOT implemented.
`InsertionSort(o)` : $Arr[T] \rightarrow Arr[T]$	
Returns the result of sorting `o`'s array(s) by insertion sort.	Implemented

Chapter 4

ROFL Compiler Front-End

The *front-end* is the layer of Ressort responsible for turning a type-checked ROFL operator into intermediate IRep code that can then be translated to C++ code. Those latter stages that convert IRep to C++ and then compile it are, conversely, called the *back-end*. The frontend's myriad duties include:

- Turning ROFL operator expressions into operator DAGs
- Annotating DAG nodes with constraints on their scheduling, code generation, and buffer allocation
- Allocating and deallocating buffer structures for each node
- Finding opportunities to fuse together operators that process the same sub-array structures
- Determining the order in which to execute sub-DAGs relative to each other
- Emitting parallel for loop structures to process sub-arrays independently
- Invoking the appropriate code generator for the ROFL primitive specified at each node

This chapter briefly surveys these phases of compilation, while Appendices A and C describe their implementation in detail.

4.1 Overview

Figure 4.1 shows the front-end pipeline, and the kinds of DAG nodes produced at each stage. Different array data types are produced in successive stages as operator primitives are refined from ROFL into IRep (Section A.1). The ultimate result of this refinement is a DAG containing complete IRep code for each node, along with code to allocate each node's buffers and metadata. The final stage of the frontend flattens the refined DAG into a linear

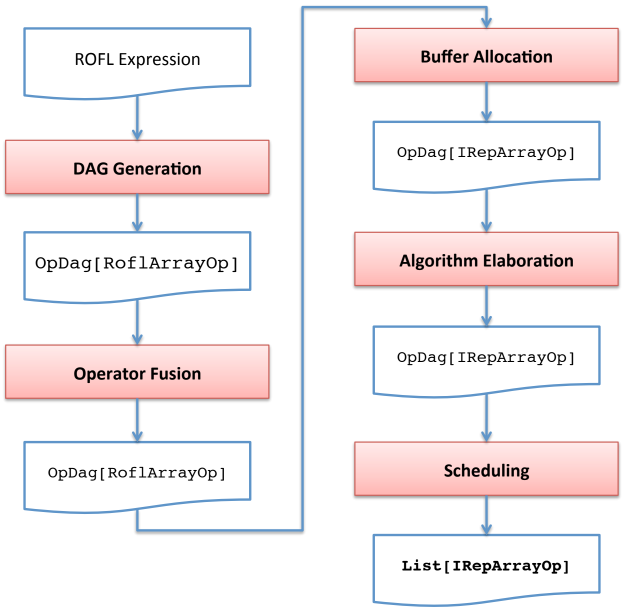


Figure 4.1: **ROFL Compilation Pipeline**: The frontend progressively refines a ROFL operator expression into (1.) a typed DAG with calculated buffer sizes (`RoflArrays`), (2.) a DAG with IRep code for each primitive and finally (3.) a linearized IRep function ready to be translated into C++

sequence of IRep AST blocks, prepends it with all buffer allocation code, wraps it in a function declaration, and hands the result off to the C++ translator.

4.2 Operator DAGs and Common Subexpression Elimination

In the first stage of compilation, the front-end breaks up the type-checked ROFL AST into separated DAG nodes, each of which contains one ROFL primitive. As it does so, it hashes and memoizes the result of each expression encountered, effectively performing common subexpression elimination on the ROFL input. This optimization is quite important in light of the current ROFL language specification, which does not support explicit binding of a ROFL expression's result to an identifier, so any operator used as input to two or more other operators would effectively be duplicated otherwise.

The initial DAG generation also sets certain attributes at each node, depending on the semantics of the node's ROFL primitive. These control how that node interacts with its inputs and outputs. Such properties include, for instance, whether the primitive raises or lowers the degree of array nesting, whether it is in-place or requires a new buffer for its output, whether it is a reduction across sub-arrays, and whether it processes its input records independently.

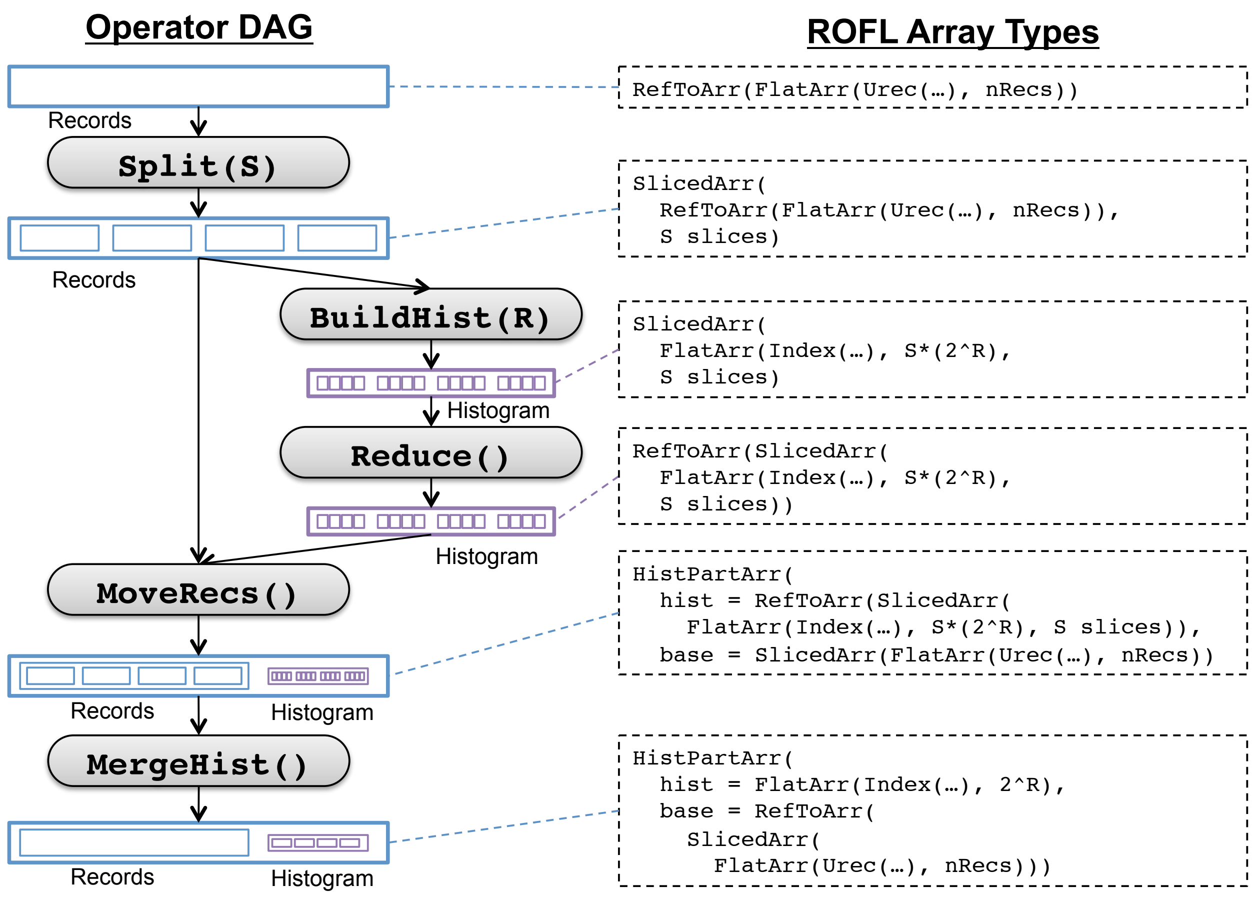

Operator DAG
ROFL Array Types
Records
Split(S)
Records
BuildHist(R)
Histogram
Reduce()
Histogram
MoveRecs()
Records
Histogram
MergeHist()
Records
Histogram
RefToArr(FlatArr(Urec(…), nRecs))
SlicedArr(
  RefToArr(FlatArr(Urec(…), nRecs)),
  S slices)
SlicedArr(
  FlatArr(Index(…), S*(2^R),
  S slices)
RefToArr(SlicedArr(
  FlatArr(Index(…), S*(2^R),
  S slices))
HistPartArr(
  hist = RefToArr(SlicedArr(
    FlatArr(Index(…), S*(2^R), S slices)),
  base = SlicedArr(FlatArr(Urec(…), nRecs))
HistPartArr(
  hist = FlatArr(Index(…), 2^R),
  base = RefToArr(
    SlicedArr(
      FlatArr(Urec(…), nRecs)))

Figure 4.2: `IRepArray` **Structures for a Parallel Partition Operator DAG.** Purple boxes represent ancillary data structures, while blue ones denote record arrays.

Figure 4.2 shows the form of this DAG for a parallel partition operator, and indicates what kind of array structure is produced by each node. It is during this DAG generation phase that the compiler determines the sizes of all data structures. For example, it infers that `S` copies of the histogram are needed as the histogram building operator is applied in parallel to `S` sub-arrays. It also decides that the output of that node should allocate a new array for the histogram, while the `Reduce()` node should re-use its input, as indicated with an explicit reference type. Section A.1 in Appendix A describes these types in more detail and explains how they govern loop generation during algorithm elaboration.

4.3 Operator Fusion: Exploiting Temporal Locality

Ressort exploits nested array parallelism at each DAG node by scheduling that node's computation across each sub-array independently using nested parallel for loops in the IR. A barrier is thus implied between a node and all nodes from which it receives its inputs (its *parents*). This has two important consequences.

First, a child node's computation cannot be applied to a sub-array immediately when it is produced by a parent, but must wait until all other sub-arrays in the parent have been processed. A single sub-array produced in the parent will likely have been evicted from cache by the time a child is able to visit it, and no temporal reuse will be exploited.

Second, even when an operator processes each of its input sub-arrays serially, any temporary arrays it produces for consumption by the next operator must be replicated for each sub-array. Such replication results in exacerbated memory overheads for some operations, and in unnecessary compulsory misses to data structures that should simply live in cache. For example, in the case of Figure 4.3, the initial `Split()` operator's output might require its sub-arrays to be processed sequentially, so only enough space to hold *one* histogram in the internal `HistPart()` operator should be needed. Yet, without further optimization, this structure will be duplicated many times.

The loss of potential performance resulting from these phenomena is clearly unacceptable, so Ressort ameliorates it via a technique called *DAG fusion*. DAG fusion identifies instances where parent and child nodes both operate on the same data with the same array nesting structures, and *fuses* parents to their children to produce *nested DAG nodes*–that is, nodes whose "operators" are themselves DAGs. This arrangement permits these DAGs-within-DAGs (*internal DAGs*) to be elaborated by the same machinery as the DAGs that enclose them.

When a fused DAG segment executes, the enclosing node generates a loop that selects the current window (Section A.2), and internal nodes execute as if the current window were the entire extent of data to be processed. The entire internal DAG executes for one window of the enclosing DAG before the next one is handled. In the case that the outer DAG's outermost sub-arrays specify sequential processing, internal array buffers require only enough space to hold the maximum size needed by any window of the outer array.

Appendix A Section A.3 describes the processes by which fusible sub-DAGs are identified and transformed into internal DAGs, as well as the adjustments in the `RoflArray` and wrapper structures to support them.

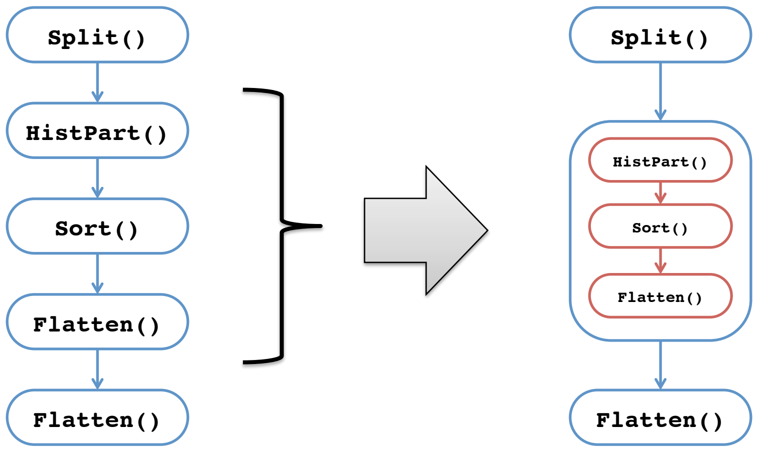


Figure 4.3: Operator DAG Fusion Example

Chapter 5

IRep: A Flexible, C-Like Parallel IR

The output of the frontend compiler is a program that implements the semantics of the original ROFL expression's composition of operators. This output is neither machine language code nor C++ code. Instead, it is an intermediate representation that is merely very close to C++ code, but which maintains some higher-level semantics and, more importantly, exists as a type-checked AST inside Ressort that can be manipulated and transformed in various ways. The present chapter presents this language, called simply IRep, describing its syntax, type system, parallel constructs, and its translation to parallel C++ with OpenMP. It also discusses a Scala-based interpreter for IRep, which has assisted in debugging the ROFL compiler and various transformation passes.

5.1 Motivation

Modern compiler backends have at their disposal an arsenal of sophisticated code generation techniques, and make highly performant choices during register allocation, instruction selection, and code scheduling. Ressort leverages this existing work by emitting C++ code as its final internal level of representation of shuffle operators, thereby entrusting these very low-level choices to the complex machinery of GCC and LLVM. However, auto-tuning still does entail forcing the compiler to explore different points in this space by feeding it with slightly different input codes. It is therefore desirable to have within Ressort a means of automatically transforming C-like code generated by operator elaboration.

This need is common to all auto-tuning systems, which have at some level a C-like program representation that can be programmatically varied. Often this has taken the form of messy string manipulation routines that simply concatenate predefined blocks concrete C code according to some parameter setting. Such systems are brittle, not easily extended, and cannot support code transformations that are independent of the semantics of each particular kernel being tuned. They also cannot support operations on code that depend on the types of variables and expressions.

We have attempted to remedy this state of affairs in Ressort by adding a new intermediate language, IRep, which is a Scala-embedded DSL for describing C-like programs without directly spelling out concrete strings of C code. Hosting it in Scala alongside the rest of Ressort allows for the construction of *code generators*, of which several examples are given in Appendix C. In this regard it is similar to and inspired by Chisel [2], a DSL developed at Berkeley for writing parameterizable hardware generators in Scala.

Ressort implements IRep by representing and constructing its ASTs directly in Scala as case classes (pattern matchable recursive data structures), and supplying some syntactic sugar to facilitate the embedding of IRep programs into Scala code. It also supplies a complete type-checker for IRep code, and a suite of routines that make it easy to write program transformations that operate on type-checked ASTs (two are described in Section **??**).

A translation layer converts type-checked ASTs into C++ code appropriate for different compilation targets and runtimes. Finally, it includes an interpreter that executes IRep programs directly in Scala.

5.2 Overview of the IRep Language

IRep was designed intentionally to resemble C, into which it is directly translated. It therefore includes most of that language's control flow constructs and data types, including functions, pointers, and structs. Some of these constructs differ slightly from their C counterparts in order to facilitate the construction of IRep code generators, to prevent unintended emission of sub-optimal C code, or to maintain higher-level semantics than could otherwise be expressed. The latter category includes parallel for loops, which map directly onto OpenMP equivalents. Explicit array operations, such as scans (prefix sum), both simplify ROFL-to-IRep compilation *and* enable translators for specific targets, such as GPUs or vector machines, to express those constructs in ways that are most efficient for those platforms. The rest of this section highlights these differences between C's structures and their IRep homologues, which can be summarized as:

- Expressions do not have side-effects; there is no `i++`
- Function calls are statements, not expressions; a special assignment statement captures their return values
- Loops are always given as a min, max, and stride tuple
- Arrays track their lengths, and always have `__restrict` semantics
- Functions can be nested, but only globals and the formal parameters of the innermost enclosing function are in scope

5.3 Language Syntax

A simple example, culled from the interpreter test suite, serves to introduce the most basic language constructs:

```
 1 val code: IrAst = {   // Binds ‘code‘ to the IRep AST at the bottom
 2    val f = Id("f")    // *Scala* declaration of IRep symbols
 3    val x = Id("x")
 4    val y = Id("y")
 5    // Now IRep code to use them:
 6    FuncDec(f, Map(x -> IRInt(), y -> Ptr(IRInt())), {
 7       Deref(y) := x * x
 8      } ) +
 9    DecAssign(y, IRInt(), 0) +
10    App(f, Map( x-> 10, y -> Ref(y)))
11 }
```

These lines declare a Scala constant `code` containing an IRep AST that defines a function `f` and applies it to a an automatic variable `y` through a pointer dereference. The first three lines inside the AST construction block declare Scala identifiers that stand in for IRep ASTs, in this case the minimal expression AST consisting merely of a symbol name (`Id()`). The function declaration uses a Scala `Map[Id, IRType]` (mapping from IRep identifiers to IRep types) data structure to describe the types of formal parameters. The `()` after primitive type names declares them as mutable; a trailing `(const=true)` would do the opposite. Evaluating this code in the interpreter yields a memory state where `y` is assigned an integer value of 100.

Plenty of syntactic sugar is at play in this example. The assignment in Line (7) exhibits three instances: `x` stands in for `Id("x")`, `x * x` is really a Scala call to the overloaded `*` operator that returns an expression AST, `Mul(Id("x"), Id("x))`, and an overloaded `:=` yields an `Assign(Deref(Id("y")), Mul(Id("x"), Id("x")))`.

