LINQits: Big Data on Little Clients

Eric S. Chung†, John D. Davis†, and Jaewon Lee‡

†*Microsoft Research Silicon Valley*
‡*Department of Computer Science and Engineering, POSTECH*

ABSTRACT

We present LINQits, a flexible hardware template that can be mapped onto programmable logic or ASICs in a heterogeneous system-on-chip for a mobile device or server. Unlike fixed-function accelerators, LINQits accelerates a domain-specific query language called LINQ. LINQits does not provide coverage for all possible applications—however, existing applications (re-)written with LINQ in mind benefit extensively from hardware acceleration. Furthermore, the LINQits framework offers a graceful and transparent migration path from software to hardware.

LINQits is prototyped on a 2W heterogeneous SoC called the ZYNQ processor, which combines dual ARM A9 processors with an FPGA on a single die in 28nm silicon technology. Our physical measurements show that LINQits improves energy efficiency by 8.9 to 30.6 times and performance by 10.7 to 38.1 times compared to optimized, multithreaded C programs running on conventional ARM A9 processors.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Adaptable architectures, data-flow architectures, high-level language architectures

General Terms

Design, Performance

Keywords

Co-processor accelerator, database, query language, big data, mobile, FPGA, ASIC

1. INTRODUCTION

Smartphones, tablet computers, and other emerging client devices have become a major catalyst for the post-PC revolution—it is estimated that by the end of 2013 nearly 1.2 billion units will be shipped world-wide, surpassing the total number of desktops and servers in the wild [15]. These devices are the sources and gateways to the world of Big Data [1], providing the opportunity to access, synthesize, and interpret information continuously. Due to stringent energy demands coupled with the recent failure of Dennard Scaling [10, 18, 14, 42, 9], today's client devices are highly constrained in their capabilities—with many emerging, disruptive applications out of immediate reach, such as augmented reality, continous speech recognition, interactive personal agents (e.g., "Watson" on a phone), and novel user experiences.

To extend the capabilities of future client devices, commercial system-on-chips (SoC) are evolving into rich, heterogeneous systems that combine conventional processors with multiple specialized, low-power accelerator cores on a single die [35, 8, 39]. Although hardware specialization is a critical ingredient for improving energy efficiency, system architects face a multitude of new challenges: (1) how to gracefully integrate custom hardware into existing software ecosystems, operating systems, and programming languages, (2) how to select custom core functionality without compromising on re-use, flexibility, and composability, and (3) how to cope with rapidly changing application requirements in a diverse mobile software ecosystem. Furthermore, fabrication costs and times are increasing, so features mapped into custom hardware must be selected judiciously, thus far limiting the "safe" accelerator candidates to functions such as video decoding, encryption, and graphics [39].

In this paper, we present LINQits[1], a flexible and composable framework for accelerating data-intensive applications using specialized logic. LINQits eschews traditional hardware-software boundaries (e.g., ISA) in favor of directly accelerating a declarative subset of the C# programming language called LINQ (Language Integrated Query). LINQ is designed to operate upon data sets or collections and exposes first-class language constructs for manipulating collections using rich query operators (similar to SQL). Unlike traditional query languages, LINQ allows the programmer to embed user-defined anonymous functions that enable elegant ways to express rich algorithms [49].

The LINQits framework provides a pre-tuned accelerator template that can be flexibly configured and mapped on to a heterogeneous SoC with reconfigurable logic or an ASIC. LINQits extracts declarative tasks in a LINQ query and configures the template to support those tasks efficiently in hardware. The template offers several key properties: (1) the ability to accelerate constrained, user-defined functions embedded within LINQ queries, (2) the ability to efficiently process large dataset sizes using multi-pass partitioning strategies that avoid excess spilling from limited on-chip memory, and (3) a customized memory system that coalesces memory accesses for improved efficiency. In general, LINQits can be applied to any system that services LINQ queries (e.g., a server or desktop) or traditional database languages (e.g., SQL). In energy-constrained settings, LINQits can vastly improve the data- and sensor-processing capabilities of future client devices.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'13 Tel-Aviv, Israel
Copyright 2013 ACM 978-1-4503-2079-5/13/06 ...$15.00.