All IRep language constructs are introduced in this section with this Scala shorthand notation, but the above code is indeed the underlying representation. It is also possible to obtain from the IRep AST implementation a “pretty-printed” form of this code, which cannot currently be written and parsed directly, but is useful when examining the output of IRep code generators:

```
Func f(
    x: IRInt,
    y: ptr_IRInt) :  {
  Deref(y) <--- x*x
}
var y: IRInt <--- 0
f(x=10, y=Ref(y))
```

Parallel and Sequential For Loops

IRep offers both normal and parallel for loops, and both are more constrained than in C. All for loops must be specified with an explicit induction variable and expressions for a minimum, maximum and stride value. Thus, the upper bound expression will be evaluated *exactly once*:

```
val stridedFor = {
    val sum = Id("sum")
    val i   = Id("i")
    DecAssign(sum, IRInt(), 0) +
    ForSeq(i, min=0, max=20, stride=4,
      (sum := sum + 1))
}
```

Thus, the above code sets `sum` to five. Constraining for loops in this fashion ensures that no complex termination predicates will mistakenly emerge increase per-iteration instruction counts. It also makes it easier to do certain kinds of transformations when the induction variable is explicit.

`ForSeq` specifies an ordinary sequential for loop; `ForPar` indicates that all loop iterations are independent and may be executed in parallel.

Functions

IRep also supports functions as a means for operator code generators to encapsulate chunks of reusable logic (the insertion sort "unroller" in Section C.4 is one example). The C++ compiler should ultimately inline most of these, assuming `-O3` is set, and avoid function call overhead.

A few features differentiate IRep functions from C++ functions. First, only primitives may be passed as arguments—structures are not allowed. Second, because expressions cannot have side effects, return values can only be captured through a special assignment statement, `AssignReturn()`.

Finally, functions *are* allowed to be nested, which makes them easier to use inside operator code generators, as those do not know the surrounding context into which they emit IRep code. However, as a compromise to simplify C++ code generation, the scope of a function's body includes only its formal parameters and global variables; the automatic variables and arguments of any enclosing functions are not visible.

5.4 Type System

In lieu of a formal enumeration of IRep's type system and typing rules, we simply note where these differ interestingly from those of C.

Fixed-Width and Machine-Dependent Integer Types

The IRep type system closely mirrors that of the C language. Our intent was to facilitate compilation into that language and to ensure portability across platforms in terms of both correctness and performance. To that end, IRep supports fixed-width signed and unsigned integer types at the 8-, 16-, 32-, and 64-bit granularities for storage of record fields requiring a precise width, and architecture-dependent integer types that match the target machine's native register widths.

Of these, the most important is `Index()`. It denotes the amount of storage required to hold an index or offset into any memory-resident, addressable buffer. The `Index()` type quite naturally maps onto the C language's `size_t` data type, and the current compiler always uses it as the type for loop induction variables in array processing loops, for histogram counter entries, and for all other kinds of array indexing operations.

There is certainly a chance that this conservative representation will waste space if used to store offsets into smaller buffers whose sizes are known statically. Indeed, the impact could be particularly acute for frequently-accessed structures (like histograms) that ought to reside in the higher levels of the memory hierarchy, and switching to a more compact data type could improve cachability in such cases. The benefit of marking variables with the `Index()` type is that it makes them obvious targets for a future compiler analysis pass that would down-size whichever indices it can prove are over-provisioned. It is not yet obvious whether or not it makes sense to generate different code variants to be chosen among dynamically once the actual sizes of buffers are known. This too is an object of future study.

Arrays

As the purpose of IRep is to represent implementations of array-processing operators, it was designed to make array-handling as straightforward and natural as possible. We do not think, for instance, that it is sensible to make IRep programmers manually maintain variables that track the lengths of arrays, as C does, and so IRep arrays do this automatically. Moreover, some common array operations, such as initialization (`memset`), prefix sum, reversal, and shift are exposed as explicit language constructs in order to hide target-specific optimizations in these routines from code generators. It should be the job of the C++ IRep *translators* to decide whether a given platform is best targeted by parallelization or utilization of dedicated instructions.

IRep does not support multidimensional arrays in the same way that C does. Rather than standardizing row- or column-major ordering, the language assumes that it is the frontend compiler's responsibility to manage data layout in flat buffers. However, it does support nested arrays, where all dimensions but the highest consist of arrays of pointers to the subsequent level's arrays.

Reference Types

In order to support function calls and structs, IRep includes an explicit `Ptr()` data type. This is something in between a C-style pointer and a C++ reference, as `Ptr`-typed variables can be declared uninitialized, but cannot be cast to an integer, and pointer arithmetic is not supported. A reference can be acquired to any valid lvalue via the `Ref()` operator. By fiat, no two references to arrays may alias each other, and the result of doing so is undefined. This restriction allows all arrays to be compiled to C++ arrays with `__restrict` semantics, which enables some compiler operations not otherwise possible under a more conservative analysis.

Chapter 6

Experimental Methodology

In order to demonstrate the functionality of Ressort, we evaluated the performance of a variety of operator compositions across several platforms (Chapter 7). Before presenting those results, we first describe in this chapter the experimental setup used to obtain them, including the hardware platforms studied, our data-gathering and correctness-checking methods, and our assumptions about the underlying format of input data to all ROFL operators.

6.1 Problem Setting and Experimental Parameters

Data Formats and Memory Layout

KP32: 32-bit Key-Payload Relations

For the purposes of this investigation, we consider only input relations whose records contain a 32-bit key and a 32-bit payload, which we refer to as *KP32 relations.* Although the intended use-cases for Ressort—namely, in DBMS query execution engines—will require processing relations of all shapes and sizes, we restricted the scope of our inquiry to the KP32 class as this represents a useful general-purpose data format that occurs quite frequently in the middle stages of query processing pipelines, even where the ultimate input or output format is more complex.

Indeed, a first step in the implementation of many complex queries will likely be to construct a *view* of the input tables that replaces each row of one or more columns by a hash of their contents, and a pointer to their original location. This view facilitates partitioning, sorting, and joining by drastically reducing the amount of data that must be moved around at each step.

Non-columnar Memory Layout

We further assume that all input records are stored in a *non-columnar* organization. Each record is stored as a key-payload tuple, and relations are really arrays of record structs.

Although this is the reverse of how column stores normally arrange their data at rest, we feel it is a reasonable structure for our experiments. In practice it will often be desirable to re-structure data this way while constructing the KP32 view, since it (1.) requires only one stream of accesses to contiguous memory and (2.) allows 64-bit machines to move entire records with a single load or store instruction. Our generated C++ code accomplishes this by moving data typed with the structures of Figure 6.1.

```
struct Rec_uint32_t_uint32_t_ {
  uint32_t field0;
  uint32_t field1;
};

struct _arr_urec__UInt32_UInt32 {
  struct Rec_uint32_t_uint32_t_*__restrict items;
  size_t len;
};
```

Figure 6.1: Input data format for all experiments in our evaluation. These C++ structures are automatically generated by translating the ROFL type `Arr(URec(Uint32, Uint32))`.

The cost of constructing these views of input data is not considered as part of our evaluation, nor is the cost of reordering complex row-oriented tables according to the results of the shuffle operators we examine. These are obviously important considerations in any practical setting, and we expect that the tasks of view construction and table reordering would themselves be good candidates for tuning in our framework. We nonetheless defer such an analysis to future work, which could examine the tradeoffs between the ease of computation on KP32 values, and the cost of constructing that representation.

Random Key Distributions

We used randomly-generated input datasets in the KP32 format described above. Our final assumption about these inputs is that the 32-bit keys values are taken from a uniform distribution on the range $[0, 2^{32} - 1]$. This setting is optimal for partitioning-based operators, as it results in generally even load balancing when different processors or threads are assigned to process different partitions. Many operators' performance will thus be sensitive to the uniformity and cardinality of the key distribution, and these attributes will ultimately determine the most efficient algorithms to apply to any input dataset. This is clearly an interesting tuning space, but it is beyond the scope of the present work. In the future, we wish to re-run the same experiments under a Zipf distribution with varying skew; for now, we consider even the optimal case of uniformity to be interesting enough on its own.

Problem Sizes

We consider three different sizes of input relations in order to illustrate the performance limitations imposed by different levels of the memory hierarchy. These categories are:

- **Small**: a handful [$O(10)$] of records, repeated many times over
- **In-Cache**: small enough [$O(10K - 100K)$] records to fit in the outermost two levels of the on-chip cache hierarchy
- **In-Memory**: relations large enough[$O(1G)$] that they do not fit in any level of cache, and involve significant DRAM traffic

6.2 Experimental Setup

We characterized the performance of different ROFL operators compiled by Ressort in Chapter 7 on different systems, ranging in size from a mobile processor to a 40-core server platform. This section describes both this set of platforms and our measurement methodology.

Platforms

Platform	**C/T**	**Freq.**	**L1**	**L2**	**L3**	**Linux**
NVIDIA Tegra 3 (Cortex A9)	4	1.6GHz	32KB I/D	1MB	N/A	3.1.0
Core i7-4765T (Haswell)	4/8	2.00GHz	32KB I/D 8-way	256KB 8-way	8MB 16-way	3.13.0
Xeon E5-2667 (Ivy Bridge)	$2 \times 8/16$	3.3GHz	32KB I/D	256KB	25MB	3.16.7
Xeon E7-4860 (Boxboro/Nehalem)	$2 \times 10/20$	2.27GHz		256KB	24MB	3.13.0

Table 6.1: **List of Platforms Used in the Evaluation**
The "C/T" column reports the number of cores and threads for platforms with simultaneous multithreading (SMT) or Hyperthreads. Multi-socket machines are reported as $N \times C/T$, where N is the number of sockets.

Platform	**Advertised**	**Measured**	
		Single Thread	Max. Threads
Cortex A9	6.0	0.83	1.5
Haswell	25	6.8	17
Ivy Bridge	60	6.7	60
Boxboro		4.8	92

Table 6.2: **Theoretical vs. Measured Memory Bandwidths (GB/s)**

Table 6.1 contains the microarchitectural parameters of the three platforms we used in this study. We selected the above machines because they provide a reasonable diversity of CPU- and system-level characteristics, such as cache sizes, core counts, memory bandwidth, frequency, and pipeline resources.

The least powerful of these machines is an ARM Cortex A9 that was included as part of an NVIDIA Tegra3 development board, and is included to illustrate the challenges of mapping shuffle algorithms efficiently to smaller cores. The second is a desktop-class Intel Core i7 with the Haswell microarchitecture—the latest available at the time of this writing. The third, Boxboro, has the much older Nehalem microarchitecture, but many hardware threads, and a very large LLC and high bandwidth memory system. Finally, the Ivy Bridge machine is included as a server-grade, high-bandwidth reference point with a reasonably recent microarchitecture.

Machine Memory Bandwidths

We calibrate our results in Chapter 7 against both the theoretical (advertised) maximum memory bandwidth of all reference platforms, and the actual throughput we could measure on each machine. Table 6.2 reports these numbers. Measured throughputs were obtained using a tool by Scott Beamer that performs a pointer chase through a 1GB-sized array of pointers that either (1.) point to an element at a constant stride offset (with a default of one cache line) in the array, or (2.) point to a random location in the array. Additionally, it measures n-degree memory-level parallelism (MLP) by performing n independent chases in the innermost loop, and can be configured either to have all threads perform this task, or merely a single one. Randomization is sometimes needed to achieve maximum bandwidth by distributing loads across multiple DRAM banks, while the MLP setting determines the number of independent requests that can be in flight at any time.

Cores, Threads, and Scaling Experiments It is important to distinguish between the number of cores or hyperthreads indicated in Table 6.1, the number of threads instantiated by the OpenMP runtime, and the meaning of the "threads" parameter reported across many experiments in the next chapter. For each of our experiments, we allowed the OpenMP runtime to determine the number of pthreads to use in each parallel region, and did not call `omp_set_num_threads()`. Instead, wherever a number of threads is reported, this

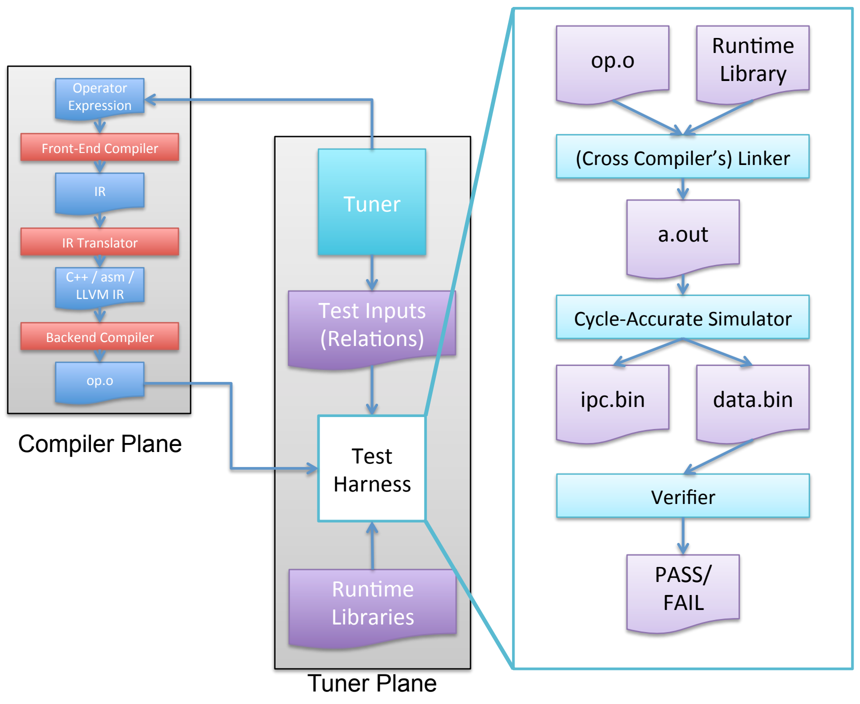


Figure 6.2: Cycle-Accurate Test Harness: Here "cycle-accurate" can either mean an RTL simulator or the host machine's hardware itself

designates the *degree of parallelism* induced by a particular operator. It is the number of sub-arrays introduced by the ROFL `Split(Rel, N)` primitive that are then processed in parallel. This primtive sets the "iteration count" of the outer OpenMP parallel for loop that parallelizes the execution of any operator applied to the result of `Split`. The loop bound is therefore also a bound on the number of active threads, since the body of the loop will not normally contain any further parallel regions.

6.3 Test and Verification

Figure 6.2 shows a magnified view of tuner plane portion of the Ressort framework. It is so called because its responsibilities are to generate and test variants of ROFL operators, measuring both their performance characteristics and their correctness. The tuner plane is further subdivided into the *tuner* itself, which generates the operators under test (OUTs), and the *test harness* which manages their execution and verification.

The "Tuner"

The tuner module is not yet an auto-tuner *per se*, but it is indeed the site where one could be integrated into Ressort. Its name derives, rather, from its job of varying the parameters to a ROFL operator generator, and invoking the compiler on these resulting variants to produce an object file that the test harness can execute. A parameterized operator is called a *benchmark*, which accepts a *configuration* set by the tuner. The latter also sets a configuration for the compiler to control the elaboration parameters described in Appendix C.

Finally, the tuner selects a problem size, and runs a *relation generator* to produce a randomly-generated array of input records. The only relation generators currently supplied with Ressort emit arrays of KP32 records with uniformly distributed keys (though the maximum range from which to draw these keys can be limited). Two such generators exist: one is simply a wrapper around an externally-hosted C++ program that writes a binary relation file out to disk, and the other is written in Scala to be run inside the JVM alongside the IRep interpreter-based test harness.

The Test Harness

The test harness is an abstraction for the various bits of machinery needed to (1.) link a compiled operator together with library code for relation I/O and performance counter measurement (2.) run the resulting executable natively or in an architectural simulator (3.) return the performance counter measurements to the Scala-based tuner plane and (4.) pass the OUT's output through an appropriate *verifier* to ensure that auto-generated code produced the correct results.

The Execution Environment

The middle box of the tuner plane diagram in Figure 6.2 represents the actual execution platform on which the OUT's performance is measured. All the results presented in Chapter 7 were obtained via direct execution on the host machine (the one that runs the JVM hosting Ressort's compiler and tuner planes) using PAPI [30] to extract native CPU performance counter values, as described further in Section 6.4.

However, the test harness infrastructure currently supplied by Ressort also supports operator execution on cycle-accurate architectural simulators and models. In particular, we have used it previously to measure performance on simulators for the research microarchitectures developed at Berkeley.

Two other kinds of execution environments do not report any performance measurements. One is the IRep interpreter, which is useful for debugging the Ressort compiler itself, and the other is an interface to `gdb` for debugging C++ and OpenMP translation.

The Runtime Harness

In all cases but that of the interpreter environment, the OUT object file is linked into a binary called the *runtime harness* that knows how to read in the input relations from disk, and call the OUT's main function with a pointer to the input relation buffers. When Ressort generates C++ code, it also emits a header file that indicates to the runtime harness which function to call, and what the types of the input and output relations are so that it can fill appropriately sized buffers.

The runtime harness is also responsible for making performance measurements during the course of operator execution and reporting the results back into the Scala-based test harness by writing them out to a file designated for this purpose.

Verifiers

Checking correctness is obviously important wherever algorithmically diverse implementations of a given kernel are benchmarked, and Ressort's test harness module includes correctness checkers for sort and partition operators on KP32 relations.

`ProblemSpecs`

When the tuner runs a benchmark, it generates a *problem specification* (`ProblemSpec`) to indicate (1.) which category of operator (sort, partition, join, etc.) is supposed to be implemented, (2.) what its input relations are, and (3.) any operator-specific parameters, such as the bits of the key to be used for partitioning. This specification succinctly enumerates all details required to determine whether or not a given output conforms to the semantics of the desired operator type.

In-Memory vs. On-Disk Verifiers

Input and output relations are passed in `ProblemSpec`s either as pointers to binary files on disk, or as Scala arrays of actual records, depending on the kind of verifier invoked. An *in-memory* verifier accepts the latter kind of data, and is intended to work with the IRep interpreter execution environment. More often, the harness uses an *on-disk verifier*, which is itself a wrapper around a C++ implementation of the most naïve version of an operator type that can compute the result of large problem sizes many times faster than its (unoptimized) Scala counterpart. Even so, we expect, and observe, that the majority of our experiment runtime was spent inside the various verifiers, as they are single-threaded, untuned, and, in the case of sort, call a standard library function that suffers a function call overhead on every comparison. The test harness therefore supports caching these results whenever multiple experiments are run with the same problem specification but different ROFL codes or compiler configurations.

6.4 Measurement

The test harness uses PAPI, the Performance Application Programming Interface [30], for accessing hardware performance counters to characterize operator efficiency as accurately as possible, and makes several efforts to eliminate or at least quantify measurement error and noise.

Performance Counters and PAPI

Per-platform Counter Availability

Not all of our experimental platforms support all kinds of counters we wished to monitor, so we used maximal subsets of these where possible and sometimes had to choose counter subsets on a per-experiment basis. All platforms studied support runtime measurement at the microsecond granularity, while also providing counts of retired instructions, data TLB misses, and L1 data cache misses. We were unable to obtain L2 cache miss statistics from the Tegra3 platform. Our x86-based machines supplied L2 cache event information, but we did not obtain LLC miss counts from them either.

PAPI and OpenMP

Since all relevant performance statistics aside from pure running time are derived from per-core or per-hyperthread event counts, they need to be aggregated across all hardware threads whenever they are collected. OpenMP hides the thread abstraction behind the parallel for loop interface so, the test harness wraps its calls to the PAPI library routines in an OpenMP parallel region that reads counters into thread-local arrays. A subsequent OpenMP critical section aggregates the per-thread counts into a global array. Thus, the harness will report statistics from all threads by allowing the OpenMP runtime to schedule the counter code on all threads.

Trials and Variance Estimation

All experiments were conducted on real hardware running Linux, and running the JVM-based Ressort framework along-side the OUT, so some amount of measurement error is unavoidable. Two methods are employed to mitigate its impact: repetition through multiple trials *inside an operator* using *different datasets* and repeated invocation by the test harness of the OUT on the *same input data*. Each has different consequences.

Error Bars Throughout all the figures in Chapter 7, error bars represent *one standard deviation* around each depicted data point, which itself is the average over all outer trials for a particular experiment.

Inner Trials

The obvious way to reduce measurement variability is to test each OUT many times across many input datasets. Because our input datasets are generated prior to launching the runtime harness, data for all trials must be created at once, and the runtime harness must be told how many trials' worth of data an input relation comprises.

To that end, each `ProblemSpec` specifies a number of *inner trials* into which to divide its input relations. If the number of inner trials is N, the verifier treats the inputs and outputs as the results of N independent experiments, checking the correctness of each one's output separately. At the same time, the benchmark must wrap its inputs in ROFL `SplitSeq(..., N)` operators whose semantics stipulate that each of the N sub-arrays must be processed independently but *in series* with each other, rather than in parallel.

When the runtime harness calls into the OUT's main function, it passes a pointer to the buffer containing input records for *all* inner trials, and it wraps either side of this call with a single start and stop of the PAPI performance counter monitoring facility. Thus, exactly one set of performance statistics is generated regardless of the number of inner trials specified, so per-record statistics represent averages across all trials.

This method has the obvious disadvantage that it is impossible to measure variance, but is nevertheless effective at mitigating the influence of noisy runtime conditions. We remedy the lack of variance measurement by adding an additional layer of repetition, as described below.

Outer Trials

To further reduce and quantify measurement error without injecting instrumentation code inside of black-box OUTs, and without requiring memory-resident buffers for further trials' worth of input data, the runtime harness' call into the OUT is wrapped an outer loop that repeats the *same experiment* multiple times using the same set of input data for all inner trials. Moreover, each iteration of this loop makes and saves a separate performance counter measurement, over which the runtime harness then calculates the mean and variance to report back to the Scala-based test harness.

This process could potentially generate misleading results in the case of small problem sizes, as it may unfairly warm up layers of the cache hierarchy, or train branch predictor state that should realistically be initially empty. However, we can control for all these side-effects by increasing the number of inner trials to be just large enough that the impact of these phenomena is negligible.

Variance with Inner and Outer Trials

Most of the performance statistics reported in Chapter 7 are stated as event counts *per record*. Since we only directly measure variance over each outer trial loop, we report the mean and variance of per-record statistics as derived values. For event count E:

$$\text{TotalRecs} = \text{NumTrials}_{\text{inner}} \times (\text{Recs}/\text{Trials}_{\text{inner}})$$

$$\mu(E/\text{rec}) = \frac{\mu(E_{\text{outer}})}{\text{TotalRecs}}$$

$$\sigma(E/\text{rec}) = \sqrt{\frac{\sigma^2(E_{\text{outer}})}{\text{TotalRecs}}}$$

One metric that is not derivable in this way is throughput, which involves the reciprocal of runtime whose variance therefore cannot be computed directly from the variance of runtime. We therefore measure that statistic directly in the runtime harness itself, and report its *per-outer trial* variance back to the Scala-based test harness.

Chapter 7

Evaluation: Tuning Results

We benchmarked the performance of operators generated by Ressort on our reference architectures in order to determine (1.) whether the primitives themselves were reasonably efficient and (2.) which composite algorithms were best for partitioning and sorting. Aside from the performance considerations themselves, it is worth noting that about thirty lines of Scala-embedded ROFL generate all the operators evaluated below. In the case of MSB-LSB sort, whose compiled output encompasses approximately four hundred lines of C++, this represents a 10x savings in code and effort compared to a manual implementation.

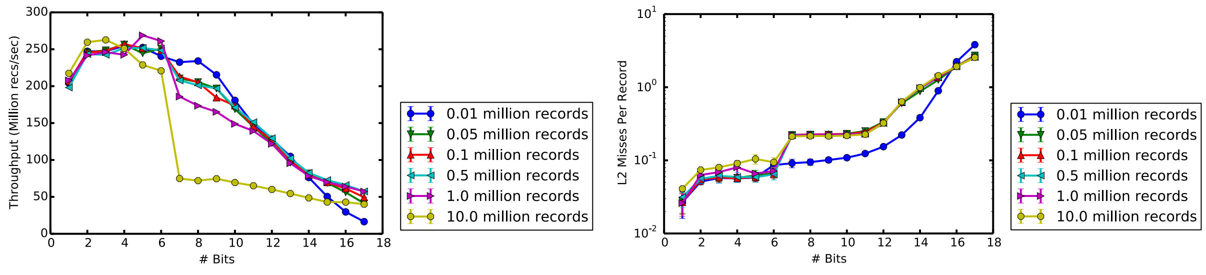
7.1 Partitioning

Since partitioning underlies our sorting algorithms, we examine it first. Depending on the particular use case, it may be desirable to partition a relation into an arbitrary number of fragments. We examine radices ranging from $b = 1$ to $b = 30$, and attempt to determine the optimal algorithm for each radix. The parameterized partition operator generators presented in Section 3.3 give rise to a design space that we explore empirically, and which is itself the basis of an even larger design space for sorting algorithms. For partitioning alone, though, the only currently controllable parameters are the radix itself, and the decision to use a multi-pass algorithm.

In-Cache Partitioning

For problems small enough to fit in the L2 or L3 cache, we measure the partitioning ability of a single thread alone. On x86-based platforms, the number of instructions per record required to execute this kernel is consistently about 23 when compiled with the `-O3` flag.

Figure 7.1 shows that a single thread operating out of cache on an Ivy Bridge machine can achieve a throughput of 250M records/second using Ressort's auto-generated partition code. At 2.5 IPC, and without any significant cache misses, this is our peak per-core efficiency for partitioning. Since each phase of this kernel is a loop with about ten serially dependent



(a) Throughput (b) Log2(L2 Misses / Record)

Figure 7.1: In-Cache Single-thread Radix Partitioning on an Ivy Bridge (50 outer trials).

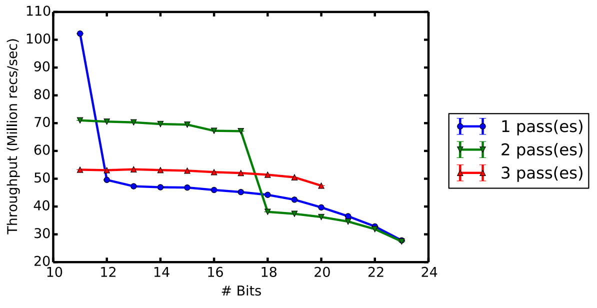
instructions, this corresponds to the regime in which two independent loop iterations fill the issue window. Vectorizing [1] the inner, per-thread partition loop could break this dependency and potentially increase IPC, but this is the best performance we can expect without doing so. The Nehalem (Boxboro) platform exhibits roughly the same performance trajectory, with lower peak IPC owing to its less sophisticated microarchitecture. Its throughput (not shown) is 120M records/second at its peak. Appendix 7.1b contains analogous performance plots for that platform.

In both cases, the relation with 10M records (80MB in size) does not fit in any layer of cache, though prefetching maintains roughly the same level of partitioning performance (the number of cache misses per record is only slightly higher than it is for the smaller problems) until the radix exceeds $b = 5$. There the TLB miss rate (shown in the Appendix D) spikes to one per four or five records, significantly limiting throughput[2]. That spike occurs only for the 10MB dataset on the Nehalem and Ivy Bridge microarchitectures, and is not seen on Haswell (Figure D.3), where there are no TLB misses and where throughput remains comparatively high. Conflict misses common to all problem sizes also begin to gradually impede performance beyond this point (Figure 7.1b), as the input, output, and histogram all compete for ways of the L1 and L2.

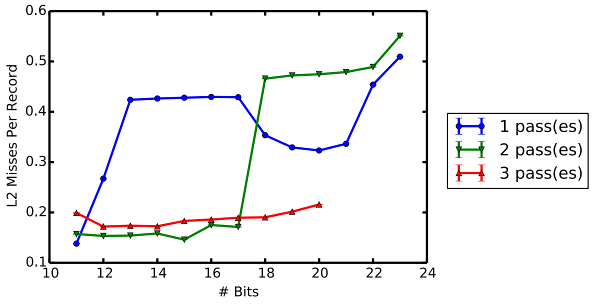
One way to alleviate this cache and TLB pressure is to split the partitioning into multiple passes, as described in Section 2.1. In this case, since the relations fit in cache, scanning them multiple times will not increase DRAM bandwidth requirements, so we would expect it to be particularly fruitful. Figure 7.2 shows that the reduction in L2 cache misses from 2- and 3-way partitioning can indeed improve throughput at high radices, though a single pass is more optimal at lower radices because of its lower dynamic instruction count.

[1]By "vectorizing" we mean actually changing the structure of the partition algorithm by introducing another ROFL `SplitPar(M)` operator on top of the one used to divide inputs between threads. This additional M-way split would result in $T \times M$ copies of the histogram for T threads. Thus, just as using T independent histograms creates thread-level parallelism, this further subdivision would create ILP. We don't yet evaluate this design point because the prefix sum operator does not currently support three-level histograms.

[2]It is not clear why transparent superpage promotion did not take place here, but we defer investigation of that effect to future work.



(a) Throughput



(b) Log2(L2 Misses / Record)

Figure 7.2: **Multi-pass in-cache partitioning of 10M records on an Ivy Bridge**: At radices higher than $b = 12$, splitting the partitioning into two or three passes at half or a third the degree can eliminate TLB and cache pressure. Figure 7.2b shows that multiple passes keep the number of L2 misses low.

Platform	**% Peak Bandwidth**		
	All Threads		Single Thread
	Theoretical	*Measured*	*Measured*
Haswell	48%	70%	70%
Nehalem		20%	50%
Ivy Bridge	55%	55%	71%
Tegra 3	2%		16%

Table 7.1: **Estimated % of Measured & Theoretical Peak BW for Partitioning**: Each column shows the partition bandwidth estimated by Section D.1 as a fraction of either the measured (with Scott Beamer's tool–Section 6.2) or of the peak bandwidth advertised by the machine's manufacturer.

Parallel Partitioning and Scaling

In theory, out-of-cache partitioning should run at the same rate as in-cache partitioning, at least for low radices. The streaming access pattern of each phase of the algorithm should trigger automatic prefetching, and the kernel should execute at the same throughput per core as its in-cache sibling after an initial startup latency. However, we observe that this does not occur. Instead, Figure 7.3 shows that even at small radices, partitioning throughput is only about five times greater than that of a single thread operating within cache, whereas it ought to be 16x higher on a machine with sixteen cores. Indeed, the IPC for out-of-cache partitioning is consistently lower than that of in-cache, often by a factor of five or more, and as Table 7.1 shows, available memory bandwidth is massively underutilized.

This behavior seems to result from an increase in LLC misses, though it is not obvious why that should be. In the single-threaded, radix $b = 2$ case, the working set is never larger than six cache lines, one of which is the current line in the input stream and four of which are

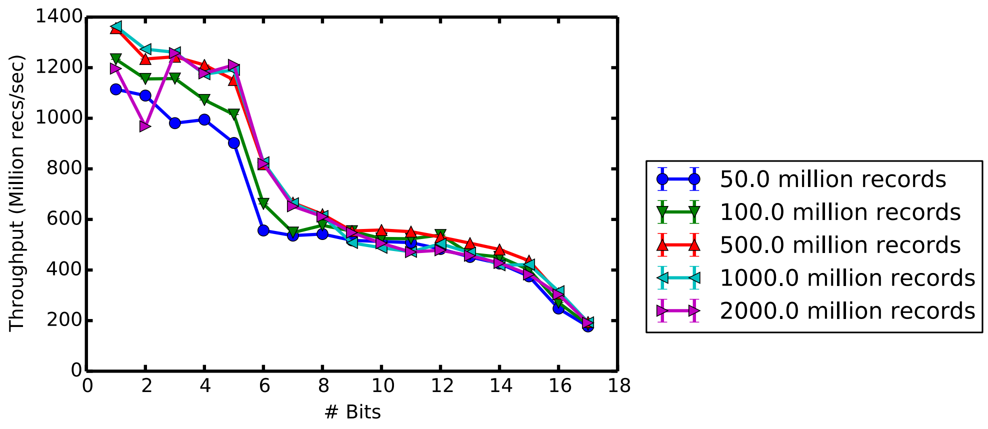
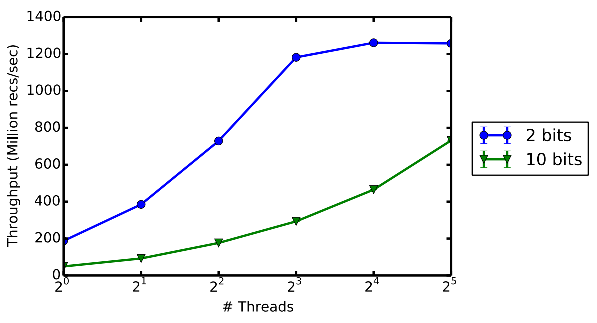
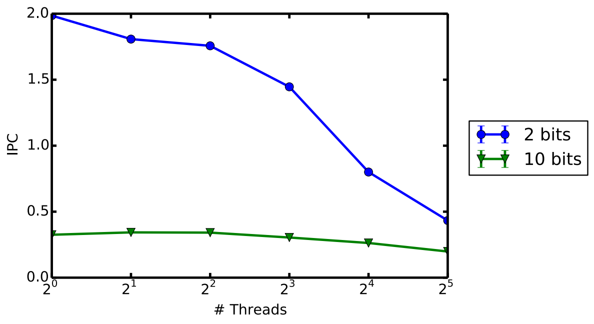


Figure 7.3: **Out-of-Cache Radix Partitioning on an Ivy Bridge** (50 outer trials, 16 threads): Peak throughput can only be realized for the ideal case of $R \in \{2^1, 2^2\}$. Even then, IPC is still less than 1.0, and declines precipitously at $b = 5$, at which point the TLB miss rate rises to one per five records.