[1]A portmanteau of LINQ and Circuits.

By targeting a stable, existing language such as LINQ for hardware acceleration, we illustrate a guiding principle behind our work—to leverage the *infiltration* [21] of language extensions in modern programming languages and to use these extensions as a natural boundary that separates traditional imperative-style code and specialized code. There could be other stable language alternatives. Our selection of LINQ is borne out of pragmatism: (1) we do not need to invent a new language for accelerators, (2) LINQ is well-suited for "big data" computations and is a natural entry and exit point in the program, (3) LINQ only requires a constrained set of operators that can be built very efficiently to leverage the expertise of logic-level designers, (4) LINQ allows embedded user-defined functions, enabling greater computational expressivity, (5) LINQ is portable and can be automatically parallelized to run on a distributed cluster of machines [24] or a low-power reconfigurable device as we show in Section 7, and (6) LINQ is a language standard with stable operators that are unlikely to disappear in the near future, making it a viable target for ASICs. Our work makes the following contributions:

- *We present the hardware design for LINQits, a tightly-coupled processor and accelerator fabric optimized for processing data collections.*
- *We show that LINQits offers a flexible and programmable hardware template that enables a single building block for various LINQ operators such as `Select`, `SelectMany`, `Where`, `Join`, `GroupBy`, and `Aggregate`.*
- *We present divide and conquer techniques in hardware that virtualize on-chip memory resources—making it possible to handle large data sets and to make off-chip memory accesses vastly more efficient.*
- *We show, using physical measurements of an RTL prototype on a Xilinx ZYNQ processor, that LINQits provides higher performance (10.7 to 38.1 times) and lower energy consumption (8.9 to 30.6 times) than optimized C code running on low-power, ARM A9 cores.*

2. BACKGROUND

Language-Integrated Query (LINQ) was introduced to Microsoft's .NET Framework [32] in 2007 and exposes first-class language constructs for manipulating sequences of data items (referred to as *collections*). LINQ was initially supported in C#, F#, and Visual Basic and has since been ported to Java, PHP, and other programming languages. Using LINQ, programmers can express rich computations in a high-level declarative language abstraction while remaining insulated from run-time implementation decisions. LINQ supports a large set of operators based on functional data parallel programming patterns and has been used to implement a variety of applications including machine learning algorithms, linear algebra, and even body tracking for the Microsoft Kinect sensor [49, 6, 31]. The seven essential LINQ operators are listed in Table 1 and should appear familiar to those who have used the SQL language.

Many LINQ operators accept one or more user-defined functions to process elements in a collection (a property that distinguishes LINQ from traditional SQL). These anonymous user-defined functions are often written in short-hand as `x => f(x)`, which maps a single variable `x` to a result `f(x)` [31]. Figure 1 shows how data extraction and filtering can be performed using these functions. First, L4 applies the `Where` operator to the initial source collection, using an anonymous function to retain items with sales over $100. The resulting collection is then passed into the `Select` operator (L5), which maps the original item into a new data type made up of the customer's zip code and payment type. L8-L9 show the equivalent filter and map operations using traditional SQL-like syntax. In LINQ, query execution is deferred until the elements are actually needed in a consuming process (e.g., L11).

Operation	Meaning
`Where`	(Filter) Keep all values satisfying a given property.
`Select`	(Map) Apply a transformation to each value in the collection.
`Aggregate`	(Fold, Reduce) Combine all values in the collection to produce a single result (e.g., max).
`GroupBy`	Create a collection of collections, where the elements in each inner collection all have a common property (*key*).
`OrderBy`	(Sort) Order the elements in the collection according to some property (*key*).
`SelectMany`	Generates a collection for each element in the input (by applying a function), then concatenates the resulting collections.
`Join`	Combine the values from two collections when they have a common property.