(a) Throughput

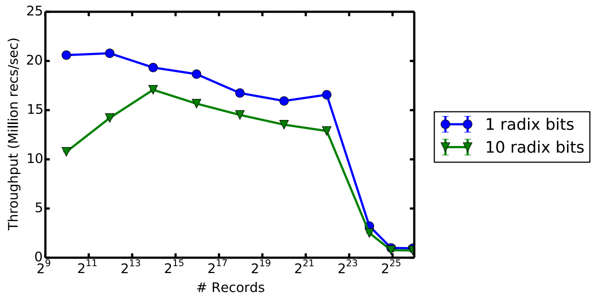


(b) Instructions / Cycle

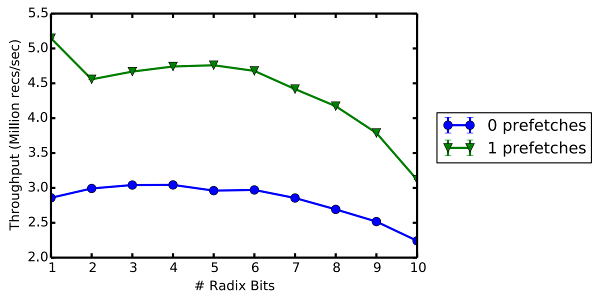
Figure 7.4: **Scaling Radix Out-of-Cache Radix Partitioning on Ivy Bridge**: 10^9 Records. Note that the $T = 2^5$ datapoint corresponds to SMT operation on this machine.

the heads of their respective output streams. All of these accesses are entirely prefetchable, so it is surprising that L3 misses still occur on Ivy Bridge and Haswell. It may well be the case that the naïve histogram-building and record-moving loops cannot generate memory requests at a sufficient rate to keep these streams active. The vectorization experiment proposed in Section 7.1 might indicate whether this is so.

Figure 7.4 shows that the IPC of a radix $b = 2$ partition on a 16-core, 32-thread Ivy Bridge does decrease by a factor of four as the number of threads is scaled. Although this degradation in per-core throughput is more than made up for by a 32-fold increase in threads, it does account for the sublinear scaling. It corresponds to 1.5x increase in L3 cache misses over the same period (see Figure D.5c in the appendix). More cores mean more competition for prefetching resources, which raises the miss rate in spite of the small working set size.



(a) Partitioning Throughput for Different Relation Sizes



(b) Throughput with and without SW Prefetch Insertion, radix 1

Figure 7.5: **Radix Partitioning on a Cortex A9**:

Partitioning performance for the less powerful Tegra 3 (Cortex A9) depends on a very different set of microarchitectural parameters. When the relation fits in the 1MB L2 cache, throughput can be as high as 20M records/second, but it degrades more than seven-fold by the time the relation is 32MB in size (Figure 7.5a). Since hardware data prefetching clearly did not succeed in hiding memory latency in this configuration, we configured the ROFL compiler to automatically unroll the histogram-building and record-moving loops and insert software prefetch hints (Appendix C Section C.2 describes this code generation parameter). Figure 7.5b shows that this did indeed restore throughput by as much as 2x, but the fact that this entails utilizing only 16% of the measured single-thread memory bandwidth (Figure 7.1) indicates there is still much performance left to be extracted.

7.2 Sorting

The space of possible sorting implementations is much too vast to examine exhaustively. Even narrowing the scope of our investigation to include only those built out of partitioning and insertion sort still leaves many variables unconstrained:

- LSB radix sort vs. MSB radix sort vs. MSB-LSB
- Which radices to use at each level
- Whether to use e.g. insertion sort at the lowest level

For both in-cache and out-of-cache sort, we compare pure LSB sort to MSB-LSB sort. In both cases, we prune the search space to include only radices between $b = 7$ and $b = 12$. Given the partitioning performance in Section 7.1, it is natural to make this restriction, as the IPC improvements at radices less than $b = 7$ are too small to justify additional passes over the input, while the very low IPC at $R > 2^{12}$ will prevent any speedup from reducing the number of relation scans.

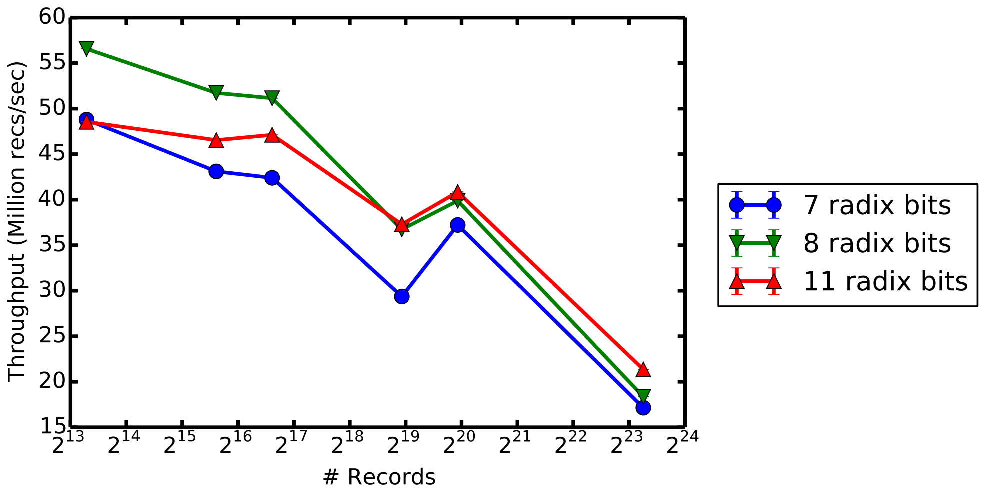


Figure 7.6: **Single-Threaded In-Cache LSB Sort on Ivy Bridge**: Radix $b = 8$ is the optimal in-L2 partitioning algorithm because it fits the histogram in L1, while $b = 11$ narrowly surpasses it once the relation outgrows the L2.

In-Cache Sort

When sorting in cache with one thread, LSB sort and MSB-LSB sort should perform similarly; MSB-LSB's possible cache-blocking and parallelization advantages should not be relevant. LSB sort's performance is easier to analyze, since it has three clear inflection points in the radix parameter: one at $b = 8$, where the number of passes required to complete the partition decreases from five to four, followed by two more at $b = 11$ and $b = 16$. Radices in between these values will be sub-optimal, since they execute the same number of instructions per record while exacerbating the microarchitectural bottlenecks of higher-radix partitioning.

We discuss only these work-optimal configurations[3]. On our Intel server-grade platforms, radix $b = 8$ turns out to be optimal, because $b = 11$ requires a histogram of size $2^{11}(8) = 80KB$, which does not fit into 32KB the L1 data cache, and so has nearly twice the L1 miss rate. As Figure 7.6 shows, though, it becomes optimal in spite of this once the relation no longer fits into the L2 ($|R |= 2^{19}$ or $4MB$).

In-Memory Sort

When relations do not fit in cache, LSB sort and MSB-LSB sort have different theoretical performance characteristics: LSB sort's throughput depends exclusively on out-of-cache partitioning, while MSB-LSB reduces the size of the output partitions during the LSB phase to the size of the output of MSB partitioning. On Ivy Bridge, we found that MSB-LSB sort performed 15-30% faster (for 500M and 1B records, respectively) than pure LSB sort, and

[3]Appendix D validates this claim with a few extra data points

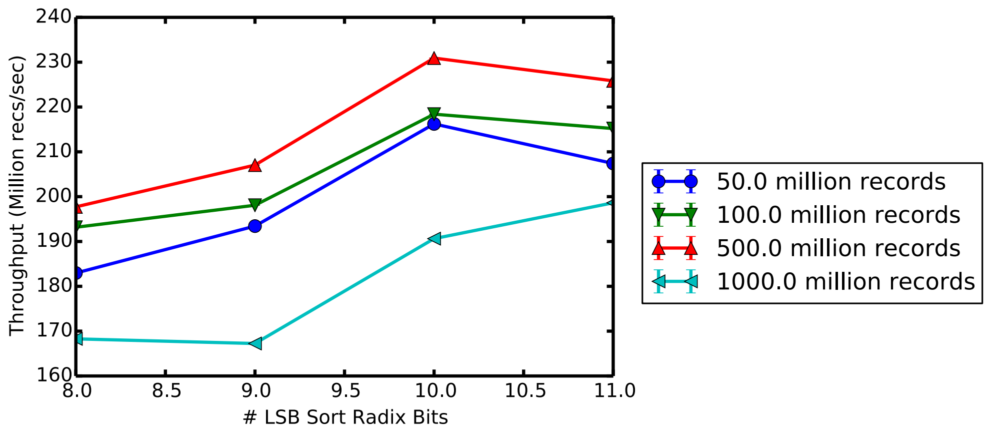


Figure 7.7: **Out-of-cache MSB-LSB Sort on Ivy Bridge**: We only show the performance curve corresponding to an initial 12-bit MSB partition, as this is always optimal. For relations smaller than 10^9 records, $b = 10$ is optimal, but $b = 11$ is marginally more efficient for larger ones.

that it was optimal to begin with twelve bits of MSB partitioning. The choice of LSB radix depends on the relation size, as Figure 7.7 shows.

7.3 Conclusion and Limitations

The base set of partition and sort operator generators included in `LibRofl` does indeed give rise to a design space from which the best algorithms can be found by a semi-automated empirical search. However, it must be noted that a few key optimizations are missing from this study, and adding these could change the tradeoffs presented above. Ressort does not presently implement the partition output buffer (Section 2.1) approach to minimize TLB and cache contention at higher radices, nor does it support any in-place variants of partitioning, or any vectorized algorithms. For these and other reasons, the throughput of Ressort's generated partition operator code is only about 75% of what [33] reported for a similar algorithm at radix $b = 10$ and only a quarter of what they measured for lower radices on a platform similar to our Ivy Bridge machine (albeit one with twice the number of sockets and memory bandwidth). Finally, we did not present any results from merge-based sort algorithms, and we have not evaluated any operators with relations of larger key widths, skewed distributions, or separate key and value columns. Nevertheless, Ressort did auto-generate C++ code for scalable partition and sort algorithm implementations.

7.4 Reflections and Future Directions

The provisional evaluation presented in this chapter represents an intermediate milestone in the development of Ressort. Aside from the performance limits imposed by the missing optimizations alluded to above, it is more generally clear that we have not succeeded in demonstrating the need outlined in Chapter 1 for structurally different decompositions of shuffle kernels on different platforms. This reflects to a great degree the incompleteness of Ressort's primitive operator catalogue (Table 3.1), which has inhibited a more exhaustive investigation of algorithmic variants.

This state of affairs raises the question of where time was spent in the course of this project, and whether that choice was a sensible one. It is true that building the infrastructure to support two DSLs, and to manage the execution and measurement of experiments, took effort that could conceivably have been better spent on the tuning and refinement of operator performance. We did try to leverage existing tools where possible, as in the case of GCC as the mechanism for machine code generation, but all stages of the front-end and all the code to represent, generate, transform, and translate IRep were written from scratch.

It is likely that much of this work could have been done more efficiently within the context of an existing framework for DSL design and code generation. The LLVM [27] and SEJITS [13] projects are obvious candidates for exploitation, but it is difficult to assess how well they could have accomodated ROFL and how much time their use might have saved. We did not build on top of SEJITS initially because that project did not support Scala. LLVM IR is a lower-level code representation than C, and would thus seem to be a strictly more laborious target, but it may become desirable to access this lower-level structure if it would provide a greater degree of control over the kinds of transformations contained within the low-level operator tuning space (Appendix C). Indeed, the compiler flow depicted in Figure 1.2 invites this possibility.

Despite their limited scope, the results in this chapter do nonetheless indicate that the core components of Ressort are in place, and it is our intent to continue building on top of them. In addition to implementing the missing primitives and investigating other algorithms such as joins, we would like to extend the compiler to support vector and SIMD architectures as well. Nearly all the algorithms discussed in this thesis have an inherent degree of data parallelism and could benefit from vectorization. Exploiting this property is even necessary to compete with previous hand-tuned algorithms presented in the literature. At a higher level, we would like to make Ressort's evaluation more realistic by integrating its auto-generated operators into an actual column store database. This would provide a better measure of how useful Ressort's optimizations ultimately can be, and would also provide motivation for extending its coverage beyond the sort/partition/join class of algorithms examined here. We hope Ressort can serve as a tool for compiling and parallelizing large fractions of whole queries, which will entail supporting operators such as projections and aggregations too. Query compilation will be the ultimate validation of ROFL's viability as a parallel programming language.

Appendix A

ROFL Compiler Front-End: Implementation Details

A.1 Internal Nested-Array Representations

Although the ROFL type system only differentiates arrays based on their degree of nesting and the type of records they contain, the front-end internally requires different array types for the result of different operations. Thus, while a `Split()` operator might yield a simple 2D array, a partition operator will produce a similarly nested array that also has an accompanying histogram data structure.

Inside the front-end, there are three levels of array representation that track these properties: ROFL arrays, IRep arrays, and IRep array wrappers. Each adds more detail: the first is purely structure, the second adds IRep symbol names and buffer allocation code, and the third encapsulates information related to loop induction variable offsets for use by operator code generators. The rest of this section describes these levels of representation in more detail.

ROFL Arrays

In the first stage of compilation, each DAG node's output is assigned a ***RoflArray***, which serves two purposes: (1.) it specifies the sizes and structures of all output data arrays of that node and (2.) serves as a unique identifier of that node's output wherever it recurs inside the structure of some other node's output (for example, two nodes' outputs might share the same histogram array, but contain two different record buffers). While the DAG's edges track immediate producer-consumer relationships between nodes, `RoflArray`s explicitly mark all instances of buffer re-use within and between nodes.

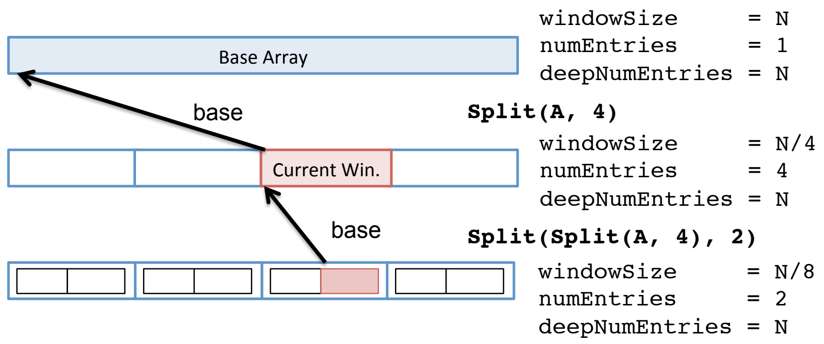


Figure A.1: Array Windows: All `RoflArray` types sub-divide a *base* array into `windows`

`RoflArray` Structure

All `RoflArray` types are either flat arrays, or flat arrays that have been sub-divided by layering additional structure on top. Figure A.1 illustrates this principle for a double application of `Split()`. The ultimate result as an array with three levels: a flat array of records at the base, one layer of sub-division on top, and finally, a second layer of sub-division upon that. A nested `RoflArray` data type contains a *base* pointer to the original array it sub-divides, and *metadata* that describes that nature of that division. The *deep base* is the final flat array at the bottom of any nested structure. Some structures may also include *ancillary structures* to determine precisely how their base arrays are sub-divided, and these are themselves `RoflArrays` with structures of their own.

Several terms are used throughout the rest of this section to describe the way in which `RoflArrays` slice up their bases. First, each sub-division results in smaller and smaller *windows*. A window is a contiguous [1] segment of the flat base array that will ultimately be processed by a canonical operator (Section C.1). A `Split(N)` operator thus reduces the *window size* as shown in Figure A.1 by its integer degree `N`.

Each layer of a nested `RoflArray` structure describes its view of the underlying record array through four parameters:

1. `deepNumEntries` is the number of records in the the flat base array at the deepest level of structure. This is the same for all layers in a `RoflArray`

2. `numElements` is the number of sub-arrays induced by a given level of array structure.

3. `windowSize` is the smallest extent of the deep base array

4. `deepNumSlices` is the total number of windows into which the deep base array is divided by all layers of nesting

[1]Contiguity is really an abstraction – a future `Rake()` operator, for example, might provide the illusion of contiguity over elements taken at some stride apart in the flat base array

These collectively supply all information needed to allocate buffers for one node based on the structure of its `RoflArray` inputs, as demonstrated in the example in Figure A.2.