Table 1: LINQ Operators (re-printed from [31]).

```
1.  // Object-oriented syntax with anonymous functions
2.  var results =
3.    SomeCollection
4.      .Where(cust => cust.sales > 100)
5.      .Select(cust => new { cust.zipCode,
6.                            c.paymentType });

7.  // Equivalent SQL-style syntax
8.  var results = from cust in customer
9.          where cust.sales > 100
10.         select new {cust.zipCode, c.paymentType};

11. foreach(var v in results) ... // do work
```

Figure 1: A LINQ excerpt illustrating two ways of expressing filtering and map operations.

K-means Example. Figure 2 gives a more sophisticated example of using LINQ and user-defined functions to implement K-means Clustering [31]. The goal of K-means is to partition N points along a D-dimensional space into K clusters. In Lloyd's algorithm, a collection of centers are first initialized with seed values followed by a set of iterative refinement steps. During each step, every point in a collection is associated with a nearest center using a distance function (L1-L14). Subsequently, for each center, all of its nearest points are averaged and normalized to form a new center. This process repeats (using the new centers as input) until convergence (or a bound) is reached.

In L17 of Figure 2, the `GroupBy` operator is first applied to an unordered collection of points. The user-defined function `NearestCenter` for `GroupBy` computes an integer key that represents the closest center to each point. This key is used by `GroupBy` to sort each of the points into K sub-collections. In L18, each of these sub-collections are mapped (using `Select`) into single values by summing up and normalizing all of the points in each sub-collection using the `Aggregate` operator. These sub-collections of normalized averages now form a new collection of centers for the subsequent iteration of K-means.

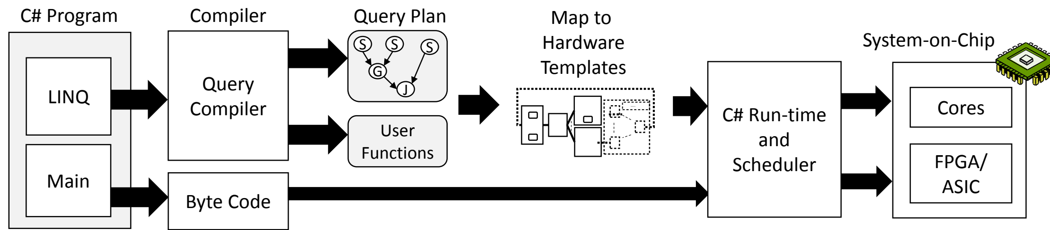

Figure 3: The LINQits Software-to-Hardware Flow. The query compiler is provided by the Dandelion compiler and runtime for distributed heterogeneous systems [38].

```
1.  int NearestCenter(Vector point, Collection centers) {
2.    int minIndex = 0;
3.    int curIndex = 0;
4.    double minValue = Double.MaxValue;
5.    foreach (Vector center in centers) {
6.      double curValue = (center - point).Norm2();
7.      if (minValue > curValue) {
8.         minValue = curValue;
9.         minIndex = curIndex;
10.     }
11.     curIndex++;
12.   }
13.   return minIndex;
14. }

15. Collection Step(Collection points,
16.                 Collection centers) {
17.   return points.GroupBy
          (point => NearestCenter(point, centers))
18.      .Select(group => group.Aggregate((x, y) => x + y) /
19.                       group.Count());
20. }

21. Collection Steps(Collection points,
22.                  Collection centers, int n) {
23.   for (i=0; i < n; i++) centers = Step(points, centers);
24.   return centers;
25. }
```

Figure 2: K-means Clustering (Lloyd's algorithm) implemented using GroupBy, Select, and Aggregate in C# and LINQ.

3. LINQITS OVERVIEW

To set the stage on how end-to-end LINQ hardware acceleration is carried out, Figure 3 illustrates our proposed software-to-hardware framework. We envision a deployment scenario where application-specific accelerators for LINQ queries are generated automatically in a centrally-managed repository (e.g., app store). The generated accelerators can either be mined automatically and fabricated as a general ASIC that supports many applications or targeted on a per-application basis to an FPGA.

As illustrated on the left of Figure 3, a programmer writing in a high-level managed programming language (e.g., C#) embeds LINQ queries to perform computations on collections of data. The LINQits framework performs an ahead-of-time compilation step to generate the hardware blocks for run-time execution. This compilation process would typically occur prior to an application's deployment from the central repository.

Query Plan Optimization. During the ahead-of-time compilation step, the LINQ queries are compiled into a query plan represented as a graph of computation nodes and communication edges. A node encodes important information including: (1) the operator (e.g., `Select`, `Join`, etc.) (2) the collection's element type (e.g., `int`, `float`), (3) the node's producers and consumers (i.e., communication edges), and (4) the user-defined functions discussed in Section 2. In the current software framework we are leveraging (called Dandelion [38]), each computation node reads all of its inputs from memory and/or storage[2] and writes the results back. To reduce communication, the query plan optimizer may choose to re-order or fuse computation nodes in the graph. For example, fusing `GroupBy` and `Aggregate` nodes can greatly reduce communication to storage or memory (relative to running each of the operators in isolation).

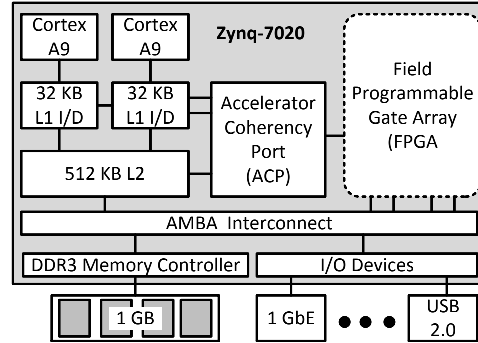

Figure 4: ZYNQ SoC overview.

Mapping to Hardware Templates. In the next stage of the ahead-of-time compilation step, the information from the nodes in the query plan are used to configure a pre-designed Hardware Template (HAT). The HAT is a special IP block that is highly parameterized at the RTL-level and can be customized to suit the particular needs of the query plan (e.g., which operators to support, what data types to use, the number of functional units, and the embedded user-defined functions). The HAT also provides hardware support for fused operators such as `GroupByAggregate`, used in some applications such as K-means and KeyCount (discussed in Section 7). The HAT must also be configured to match the target system-on-chip characteristics such as the memory datapath and area constraints. Section 5 will later describe in greater detail how the HAT supports the major operators presented in Section 2.

Run-time Scheduling. In our system, not all LINQ queries can be mapped automatically due to various constraints in the hardware templates. If the user-defined functions are not side-effect free or operate on complex data structures (e.g., linked lists), the C# run-time has the option of deferring the execution to software, which preserves

[2]Our current implementation only handles in-memory collections; enabling LINQits to process data from storage directly is a straightforward extension.

the original application's intent. Another scenario to consider is the need to amortize the cost of initiating the HAT between software and hardware. Selecting a problem size that is too small may slow down the application, negating the benefits of hardware acceleration. In the last stage of Figure 3, the C# run-time has the opportunity to decide dynamically whether one or more nodes of the optimized query plan should execute in hardware or software. If acceleration is not possible or profitable, the run-time can transparently fallback to execution in software.

4. TARGET PLATFORM

As a proof-of-concept, the LINQits framework is prototyped on a new heterogeneous processor called the ZYNQ Programmable System-on-Chip (SoC) [48]. ZYNQ is a commercially-available SoC that exemplifies a significant milestone in heterogeneous processors coupled with programmable logic. The ZYNQ processor combines dual ARM Cortex-A9 processors with FPGA programmable logic—operating in a small power envelope of 0.5-2W. An instance of the ZYNQ architecture is shown in Figure 4. Each A9 core has 32KB of instruction and data caches, a shared 512KB L2 cache and a variety of peripherals, including FPGA fabric. The FPGA fabric is *coherent* with the cores' L1 and L2 caches through the Accelerator Coherency Port (ACP). Finally, both the cores and the FPGA have access to 1GB of DRAM through the AMBA AXI bus. More system details are given in Table 2 in Section 7. For our purposes, the ZYNQ platform is an excellent proxy for studying the opportunity of hardware specialization in future, low-power client devices such as smartphones and tablets, because it offers similar core capabilities and memory capacity.