Types of `RoflArrays`

`RoflArray`s are categorized hierarchically, and each type is considered to be flat, nested, or a "wrapper":

- **Flat arrays** don't contain any nesting, and just identify a buffer of records
- **Nested arrays** add a layer of structure on top of a any kind of array, and indicate how its elements should be accessed
  - *Sliced arrays* are the result of applying a `Split()` operator, and are simply views of their base arrays divided into N equally-sized segments
  - *Histogram-partitioned arrays* divide their base arrays with a histogram structure, which they contain as an ancillary `RoflArray`. Their sub-arrays are accessed via an index into the histogram to find the proper offset in the base.
- **Wrapper arrays** don't actually add structure, but add metadata to a flat or nested array:
  - *Histogram* arrays add the MSB and LSB used in the radix partitioning of which they are the result
  - *Reference arrays* are pointers to `RoflArray`s produced elsewhere, an so should not be allocated where they appear

A.2 `RoflArray` Buffer Allocation

`RoflArray` Structure Generation

As the initial operator DAG is created, a `RoflArray` is built for the output of each node according to the semantics of its ROFL primitive and based on the `RoflArray` structures of its inputs. To illustrate this procedure, we examine the case of a radix partition operator, shown in Figure A.2, which applies the following ROFL program to its input array `R` in parallel with up to `S` threads:

```
val op = {
   val splitInput = Split(R, S)
   MergeHist(
      MoveRecs(
         array = splitInput,
         hist  = BuildHist(splitInput, radix = R)))
}
```

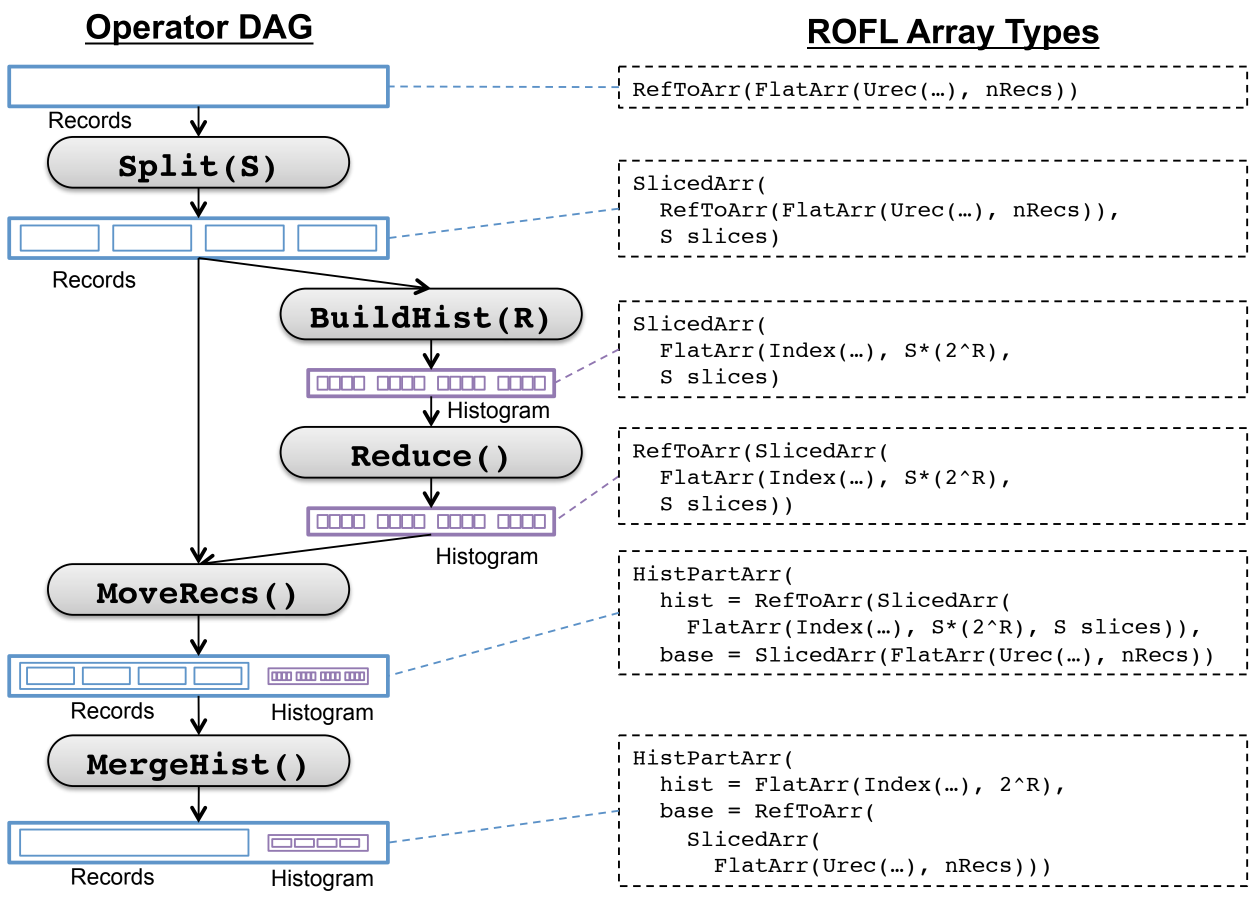

Operator DAG
ROFL Array Types
Records
Split(S)
Records
BuildHist(R)
Histogram
Reduce()
Histogram
MoveRecs()
Records
Histogram
MergeHist()
Records
Histogram
RefToArr(FlatArr(Urec(…), nRecs))
SlicedArr(RefToArr(FlatArr(Urec(…), nRecs)), S slices)
SlicedArr(FlatArr(Index(…), S*(2^R), S slices)
RefToArr(SlicedArr(FlatArr(Index(…), S*(2^R), S slices))
HistPartArr(hist = RefToArr(SlicedArr(FlatArr(Index(…), S*(2^R), S slices)), base = SlicedArr(FlatArr(Urec(…), nRecs))
HistPartArr(hist = FlatArr(Index(…), 2^R), base = RefToArr(SlicedArr(FlatArr(Urec(…), nRecs)))

Figure A.2: `IRepArray` Structures for a Parallel Partition Operator DAG. Purple boxes represent ancillary data structures, while blue ones denote record arrays.

The input to this DAG is a flat array of records–ROFL type `Arr(URec(...))`. As a `RoflArray`, this translates to a `FlatArr` type. Applying the `Split()` operator yields a `SlicedArr` that simply adds metadata–the number of slices–on top of the flat reference type.

The `BuildHist()` operator requires a histogram buffer be allocated for it to fill. However, since its input is the nested `SlicedArr` produced in the previous step, ROFL semantics require it process each slice independently, and in parallel if possible. This means a separate histogram copy per slice is needed, or $(2^R + 1)S$ elements in total [2] More generally, an ancillary data structure must be duplicated S times when created based on an input array for which `deepNumSlices` $= S$. Then, all upper layers of the input array's nesting structure must be reproduced on top of the newly-allocated flat array. Note, therefore, that the output

[2]Histograms are always padded with an extra element so that the length of the nth bucket can always be computed by subtracting $\mathrm{Hist}[n+1] - \mathrm{Hist}[n]$, even for the last bucket, without requiring an extra branch instruction in the inner loop to handle that special case.

of `BuildHist()` is also a `SlicedArr`, but its base is a bare `FlatArr` rather than a reference to one. This causes it to be allocated as a new buffer.

The prefix sum operation implied by `Reduce()` operates in-place, and thus returns a reference to the sliced array originally produced by `BuildHist()`.

Next, `MoveRecs()` allocates a new buffer at its output to contain records for each partition, for each thread, and reuses the same histogram array as produced by `BuildHist()`. The result is a compound array structure, `HistPartArr`, that contains a reference to an existing histogram, and a new `SlicedArr` whose base is a flat array equal in size to the original input relation. Note that this is now a doubly-nested array, since one index determines which partition to examine, and another which per-thread slice.

The reverse is true for the output of `MergeHist()`. This merges per-thread histogram data into a single unified view by discarding all but the highest-numbered thread's offset for each partition, leaving only the offsets of the partitions themselves. Thus, a new histogram buffer is allocated, while the record buffer is a reference to the one already allocated by `MoveRecs()`.

Conversion to `IRepArrays` and Allocation

Once a DAG has been constructed and `RoflArray`s assigned, the next compilation stage assigns an IRep identifier to each allocatable `RoflArray`, declares constants for metadata such as radix bits and slice lengths, and generates code to heap-allocate each base buffer array. The resulting array representations are called `IRepArray`s because they now associate each node's inputs and outputs with actual IRep code.

Deallocation code is also generated at this point. Although it would be possible in future versions of the compiler to walk the allocated DAG and, based on simple liveness analysis, deallocate each buffer at the last node that sees it, or even reuse it without calling the allocator again, the current version of the front-end is much simpler. It merely appends a deallocation statement for each buffer to the end of the code generated for the final output node of the DAG. To avoid deallocating the output buffers themselves, it walks the `IRepArray` structure of the output and marks all arrays visible at the output as non-deallocatable.

Array Wrappers and Code Generation

The final level of detail imposed on array structures prior to algorithm elaboration is the *`IRepArrayWrapper`*. Code generators for primitive operators use these wrappers as opaque handles via which to access an array's elements without needing to understand the details of its structure. Most code generators simply expect to process each record of a window, and merely need a means of accessing the nth element, along with the window length. Wrappers encapsulate this information, and control the generation of parallel for loops to process all sub-arrays of a node's input, ensuring that operator code generators need only generate the innermost record processing loop.

Windows and Offsets

Wrappers encapsulate the indices and offsets used as induction variables in nested array processing loops. At any point in time, the innermost such loop of an operator's implementation[3] will access a contiguous extent of records starting at some offset in the flat array at the base of the whole wrapper structure (the deep base). This extent is called the *current window*. Wrappers export as part of their API a means of obtaining a valid IRep expression that denotes the size of this window. They also generate code to access the nth element of that window.

Accessor State Mementos

The most natural way to element access expressions for complex array structures would be to substitute each base array's access expression into the next layer above it, gradually building up an lvalue AST. However, this naïve approach quickly produces an exponential explosion in accessor expression size[4]. The compiler solves this problem by declaring variables that track the offset of each wrapper layer's current window in its base array, and thus effectively memoize common subexpressions that would otherwise be replicated throughout a complex accessor expression. As the compiler generates each nested loop layer for a DAG node, it must also insert the accessor state memento update code for the corresponding layer of each input and output wrapper.

Leader and Follower Wrappers

Wrapper creation and nested loop generation are tightly coupled. For each DAG node, one array structure is elected as the *leader*, and an array wrapper is generated for it. Its offsets and indices will be set by the nested for loops emitted to implement the operator, and all other arrays accessed by that operator are treated as *followers*: their array wrappers will copy all the induction and offset variables used at each level of the leader. Usually the output wrapper of the first input to a DAG node is chosen as the leader.

Included in the wrapper API is a set of routines to generate follower wrappers from a given leader wrapper for any suitable `IRepArray`. Suitability will in many cases simply be defined as having the same degree of nesting, but operators that change the nesting level will have input and output arrays with different degrees. In the case of flattener nodes, the output arrays' wrappers are generated from the leader wrapper's base array (i.e. the array one level inside the outermost layer). For nester nodes, the leader wrapper is *padded* with an empty shell wrapper, and the output wrapper is generated as a follower of the padded input.

[3]unless it is a nested reduction

[4]In one experiment, a triply-partitioned array generated a C++ array subscript containing more than 2MB of code!

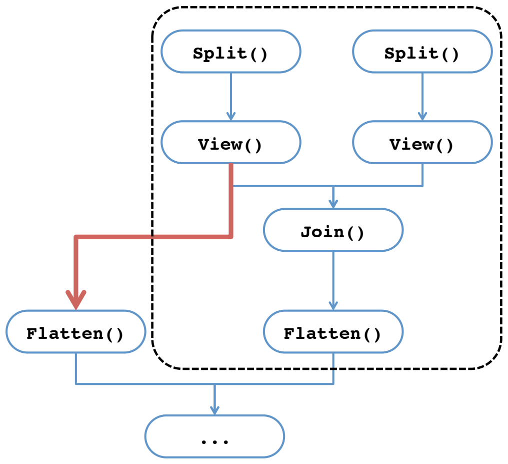


Figure A.3: A Non-fusible Operator DAG: The output of the left `View()` node is seen by a node outside of the candidate fusion segment, so it cannot be fused.

A.3 Operator Fusion: Exploiting Temporal Locality

Identifying Fusion Candidates

Candidates for fusion consist of all nodes between a set of *nesters*, which raise the degree of nesting, and a single *flattener*, which lowers it. The first step of the fusion transformation is to identify all such candidates (called *segments*) as $(\{\text{nesters}\}, \text{flattener})$ tuples, choosing only those that further satisfy the following constraints:

1. No internal node between the nesters and flattener may have an output that is seen by any node outside the fusion segment
2. No node inside the fusion segment may be a reduction

Constraint (1) is necessary because the output of an intermediate node may no longer be visible after the merged DAG has executed. Otherwise, reuse of internal buffers would not be possible. Figure A.3 illustrates a case where fusion cannot be applied without violating this constraint.

The second constraint results from the fact that reduction operations, by definition, require access to *all* sub-arrays of a nested array, and so cannot be forced to examine these one-at-a-time. This constraint is enforced by treating each reduction node as both a nester *and* a flattener. In the fusion segment identification algorithm described below, this results in segments being split along reduction nodes. Algorithm 2 identifies candidate segments for DAG fusion. Segments are found by examining all flattener nodes, and traversing backwards from each one towards the nester that produced the level of nesting that it removes. All nesters and flatteners encountered along the way decrease and increase the *level* value, respectively, until it is zero, in which case the desired nester has been found.

Algorithm 2 Fusion Segment Identification

```
1:  function FindNesters(node, level)
2:      $P \leftarrow parents(node)$
3:      if $lowersLevel(node)$ and not $isReduction(node)$ then
4:          return $\cup_{\{p \in P\}}$FindNesters$(p, level + 1)$
5:      else if $level = 1$ and  $raisesLevel(node)$ or $isReduction(node)$ then
6:          return $\{node\}$
7:      else if $raisesLevel(node)$ and not $isReduction(node)$ then
8:          return $\cup_{\{p \in P\}}$FindNesters$(p, level - 1)$
9:      else
10:         return $\cup_{\{p \in P\}}$FindNesters$(p, level)$
11:     end if
12: end function
13:
14: $Segments \leftarrow []$
15: for node $n$ in a depth-first pre-order traversal of the DAG do
16:     $N \leftarrow$ FindNesters$(n, 1)$
17:     if $lowersLevel(n)$ or  $isReduction(n)$ then
18:         if Segment $s = (n, N)$ does not violate Constraint (1) then
19:             $Segments \leftarrow Segments + [s]$
20:         end if
21:     end if
22: end for
```

Virtual Input and Output Nodes and Arrays

After packaging a fused segment into the body of its enclosing outer node, it is necessary to disconnect the inner DAG's uppermost (i.e. leaf) nodes from their inputs, as the elaboration procedure would otherwise traverse the inner DAG all the way up to the leaf nodes of the outer DAG. To effect this severance, we replace the inner DAG leaf nodes' inputs with *virtual nodes*. These are special "fake" operators with no implied computation, but with *virtual* output arrays of the appropriate type, and metadata linking them to the real output arrays of the enclosing node's inputs.

In the standard case where an input node's output array type supports parallel sub-arrays, the corresponding virtual array type is just a reference to the actual input array. However, when parallel sub-arrays are not allowed, a special virtual array is created whose length is equal to the maximum window size of the input array to which it is linked. This facilitates the space-saving optimization described above, wherein only enough buffer space for one window's worth of records is used at internal DAG operators. Once virtual input nodes of appropriate type have been selected, the output array types for all nodes in the inner DAG are regenerated. This ensures that any new input array shapes will be reflected all throughout the inside DAG. It also ensures that the layer of array structure produced by the nesters will not result in for loop generation for any internal nodes' wrappers, and that these will be marked as "controlled" by the loop generated for the outer node.

Virtualizing Output Arrays

Special care must be taken with the output array of the internal DAG for two reasons. First, it is the point of linkage between the internal DAG's possibly flattened array view and the real array that the enclosing node presents to the outside world. Second, any buffers allocated by the internal DAG that remain visible at the output node must be allocated as outputs of the *enclosing node* in order to ensure their liveness across the boundary.

Output virtualization therefore consists of these steps:

1. Traverse the entire internal DAG and identify a set A of all arrays allocated inside it

2. Examine the complete structure of the output array of the output node, and identify all references to arrays in A

3. Generate a copy of the output where all references to arrays in A have been replaced by the arrays themselves

4. Traverse the internal DAG again and replace all array allocations with references to the arrays allocated by the output

Appendix B

IRep: Code Transformations and C++ Generation

Most IRep constructs map straightforwardly onto C++ equivalents, but some require special treatment. Length-tracking arrays are converted to struct types, array operations are converted to for loops, and loop bound expressions are extracted into constant declarations. These needs motivate the inclusion of a general mechanism for writing transformations on type-checked IRep ASTs, of which this section describes several instances.

Common Transformations

Some transformations supplied by the IRep language layer are useful to all possible targets. Many of these considerably improve the legibility of auto-generated IRep programs by removing extraneous cruft. In all cases, transformations operate on type-checked ASTs where all local symbol names have been assigned globally unique identifiers consisting of their original identifier and a pointer to the *scope* (symbol table, or inside the compiler, a `TypeEnvironment`) in which they were declared.

AST Dataflow Analysis

Ressort supplies a very simple analysis pass to build a graph of dataflow between the scoped symbols of a type-checked AST. It constructs a table representing the immediate seen-by relation between scoped symbols by walking the AST and updating symbols' entries according to the set of rules in Definition 2.

Definition 2 *A symbol S is considered to be* seen by *a* root symbol *T iff S appears inside any expression in:*

1. *an assignment to T*
2. *the condition clause of an if-else statement that may modify T*

3. *the bounds or stride of a loop whose body may modify T*
4. *a function call where any actual parameter is an expression containing T*
5. *any argument to an array operation (see Section 5.4) on T*

By root symbol *we mean that T is the base of an lvalue expression AST.*

This notion of a root symbol in Definition 2 means, for example, in the case of an array subscript lvalue, that T appears as the name of the array, rather than in the subscript. Note that S will still be seen by T in the case of:

```
T[1+S] <-- 7;
```

because it appears inside an expression in an assignment to T.

Note also that this analysis is conservative, since S will be seen by T if T is assigned a reference to S *anywhere in the program.* It is also the case that conditional assignment of S to T adds S to T's seen-by table entry permanently regardless of whether or not the condition expression would or could ever be satisfied. Rule 4 arises from the fact that no inter-procedural analysis is performed, so it is simply assumed that all root symbols T in a function call see all expressions S appearing in any argument. The seen-by relation should thus be viewed as an upper bound on possible dataflow.