Run-time Operation. When a LINQits-enabled application is launched, the LINQits-aware C# run-time must preconfigure the hardware to operate in tandem with managed code. In an FPGA, a bitstream can either be programmed at boot-time or on a per-application basis, depending on how many applications use LINQits. In the ZYNQ platform we target, when a LINQits hardware template is instantiated and running in hardware, a memory-mapped set of registers enables the ARM cores to set the desired operation (operator selection), configure which user-defined functions to activate, store pointers to where shared memory structures are kept, and provide signals to the ARM cores for completion.

FPGA vs. ASIC. In an FPGA implementation of LINQits, only the features required by a specific or set of applications would be instantiated, given the reconfigurability option of the FPGA. In an ASIC, all operators must be instantiated prior to fabrication to provide high coverage across a large spectrum of LINQ applications; the ASIC must also support multiple configurable data types at run-time. In an ASIC, the hardware template could be augmented with software-programmable cores that enable even more flexibility in the template—we leave this aspect of LINQits to future work.

5. OPERATOR CHALLENGES

A distinguishing characteristic of LINQits is the ability to handle a wide range of both simple and complex operators. Existing commercial [22] and academic [33] systems focused primarily on simpler queries such as `Select` and `Where`

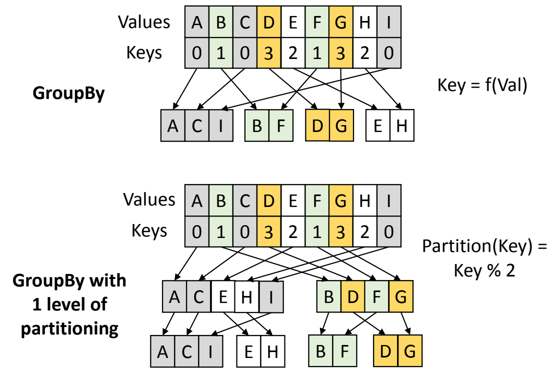

Figure 5: Example of `GroupBy`.

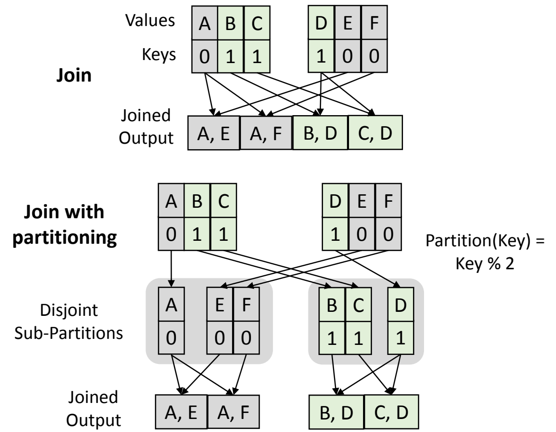

Figure 6: Example of `Join`.

while delegating more complex queries such as `GroupBy` and `Join` to conventional general-purpose processors (or simply do not support them). These complex operators introduce significant difficulties due to their irregular memory access characteristics and potentially large memory footprints that exceed aggregate on-chip memory capacities.

Overall, the HAT architecture supports six of the seven major operators described in Section 2 (`OrderBy` is the exception). In this section, we devote our discussion to what is needed to accelerate two of the most complex queries, `GroupBy` and `Join`. Section 6 will later discuss how we trivially extend the HAT to support the remaining operators.

GroupBy and Join Algorithms. Figure 5 shows the computational pattern of `GroupBy`, where a collection of unsorted values are "grouped" into four disjoint partitions based on a user-defined value-to-key function. In hardware, the user-defined functions can be trivially mapped into a parallel array of functional units that process each input from memory independently. However, the distribution of keys and final partition sizes are not known *a priori*, making it difficult to know for each key-value pair where the final values should be written in memory. A typical implementation of `GroupBy` in software uses a hash table (indexed by the element key) to dynamically create and update arrays for the groupings. On the first insertion of a given key, an array is allocated and the element is added to it. Subsequent elements that match the same key are simply appended. Unfortunately, hashing negatively impacts cache and DRAM locality, especially in large-scale data set sizes.

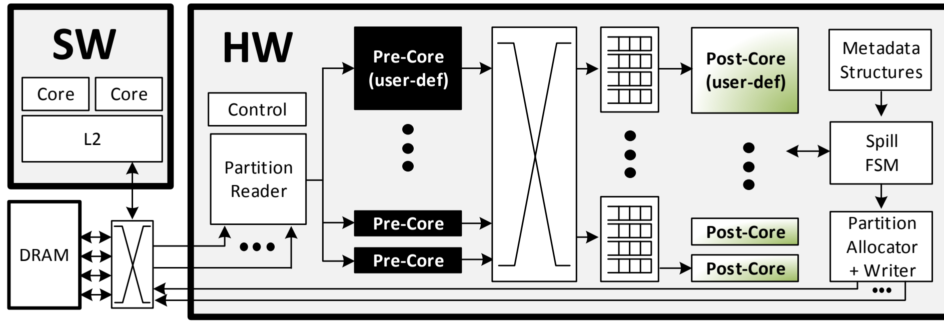

Figure 7: LINQits hardware template for partitioning, grouping, and hashing.

The `Join` operator also poses similar challenges, as shown in Figure 6. In a `Join`, two large data collections are matched by a common key followed by a user-defined function that takes both values associated with the common key and computes an output (also based on a user-defined function). For very large data set sizes, a hash join is typically used to avoid an expensive nested, pairwise comparison. In the hash join, elements from the smaller of the two partitions are inserted into a hash table. Elements of the larger partition are then used to probe the hash table for matching keys. Like in `GroupBy`, implementing a naive hash join similarly exhibits poor caching and memory behavior.

Partitioning. Partitioning is a well-known software strategy for improving the locality and performance of `GroupBy` and `Join` queries in databases [16, 12, 41] and distributed run-times [24]. As illustrated in Figure 5 (bottom), a pre-processing step is first carried out by computing a key partition function on each key value. In the example, the partition function divides the initial collection into two disjoint sub-partitions, each with non-overlapping keys. In the final `GroupBy`, the two disjoint sub-partitions are processed with independent hash tables with non-overlapping key values.

During the partitioning steps, fewer random accesses are made to DRAM because sub-partitions are built up contiguously (versus writing in one pass to the final partitions directly). Partitioning comes at the cost of $O(np)$ operations, where n is the collection size and p is the number of partitioning passes. At the end of the partitioning phase, each individual sub-partition would be inserted into a private hash table scaled to fit within on-chip memories. The same optimization can be applied to `Join` (bottom, Figure 6). Here, two disjoint sub-partitions are created followed by smaller hash-joins that can potentially fit in smaller data structures such as caches.

6. GENERAL HARDWARE APPROACH

Our key idea behind the LINQits hardware template is to adopt the same divide-and-conquer approaches used in software to make hardware-based hash tables practical to implement for queries that operate on large-scale dataset sizes. In the context of a small SoC such as ZYNQ, the HAT must operate with a limited amount of on-die memory storage (typically 1MB or less). Naively implementing a hardware-based hash table using this limited storage can hurt performance due to excessive conflicts and spills. A key strategy we take is to develop hardware that performs in-place, multi-pass partitioning prior to actually carrying out the final hash operation needed in a multi-level `GroupBy` or `Join`. We also observe that both hashing and partitioning can leverage the same physical structures with minimal impact to hardware complexity and area.

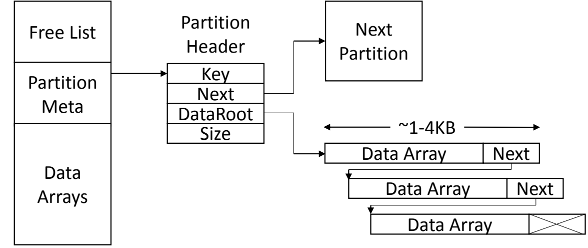

Figure 8: Data Structures for LINQits partitions.