Dead Code Elimination

The main purpose of the dataflow analysis pass described above is to build a primitive dead code elimination (DCE) tool. Ressort's DCE works by iteratively computing the transitive closure of the seen-by relation for those symbols whose immediate seen-by sets are empty. This continues until no new symbols are found to be seen only by no others or only by others already in the set of dead symbols. Then, all declarations of dead symbols are removed from the AST during a depth-first traversal, during which time any blocks, loops, or if statements are removed whose bodies are empty or contain only `Nop`s.

A base case is necessary to ensure the entire AST is not pruned away. We define a notion of being seen by *the output* keep the output array of a compiled operator off the dead symbol set. Thus, the seen-by rules in Definition 2 are amended as follows:

- Any symbol inside a `Return()` statement is seen by the output
- Any symbol inside a `Printf()` is also seen by the output

Any symbol seen by the output will never have an empty seen-by set.

It should be noted that this DCE pass is not a very powerful one, owing the conservative dataflow analysis used as its input. It could be improved significantly if a static single assignment (SSA) transformation were applied prior to dataflow analysis, as this would prune all updates to variables *after* their actual output-visible values are captured. However, we deemed this effort unnecessary for the main use case of Ressort's DCE, which was simply to eliminate vast swathes of superfluous accessor state memento (Section A.2) updates generated by the frontend compiler.[1]

Translation to C++

Array and Record Structs

Since IRep arrays have an implicitly stored length, translation to C++ involves storing that length explicitly. Ressort handles this by generating a distinct C++ structure type for every type `T` used inside an IRep `IRArray(T)` type in a compiled operator's AST, and then replacing all references to lvalues with that type to structure-indirect lvalues. If the original value was an automatic variable, it a `HeapAlloc()` is also emitted at the modified declaration site. This is somewhat complicated in the case of nested arrays, where a loop is required to produce the intended allocation:

```
var a: arr_arr_IRInt[10][20]
```

The above pretty-printed IRep array declaration translates into:

```
Typedef _arr_IRInt {
  items: arrptr_IRInt
  len: Index
}

Typedef _arr_struct__arr_IRInt {
  items: arrptr__arr_IRInt
  len: Index
}

var a: _arr_struct__arr_IRInt
a.len <-- 20
a.items <-- HeapAlloc[ ArrExpr(struct _arr_IRInt,a.len) ]
ForPar(_arr_i_0 = 0...20 by 1) {
  Deref(a.items)[_arr_i_0].len <-- 10
  Deref(a.items)[_arr_i_0].items <--
    HeapAlloc[
        ArrExpr(BaseExpr(IRInt(false)),
        Deref(a.items)[_arr_i_0].len) ]
}
```

[1]This is not merely for aesthetic reasons or to assist in the debugging of compiled operators. There are various architectural reasons why some of these state declarations were duplicated, and though those declarations are theoretically idempotent, the C compiler targets did not support this usage.

Despite the efforts to avoid `malloc` (`HeapAlloc`) calls inside operators' record processing loops, this structure is unlikely to impact performance in practice, since nested arrays will rarely be used as automatic variables emitted by operator code generators (Appendix C).

We note also that the type names in this output is mangled: `arrptr__arr_IRInt` is really `Ptr(Arr(_arr_IRInt))`, a pointer to an array of `_arr_IRInt` structures. Since the name of a array struct type corresponds to the type of the array, it must be mangled into a permissible C++ identifier.

Function Linearization

In order to emit C++ code, the IRep translator must un-nest all nested function declarations. This is assisted by the constrained scope available to function bodies, which means that function declaration statements may simple by extracted and prepended before the original outer function declaration. Extracted functions are also renamed if another function of the same name would appear in the same global scope. For example:

```
Func f1(
    x: UInt,
    y: ptr_UInt) :  {
  Func f1(
      x: UInt,
      y: ptr_UInt) :  {
    Deref(y) <-- x*x
  }
  f1(x=x, y=y)
  f1(x=Deref(y), y=y)
}
```

Translates into

```
Func __LIN__f1_1(
    x: UInt,
    y: ptr_UInt) :  {
  Deref(y) <-- x*x
}
Func f1(
    x: UInt,
    y: ptr_UInt) :  {
  __LIN__f1_1(x=x, y=y)
  __LIN__f1_1(x=Deref(y), y=y)
}
```

Parallelization with OpenMP

The design of ROFL deliberately exposes as much parallelism as possible through IRep parallel for loops. The more independent sub-structures are described by a ROFL array's type, the more layers of nested parallel looping will be generated. This sounds like a strict benefit for efficiency; each parallel construct will induce a corresponding OpenMP loop in the C++ translation, and will supply the latter's compiler and runtime with more parallelism to exploit. However, it is not necessarily clear that so large a concession of control is optimal. As an alternative, Ressort supports a mode of operation in which OpenMP parallel for loops are generated only the outermost parallel IRep loops. We anticipate that future versions of the compiler will exploit parallel inner loops for vectorization, depending on the complexity of control flow inside.

Appendix C

Code Generation for Shuffle Kernel Primitives

The final stage of the frontend compiler described in Chapter 4 is *algorithm elaboration*. This entails the generation of IRep code (i.e. construction of an AST) that actually implements the primitive operators supplied by ROFL. In this chapter, we examine (1.) the common infrastructure and interfaces provided by the frontend compiler to code generators for each primitive, (2.) the space of *low-level tuning* choices that code generators must make, and (3.) details of individual operators' implementations. Chapter 7 will later present the impact of these parameters in our experiments.

C.1 Overview

Harnessing Scala to Build Parameterized Code Generators

For almost every ROFL primitive, there exists a corresponding *code generator* whose job it is to spell out in the intermediate language the actual algorithm implied by the semantics of that primitive in ROFL. A code generator resembles a ROFL operator generator of the sort described in Chapter 3 in that it is also a Scala program whose output is itself a program in the IRep DSL generated according to an algorithm-specific *configuration* like that shown in Figure C.1. We refer to this space of different output programs generated by different configuration parameters as the *low-level tuning space*, and discuss that space here for a few algorithms in particular.

Hiding Details of Relation Structure

Code generators mostly expect to transform one input array of records into another, and are not concerned with the details of how those records are organized into nested array structures. The frontend compiler therefore presents to code generators an interface that hides many of these details behind an `IRepArrayWrapper` data type. (Section A.1).

```
case class RecStreamConfig(
  unroll:         Int = 8, // Number of times to unroll the stream loop
  prefetch:       Int = 1, // number of SW prefetch instructions to emit
  prefetchStride: Int = 8, // stride of successive prefetch instructions
  prefetchOffset: Int = 8) // dist. from end of regular loads to start prefetch
```

Figure C.1: Record Stream Configuration: all code generators that process their input records in a sequential stream can ask the Ressort framework to unroll their main loop and prefetch future inputs wherever hardware prefetchers fail to.

Only a few kinds of operators break this abstraction by imposing constraints on the types of their input wrappers. For example, nested array reductions require a nested array wrapper. Such exceptions are marked in the first step of DAG generation at any node requiring special treatment. We distinguish *canonical operators* from these special ones, and use the former term to denote those that expect flat arrays of records as input and produce one as output. Code generators for canonical operators rely on the `IRepArrayWrapper` abstraction to indicate how many records to process (via the `windowSize` field), and what lvalue refers to the nth record in the input and output arrays.

Where nested array parallelism exists in a given operator composition, it is the job of the frontend compiler to emit the nested parallel for loops that exploit it; the code generator is responsible merely for innermost loop that processes records directly. Code generators need to produce this loop themselves because they might not examine each input record only once, and might access indices out of order.

C.2 Record Streams

Many code generators process their input relations sequentially in a streaming access pattern. By abstracting the loop generation logic for this pattern into a library, Ressort allows all *streaming operators* to benefit from shared transformation passes without any additional work on the part of the generator author. In particular, each such operator accepts a streaming configuration (Figure C.1) that specifies whether the library should unroll the record processing loop and insert software prefetch instructions. The streaming library automatically performs this transformation and inserts any necessary fix-up code to handle the case where the input size is not a multiple of the unrolling factor. The prefetch is expressed as an IRep `Prefetch()` statement, which in the GCC-based C++ backends results in a `__builtin_prefetch()` directive. We examine the impact of this parameter on a partitioning operator for the ARM Cortex A9 platform in Section 7.1.

Although this example is of limited general-purpose applicability due to the prevalence of highly-accurate hardware prefetching support in modern microarchitectures, it demonstrates the flexibility of Ressort's embedded DSL approach to code generation. It is useful for operators that do very little computation on their input records and are thus dominated by loop bookkeeping. Moreover, we intend in the future to add analogous support for more interesting access patterns such as hash table probes.

C.3 Partitioning Primitives

Much of the code generation currently implemented by Ressort pertains to histogram-based partitioning. We examine some of these primitives in more detail in this section, but note that they share a few common features.

Wrappers for Histogram-based Arrays

Firstly, they do break somewhat the abstraction of flat record arrays that other code generators rely on, as they require a specific kind of ancillary data structure, the histogram, which they manipulate in various ways. At the time of this writing, the requirement of a histogram-based input for some operators is still not reflected in the ROFL type system and type checker, so violations of that requirement manifest instead during the later elaboration phase of compilation in the frontend.

Some of these primitives also break the abstraction by requiring a nested array as input. The `MergeHistograms()` operator is one prominent example, since its job is to consolidate per-thread histograms into a single, master histogram. In this case, as in others, we rely on the `isInternalReduction` flag, which gets set for any `MergeHistograms()` DAG node to indicate that it removes one layer of nesting from one of its own internal data structures. This causes an appropriately-modified output wrapper to be generated for the `MergeHistograms()` code generator.

Histogram-Specific Array Operations

Histogram-based algorithms also require certain more specialized array operations: prefix sums and array rotations. Ressort supports these through dedicated constructs in the IRep language (Section 5.4) accessed through methods supplied by `IRepArrayWrapper` objects.

Histogram Prefix Sums

For example, the ROFL `ReduceHistograms()` primitive must be applied to the output a histogram building operator in order to produce one whose entries indicate the *offsets* of each partition's beginning in the output buffer, as opposed to a mere count of the number of records in each partition. This is naturally accomplished as a prefix sum, whose domain of operation is the entire extent of the histogram wrapper's current window (Section A.1). Thus, the code generator for `ReduceHistograms()` simply returns the result of the wrapper's built-in `prefixSum()` method, which imposes the offsets and indirections specified by all layers of the histogram's wrapper type in order to finally produce an IRep `PrefixSum()` statement over a suitably translated range.

However, `ReduceHistograms()` also supports a flag to indicate whether the histogram is divided among multiple threads (i.e. it is itself a nested array), in which case this reduction must (1.) check that the input array is nested and (2.) use a slightly different algorithm to take account of the per-thread semantics. One concern that might arise in this context is the serialization bottleneck imposed by a large prefix sum reduction in the case of a high-radix operator and large number of threads. The obvious solution to this difficulty is to introduce a parallelized prefix sum tree code generator. The architecture of Ressort's compiler makes this a rather trivial substitution of one module for another, but as we have not yet seen histogram reduction as a bottleneck in any experiments thus far, we have neglected to implement this feature.

Post-Move Histogram Restoration

Aside from the prefix-sum and histogram merge operations mentioned above, these partitioning methods require one additional array operation: after the evaluation of `MoveRecordsHist()`, which distributes records to their designated partitions, the histogram's contents will correspond to the locations of the *ends* of each partition in the output buffer, rather than their beginnings. Because it is the offset of the beginning that is needed during subsequent operations, the histogram must be *restored* to the post-`ReduceHistograms()` state.

The `RestoreHistogram()` primitive accomplishes this task by generating an *array rotation* operation, which, like prefix sums, is another IRep language construct provided by the wrapper API. It simply shifts all elements in a histogram array over by one, wrapping the (in this case) the rightmost element around to the zeroth position. Although this issue could have been addressed in other ways, we think supporting the left- and right-rotate operations as native language constructs will allow for more natural expression of similar tasks in the future.

C.4 Sorting Primitives

As Section 3.3 explains, the ROFL language includes few sorting algorithms as primitives, as most efficient sorting methods derive from compositions of other, non-sorting operators. Currently only insertion sort is supplied directly, though a future version may provide, for instance, a bitonic sort network generator. Insertion sort itself is nevertheless a useful illustration of how low-level tuning in code generators works.

Insertion Sort

A straightforward insertion sort code generator's main body is concise enough to fit entirely in the few lines of Figure C.2:

```
// Default, untuned case: just emit two nested loops
def genInsertionSort(
       input: IRepArrayWrapper,
       output: IRepArrayWrapper): IrAst = {
    ForBlock(i, input.windowSize, {
        DecAssign(tmp, recType, input.access(i)) +
        DecAssign(j, Index(), i) +
        While((j > 0) && (Key(output.access(j-1)) > Key(tmp)), {
            Assign(output.access(j), output.access(j-1)) +
            Assign(j, j-1)
          }) +
        Assign(output.access(j), tmp)
      })
}
```

Figure C.2: Default code generator for the `InsertionSort()` ROFL primitive: buffer layout is hidden by wrapper types, only two nested loops are returned as an IRep AST

This excerpt demonstrates two important features: first, that a modicum of syntactic sugar permits the expression of IRep programs almost natively alongside their supporting Scala scaffolds, and second, that even simple programs are effectively *templates*, or recipes for applying a primitive operator over any input and output buffer structure containing records of any type. `Key()` expressions constrain comparison to only those bytes of a record that matter for sorting, which may otherwise be arbitrarily typed. The only fixed type is `Index`, which denotes an offset into a buffer and normally translates to C++ `size_t`.

A more interesting variant of this simple recipe is one that attempts to perform insertion sort entirely in registers. The C++ translation of any IRep output from the above generator will necessarily compile into a loop of $O(N^2)$ dynamic loads and stores, since every insertion requires accessing array elements with the loop induction variables as indices. It is possible, however, to "unroll" this loop completely, and generate a different instruction sequence for an insertion at each possible index in a small, fixed-size array. By eliminating all but the initial loads into the buffer and stores back out to memory, the number of dynamic memory references decreases to $O(N)$.

Figure C.3 depicts a simple recursive Scala function that generates IRep code to insert the nth element into a register-allocated buffer. Assuming some other code is generated to declare these registers–really just automatic variables entrusted to the C++ compiler's judgment–the `insertNth` code is passed a function `buf()` that tells the name (lvalue AST) of the nth of these. Recursive Scala function calls elaborate a large IRep AST.

This will of course result in redundant code sequences when done for all $n < N$, as each n duplicates the work of all $m < n$. Somewhat surprisingly, GCC with the `-O3` option manages to coalesce all redundant such code sequences into one compact block, at least on certain

```
def insertNth(n: Integer)
      (implicit elem: LValue, buf: Int => LValue): IrAst = {
    if(n > 0) {                                // Scala ‘if‘ controls codegen
      IfElse(Key(buf(n-1)) > Key(elem), { // IRep ‘If‘ to be emitted
          (buf(n) := buf(n-1)) +
          insertNth(n-1)                       // Recursive Scala call results
        },                                     // in complex AST structure
        (buf(n) := elem))                      // ‘buf‘ function tells how
    } else {                                   // to access nth array elem,
      (buf(n) := elem)                         // in reg-alloc’d buffer
    }
}
```

Figure C.3: Scala code for an “unrolled insertion sort” IRep code generator: Returns an AST implementing the insertion of the nth element into an insertion-sorted output array.

architectures. Section D.2 reports the results of this method from a few of them. Although there is an obvious increase in static code size, the cost of a few compulsory instruction cache misses can be recuperated in reduced dynamic instruction count if the number of sub-problems is sufficiently large.

Appendix D

Supplementary Performance Data

This appendix contains supplementary performance data from the evaluation in Chapter 7. For the most part, we tried to present results from only one platform there if all others followed similar curves, deferring those instead to this appendix.