Figure 7 illustrates the physical organization of our proposed HAT architecture, spanning both hardware and software on the ZYNQ platform. The heart of the HAT is a data steering and storage engine that enables fine-grained re-partitioning of data into multiple hash entries, implemented using a network-on-chip and physical queues.

At run-time, data is streamed in from main memory and processed by a collection of pre- and post-core modules that implement the user-defined functions described in Section 2; the outputs of the pre-cores are used by the network-on-chip for steering key-value pairs into destination queues. When a queue reaches capacity, it is streamed out into main memory to form new sub-collections that become the sources for subsequent invocations of the HAT.

For operators that associate computation with items already sorted into queues (e.g., `Aggregate`), post-cores are placed in the back-end of the HAT pipeline to handle post-processing operations. Finally, the metadata structures shown in Figure 7 contain information about created partitions (i.e., partition ID and address table), groups, keys, element counts, and other relevant information required to traverse the data structures for the various passes of the LINQ operators. In the next sections, we discuss two important modes of operation: partitioning and hashing. Section 6.3 will later describe how these building blocks implement six out of the seven essential LINQ operators from Section 2.

6.1 Partitioning Mode

Figure 9 (top) shows the hardware structures for operating the HAT in partitioning mode.

Partition Reader. The partition reader is a specialized DMA engine responsible for reading bulk data stored in *par-*

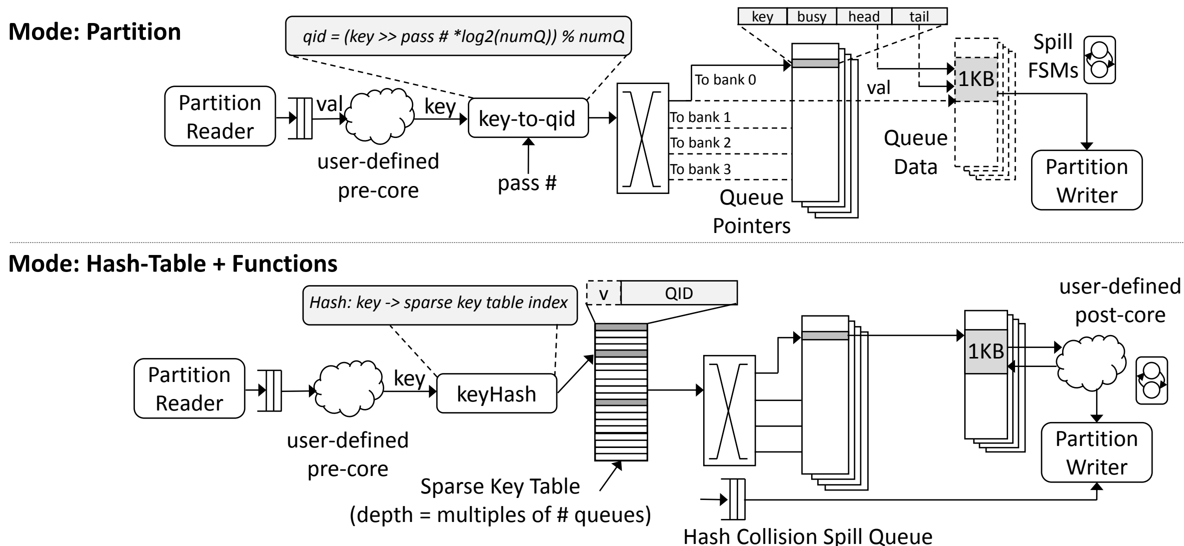

Figure 9: LINQits Hardware Template (HAAT) Configured for Partitioning (Top) and Hash-Table (Bottom). Within the HAT, there can be multiple parallel lanes that access independent banks for the queue data.

titions, which are data structures that realize LINQ collections in main memory (note: we refer to collections and partitions interchangeably). At run-time, partitions are initialized by software on the ARM cores, which share the same coherent memory subsystem with the FPGA. Figure 8 illustrates how software organizes the partition data structure. In main memory, the freelist records a list of fixed-sized data arrays on the order of 1-4KB—the size of a memory page. The partition meta block contains a list of partition headers, each with four fields: (1) a key or partition ID, (2) a next pointer for chaining multiple partitions (e.g., used for `GroupBy` that produces a collection of collections), (3) a data index that points to the beginning of a linked list of arrays, and (4) the number of elements in the collection. At run-time the partition reader processes a starting metablock (programmed by the ARM core through memory-mapped I/O registers) and begins fetching data until the partition (or a chained group of partitions) is completely read.