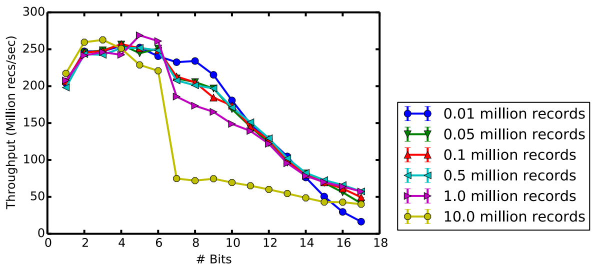
D.1 Partitioning Results

Analysis of Memory Bandwidth Usage

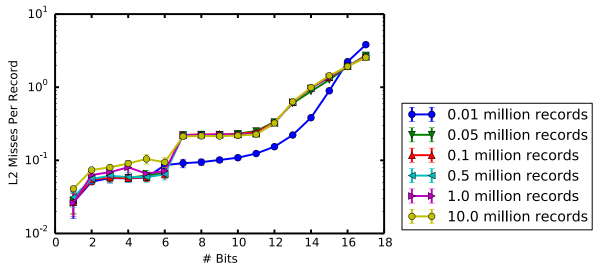
To calibrate our radix partitioning results against the memory bandwidth measurements in Table 6.2, we propose a simple analytic model of memory traffic for the partitioning operator. For a KP32 relation R of N records, the following memory references are generated for each record:

1. Build Histogram:
    a) Read 8 bytes from each $R[n]$ – **sequential**
    b) Read 4 bytes from Hist[(part)] – **in cache**
    c) Write 4 bytes to Hist[(part)] – **in cache**
2. Move Records:
    a) Read 8 bytes from each $R[n]$ – **sequential**
    b) Read 4 bytes from Hist[(part)] – **in cache**
    c) Write 4 bytes to Hist[(part)] – **in cache**
    d) Write 8 bytes to Output[(offset)] – **random**

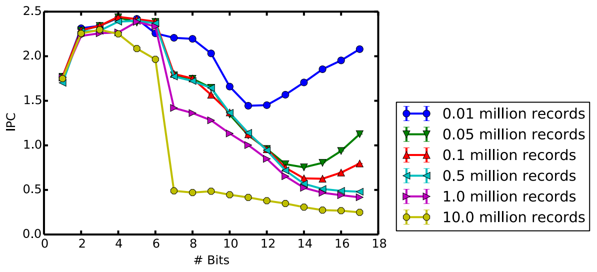
In this model, we assume the lines marked "in cache" really do correspond to cache hits, that lines marked "sequential" read the indicated number of bytes from DRAM, and that the one marked "random" generates:



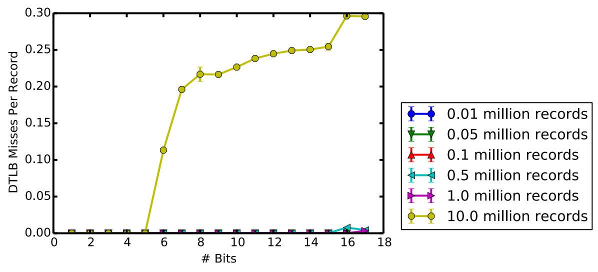
(a) Throughput



(b) L2 Misses / Record



(c) Instructions per Cycle



(d) DTLB Misses / Record

Figure D.1: **Single-thread Radix Partitioning on an Ivy Bridge (50 outer trials)**: There is a moderate upswing in IPC as the radix surpasses $R = 2^{14}$ (in the case of the smallest relations), but this is due to an increased instruction count from histogram prefix sum phase, which contributes at least $16K > 0.01M$ operations to the overall runtime and is quite visible when the relation size is small.

- For low radices ($r < 8$): one 8B read and one 8B write = 16B
- For high radices: one 64B read and one 64B write (two cache lines) = 128B

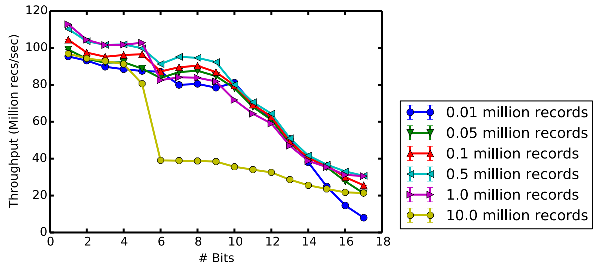
However, since peak throughput can only be realized in the low-radix regime, we assume a transfer of 16 bytes for the output step, giving a total transfer of 24 bytes per record. Thus,

$$BW_{\text{DRAM}_i}(N) = 24\frac{\text{bytes}}{\text{rec}}N \tag{D.1}$$

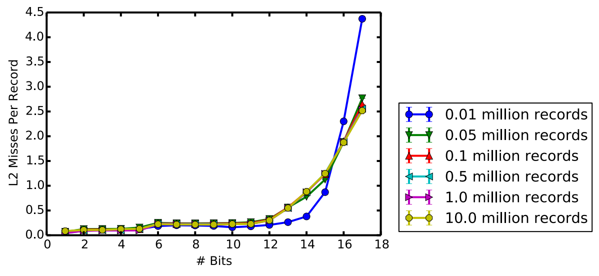
We use this model to compute the estimated fraction of theoretical and measured peak bandwidth achieved by our radix partition operator on each machine. We report numbers only for the radix that gives each platform's highest throughput at the maximum number of threads. Table 7.1 contains these estimates.

Ivy Bridge

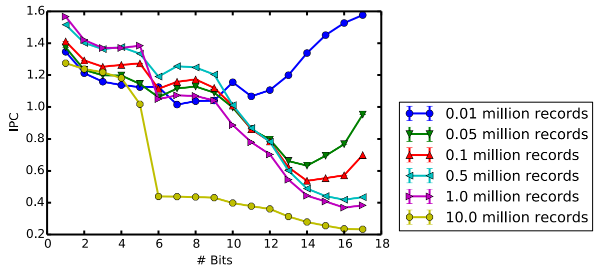
Partitioning on the Cortex A9 (NVIDIA Tegra3) The contrast between the performance profiles for 16M- and 128K-record relations cleanly reveals the difference in throughput



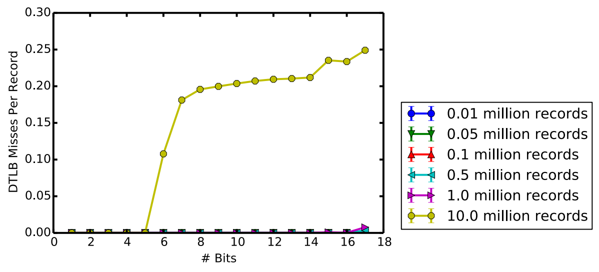
(a) Throughput



(b) L2 Misses / Record

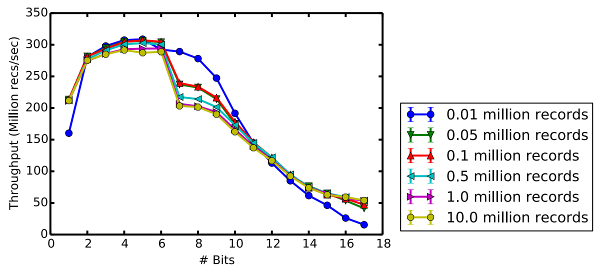


(c) Instructions per Cycle

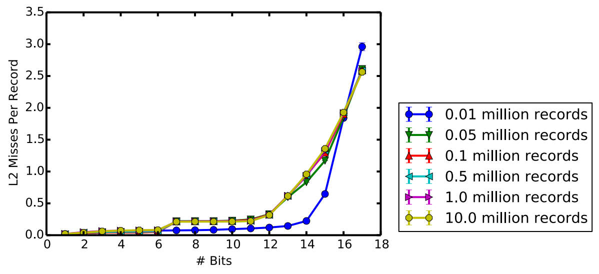


(d) DTLB Misses / Record

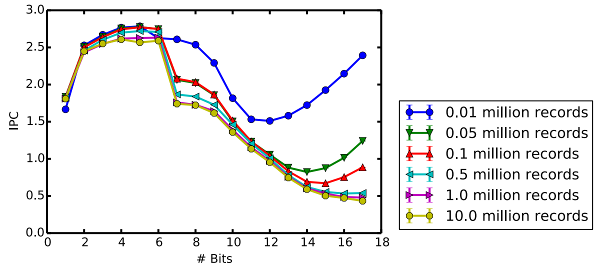
Figure D.2: **Single-thread Radix Partitioning on Boxboro (50 outer trials).**



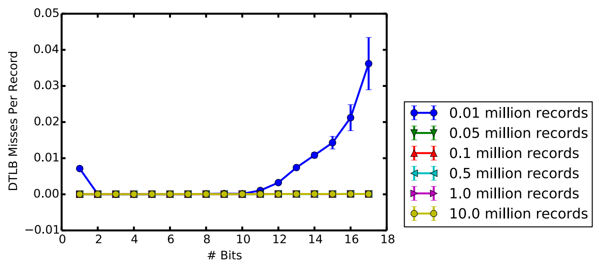
(a) Throughput



(b) L2 Misses / Record

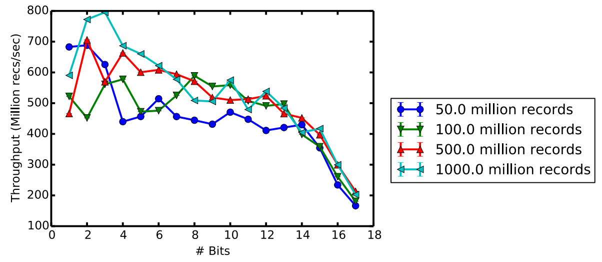


(c) Instructions per Cycle

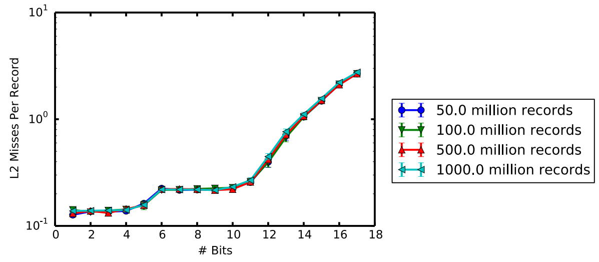


(d) DTLB Misses / Record

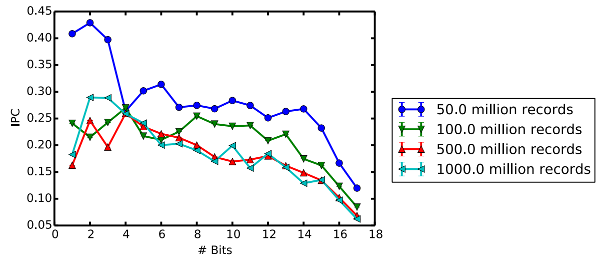
Figure D.3: **Single-thread Radix Partitioning on Haswell (50 outer trials).**



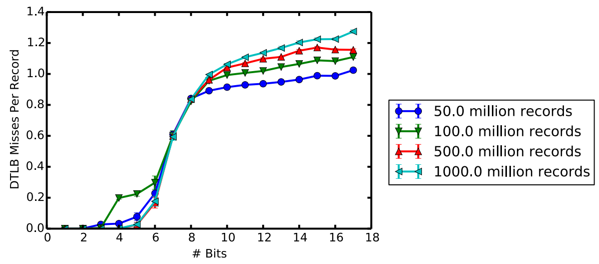
(a) Throughput



(b) L2 Misses / Record

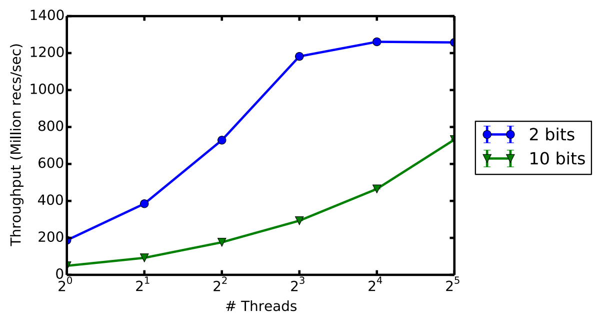


(c) Instructions per Cycle

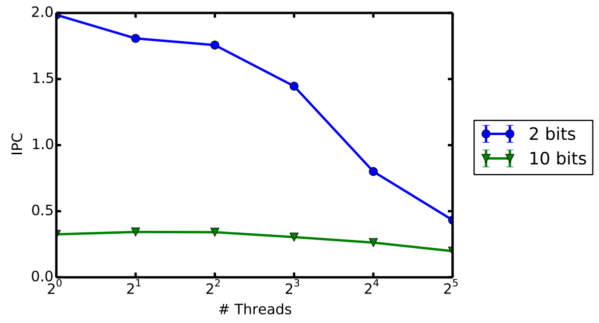


(d) DTLB Misses / Record

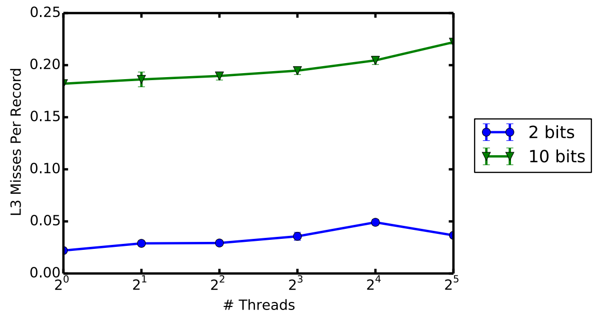
Figure D.4: **40-thread Radix Partitioning on Boxboro (50 outer trials).**



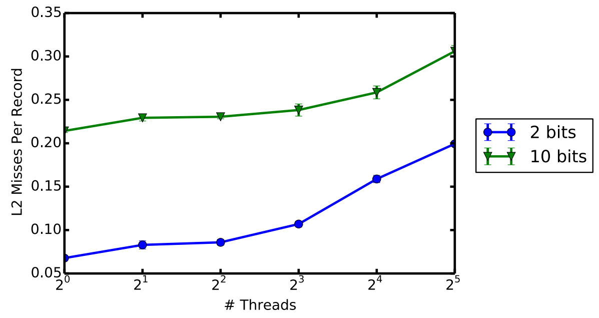
(a) Throughput



(b) Instructions / Cycle

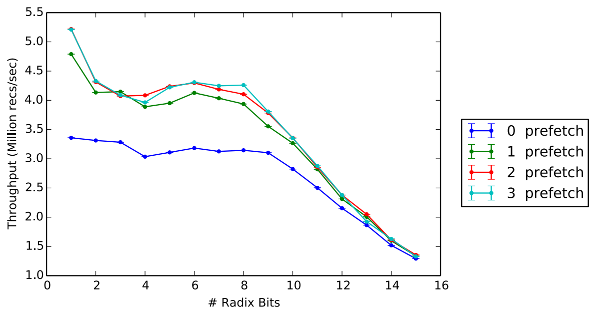


(c) L3 Misses / Record

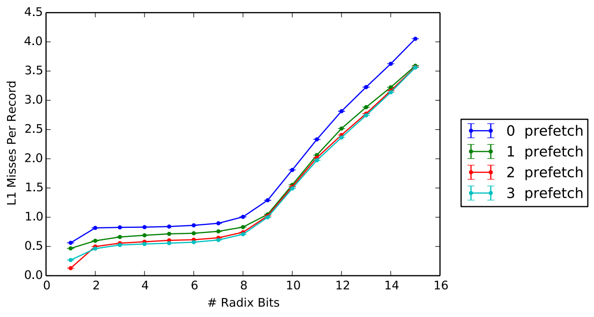


(d) L2 Misses / Record

Figure D.5: **Scaling Radix Out-of-Cache Radix Partitioning on Ivy Bridge**: 10^9 Records



(a) Throughput



(b) L1 Misses per Record

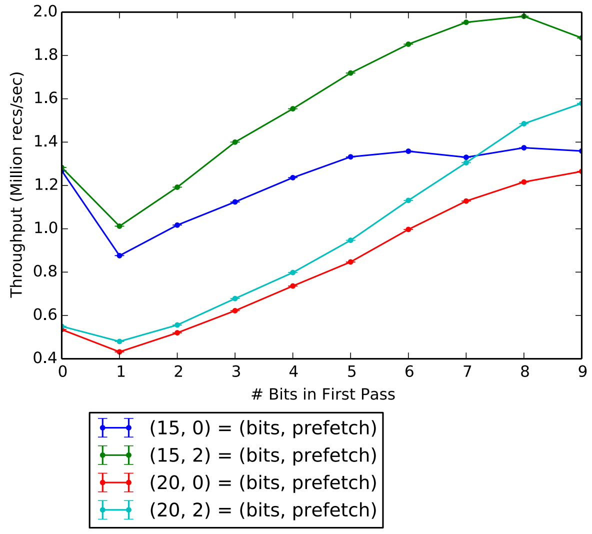
Figure D.6: **Software Prefetch-Assisted Partitioning on an NVIDIA Tegra3**: Results obtained from a record stream configuration (Section C.2) with 4 degrees of unrolling, 8 record stride and offset.

for problems that fit in the L2 cache versus those that do not (recall from Table 6.1 that the Tegra3 has a 1MB L2, which is just enough to contain 128K*8B=1MB records).

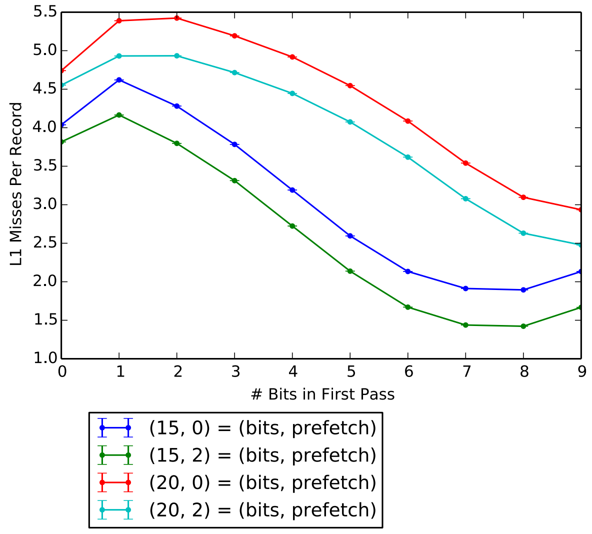
It is striking that throughput is so bad for large relation sizes *even at low radices*, which should make essentially streaming accesses to the outer levels of the memory hierarchy. It is regrettable that we do not have access to L2 miss statistics from the Tegra3 board, which would indicate whether or not this is the case. Since the L1 miss, DTLB miss, and instruction counts per records are all the same in the low radix regime, and since the L1 miss rate is roughly one per record, it seems likely that these L1 misses are also compulsory L2 misses for the larger relations.

A simple change to the ROFL compiler configuration (Section C.2) tests this hypothesis by generating partition code with software prefetch instructions automatically inserted into an unrolled record processing loop. Figure D.6 shows the resulting throughputs obtained for 16M records with different numbers of prefetch instructions in a loop unrolled four times. Each prefetch is taken starting at an offset of eight records (one cache line) with a stride of one cache line. Throughput improved by approximately 25%. While this is far from the order of magnitude gap between large and small sizes, it does suggest that with additional tuning, this gap could be closed.

On the Tegra 3, it is worthwhile to use multiple passes even at $r = 15$, and even more profitable at $r = 20$. Figure D.7 omits the $r = 12$ curve in order to highlight the impact of software prefetch: in the case of $r = 15$, multi-pass is only worthwhile when prefetch is used; without it, the difference is negligible.



(a) Throughput



(b) L1 Misses per Record

Figure D.7: Multi-pass Partitioning on Tegra 3: 16M records

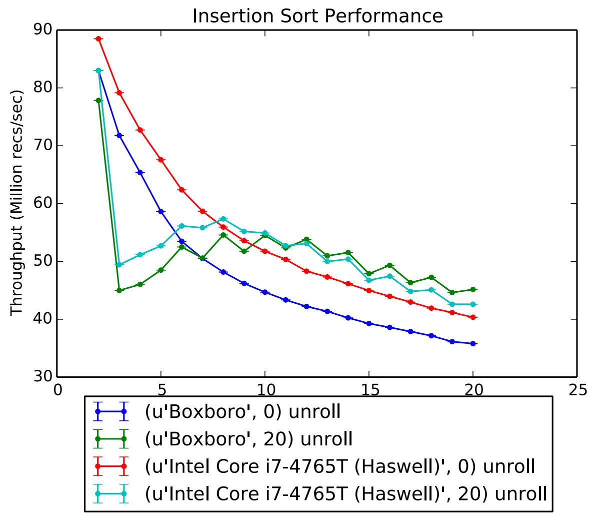
D.2 Sorting Results

Insertion Sort

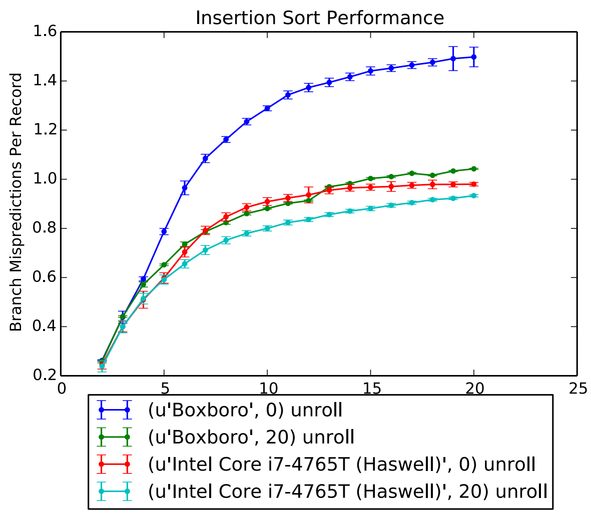
Insertion sort is the only sorting primitive currently supplied by ROFL. Its quadratic runtime makes it suitable only for the Small Sort problem size. As such, it is really a benchmark of core microarchitecture, rather than memory hierarchy or interconnect performance, so we report only results for a single thread.

Results from the Intel platforms Figure D.8 shows sorting for the two Intel platforms. The newer Haswell core consistently outperforms the older Nehalem. One obvious microarchitectural difference is a stronger branch predictor in Haswell, which reduces the misprediction rate by a third in the steady state. The mispredict rate in either case levels off at one per record because insertion sort has one highly predictable for loop back branch, and an inner, data-dependent while loop branch that's taken n times for the nth element in the worst case. In the steady state, this inner branch will always be predicted taken and then mispredicted when a record exits the insertion loop. The higher misprediction rate on the Nehalem must result from failure to reach the predict-always-taken state for that inner branch.

Impact of "Unrolling" The unrolled insertion sort code generator (Figure C.2 in Section C.4), which produces fully in-register insertion sort code, did improve throughput somewhat. The improvement is more pronounced on Nehalem than on Haswell, and seems due

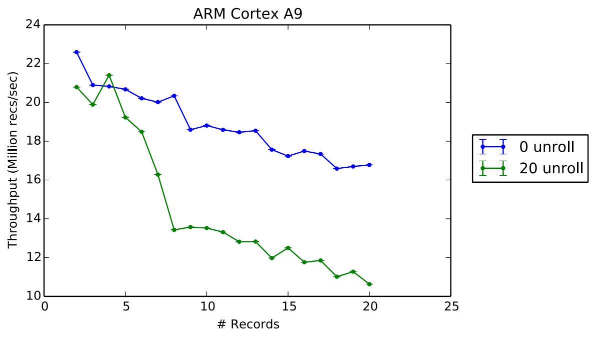


(a) Throughput

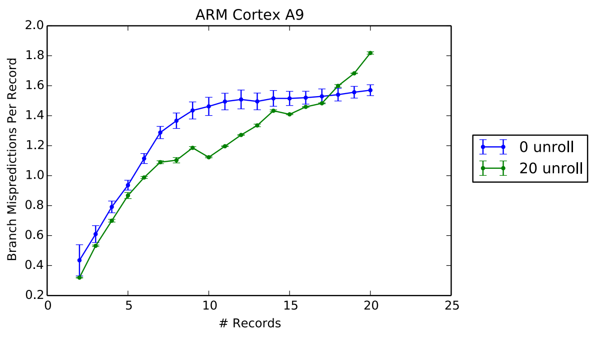


(b) Branch Mispredictions

Figure D.8: Small Sort: Insertion Sort on the two Intel platforms. Zero unroll indicates normal insertion sort was used, while 20 unroll means insertion functions for up to 20 elements were generated (Figure C.2).

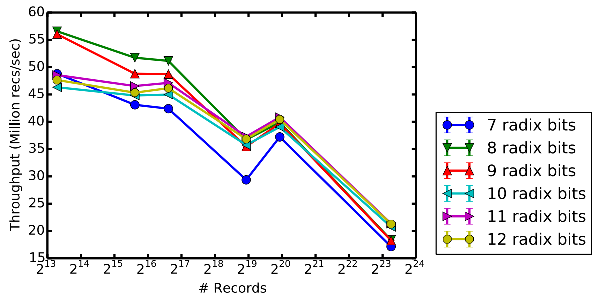


(a) Throughput

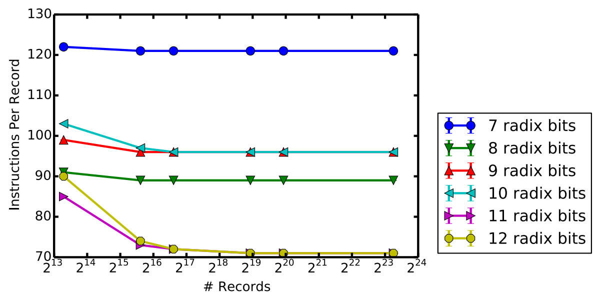


(b) Branch Mispredictions

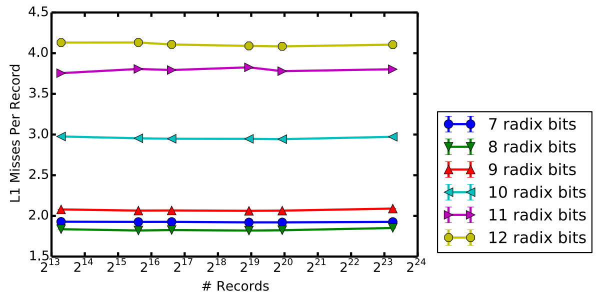
Figure D.9: Small Sort: Insertion sort on the Tegra 3



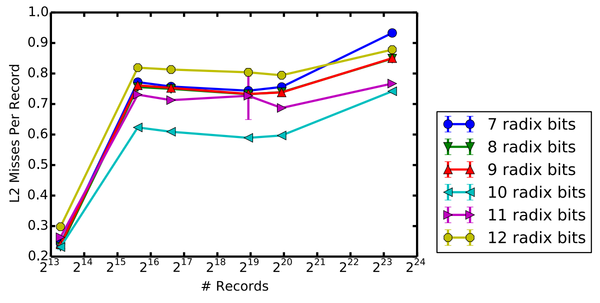
(a) Throughput



(b) Instructions per Record



(c) L1 Misses / Record



(d) L2 Misses / Record

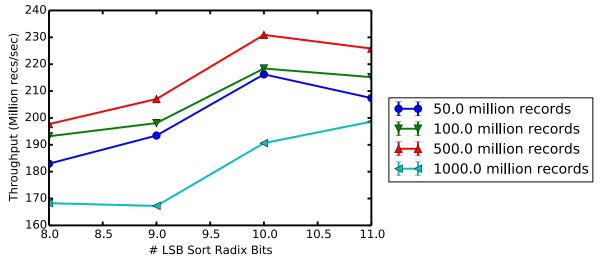
Figure D.10: **Single-Threaded In-Cache LSB Sort on Ivy Bridge**

mostly to higher branch prediction accuracy. In fact, the efficiency difference on Nehalem is roughly the equal to the difference between the unrolled code's performance across the two machines.

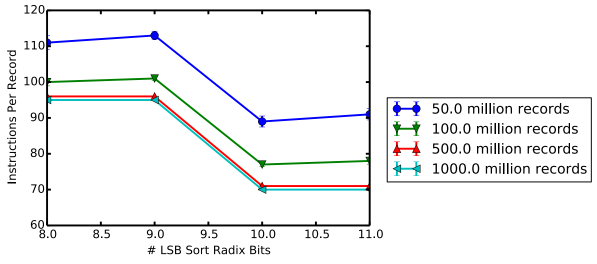
Results from the Tegra 3 It is unsurprising that the throughput curve of the Tegra 3's Cortex A9 core lies entirely below the minimum throughput of both Intel platforms. [1] The lower clock rate, simpler pipeline, and higher non-CISC instruction count all contribute to this effect. Unrolling actually degrades throughput, since the compiler was unable to perform the same code scheduling as it did in the x86 case, and so the dynamic instruction count is *higher* on ARM rather than lower.

LSB and LSB-MSB Sort Results

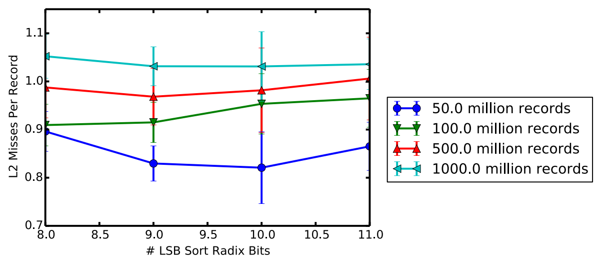
[1]However, it has an odd profile: whereas both Intel machines' throughput curves exactly track the expected $O(N/N^2) = O(1/N)$ pattern, the Tegra 3's throughput degrades linearly.



(a) Throughput



(b) Instructions per Record



(c) L2 Misses / Record

Figure D.11: Single-Threaded In-Cache MSB-LSB Sort on Ivy Bridge with 12-bit MSB Partition

Bibliography

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively parallel sort-merge joins in main memory multi-core database systems". In: *Proceedings of the VLDB Endowment* 5.10 (2012), pp. 1064–1075.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a Scala embedded language". In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1216–1225.

[3] C. Balkesen, J. Teubner, G. Alonso, and M.T. Ozsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*. 2013, pp. 362–373. DOI: `10.1109/ICDE.2013.6544839`.

[4] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. "Multi-core, Main-memory Joins: Sort vs. Hash Revisited". In: *Proc. VLDB Endow.* 7.1 (Sept. 2013), pp. 85–96. ISSN: 2150-8097. DOI: `10.14778/2732219.2732227`. URL: `http://dx.doi.org/10.14778/2732219.2732227`.

[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. *Efficient Main-Memory Hash Joins on Multi-Core CPUs: Does Hardware Still Matter?* Tech. rep. ETH Zürich. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.362.4020&rep=rep1&type=pdf`.

[6] Kenneth E Batcher. "Sorting networks and their applications". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM. 1968, pp. 307–314.

[7] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. "Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology". In: *25th Anniversary International Conference on Supercomputing Anniversary Volume*. ACM. 2014, pp. 253–260.

[8] Spyros Blanas, Yinan Li, and Jignesh M Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs". In: *SIGMOD Conference*. 2011, pp. 37–48.

[9] Spyros Blanas, Yinan Li, and Jignesh M. Patel. “Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’11. Athens, Greece: ACM, 2011, pp. 37–48. ISBN: 978-1-4503-0661-4. DOI: `10.1145/1989323.1989328`. URL: `http://doi.acm.org/10.1145/1989323.1989328`.

[10] Spyros Blanas and Jignesh M Patel. “Memory footprint matters: efficient equi-join algorithms for main memory data processing”. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 19.

[11] Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. “Implementation of a portable nested data-parallel language”. In: *Journal of parallel and distributed computing* 21.1 (1994), pp. 4–14.

[12] Peter A Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *CIDR*. Vol. 5. 2005, pp. 225–237.

[13] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. “SEJITS: Getting productivity and performance with selective embedded JIT specialization”. In: ().

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. “Introduction to algorithms second edition”. In: *The Knuth-Morris-Pratt Algorithm, year* (2001).

[15] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[16] David J DeWitt and Robert Gerber. *Multiprocessor hash-based join algorithms*. University of Wisconsin-Madison, Computer Sciences Department, 1985.

[17] Rhys Francis, Ian Mathieson, and Linda Pannan. “A fast, simple algorithm to balance a parallel multiway merge”. In: *PARLE’93 Parallel Architectures and Languages Europe*. Springer. 1993, pp. 570–581.

[18] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. “Cache-oblivious algorithms”. In: *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE. 1999, pp. 285–297.

[19] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. “Vector extensions for decision support dbms acceleration”. In: *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE. 2012, pp. 166–176.

[20] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. “VSR sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors”. In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 26–38.

[21] Jiong He, Mian Lu, and Bingsheng He. “Revisiting co-processing for hash joins on the coupled cpu-gpu architecture”. In: *Proceedings of the VLDB Endowment* 6.10 (2013), pp. 889–900.

[22] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. “Vectorized data processing on the cell broadband engine”. In: *Proceedings of the 3rd international workshop on Data management on new hardware*. ACM. 2007, p. 4.

[23] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. “GPU join processing revisited”. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM. 2012, pp. 55–62.

[24] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. “Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs”. In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1378–1389.

[25] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. “Application of hash to data base machine and its architecture”. In: *New Generation Computing* 1.1 (1983), pp. 63–74.

[26] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. “The vertica analytic database: C-store 7 years later”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1790–1801.

[27] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[28] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU”. In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 451–460.

[29] Rich Martin. “A Vectorized Hash Join”. In: *unpublished course report, University of California at Berkeley* (1996).

[30] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. “PAPI: A Portable Interface to Hardware Performance Counters”. In: *In Proceedings of the Department of Defense HPCMP Users Group Conference*. 1999, pp. 7–10.

[31] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. “Hash-Partitioned Join Method Using Dynamic Destaging Strategy”. In: *Proceedings of the 14th International Conference on Very Large Data Bases*. VLDB ’88. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 468–478. ISBN: 0-934613-75-3. URL: `http://dl.acm.org/citation.cfm?id=645915.671817`.

[32] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proceedings of the VLDB Endowment* 4.9 (2011).

[33] Orestis Polychroniou and Kenneth A Ross. “A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 755–766.

[34] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. “SPIRAL: Code Generation for DSP Transforms”. In: *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”* 93.2 (2005), pp. 232–275.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[36] Nadathur Satish, Mark Harris, and Michael Garland. “Designing efficient sorting algorithms for manycore GPUs”. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–10.

[37] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. “Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 351–362. ISBN: 978-1-4503-0032-2. DOI: `10.1145/1807167.1807207`. URL: `http://doi.acm.org/10.1145/1807167.1807207`.

[38] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. “Cache Conscious Algorithms for Relational Query Processing”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc. 1994, pp. 510–521.

[39] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. “Vectorization vs. compilation in query execution”. In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM. 2011, pp. 33–40.

[40] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O’Neil, et al. “C-store: a column-oriented DBMS”. In: *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment. 2005, pp. 553–564.

[41] Richard Vuduc, James W Demmel, and Katherine A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[42] R Clint Whaley and Jack J Dongarra. “Automatically tuned linear algebra software”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 1998, pp. 1–27.

[43] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. "Navigating big data with high-throughput, energy-efficient data partitioning". In: *ACM SIGARCH Computer Architecture News* 41.3 (2013), pp. 249–260.

[44] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. "Q100: the architecture and design of a database processing unit". In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM. 2014, pp. 255–268.

[45] Marco Zagha and Guy E. Blelloch. "Radix Sort for Vector Multiprocessors". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 712–721. ISBN: 0-89791-459-7. DOI: 10.1145/125826.126164. URL: http://doi.acm.org/10.1145/125826.126164.

[46] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. "Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures". In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1374–1377.

[47] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. "MonetDB/X100-A DBMS In The CPU Cache." In: *IEEE Data Eng. Bull.* 28.2 (2005), pp. 17–22.

[48] Marcin Zukowski, Niels Nes, and Peter Boncz. "DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing". In: *Proceedings of the 4th international workshop on Data management on new hardware*. ACM. 2008, pp. 47–54.