Pre- and post-cores. Streams of data from the partition reader are fed into the **pre-cores**, which are used to support the anonymous functions in LINQ operators. Pre-cores implement user-defined logic functions that perform value-to-key generation for `Select`, `Where`, `GroupBy`, and `Join`. The **post-cores** are the dual to the pre-cores, associated with the data item storage. These cores implement the user-defined functions for queries such as `Aggregate` and `Join`, described further below in Section 6.3. The keys from the pre-cores are used to steer key-value pairs to the downstream queues. For small HAT instances, the data steering is facilitated by a crossbar, but for large HAT instances, a scalable network is required for data item movement. In LINQits, the user-defined functions realized as pre- and post-cores must be side-effect free and cannot update global state. Section 6.4 gives more details on user-defined function constraints.

Queue Insertions. When partitioning, the pre-core outputs are passed into a special block that maps generated keys into separate queues (key-to-qid block in Figure 9). The key-to-qid block selects bits from the user-generated key to determine its destination queue. In an example HAT with 64 queues, bits 5 to 0 of the key would be used to index the queues during the first partitioning phase, followed by bits 11 to 6 for phase 2, etc. One partitioning step, for example, will sub-divide a single partition into 64 sub-partitions. This process, when repeated on the 64 sub-partitions again, will create at most 4096 sub-partitions (and so forth). After a desired number of partitioning steps are performed, a `GroupBy` (or `Join`) operation can take place on much smaller sub-partitions that can fit into on-chip memories.

Partition Allocator and Writer. Given that partition sizes and locations are not known *a priori*, the HAT must perform dynamic memory allocation of arrays and partitions (for writing) as it processes a stream of key-value pairs. The Partition Allocator and Writer (PAW) modules are specialized DMA engines that manage memory block allocation with a list of free arrays. Arrays are linked together that map to the same queue (and have the same key) by placing the address of the next array in the last entry of the queue (i.e., building up a linked list) before the queue is written to memory. The queues are sized to burst in large data chunks to maximize DRAM row-buffer locality. These "write-back" bursts are initiated in parallel with queue insertions.

6.2 Hash Table Mode for GroupBy and Join

Figure 9 (bottom) shows the activated hardware structures when operating the HAT in hash mode, needed for the final stages of `GroupBy` and `Join`. In this mode, the same queues for partitioning are repurposed into hash table entries. Like before, the partition reader streams in data that is fed into the pre-cores to generate the keys.

Sparse Key Table. During hashing, all of the HAT's queues can ideally be allocated efficiently using a fully-associative table that maps any unique key to an available queue. Unfortunately, the FPGA does not support content-associative memories efficiently. Instead, we use a direct-indexed sparse key table (shown in Figure 9, bottom) that is sized to multiples of the number of queues. When a key

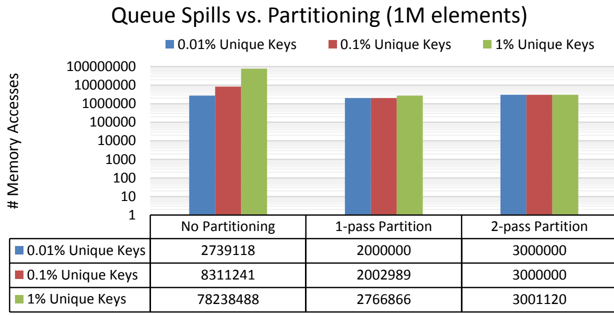

Figure 10: Number of Spills vs. Number of Partitioning Phases.

arrives, a key hash function indexes into the sparse key table to determine if there is a valid entry. If not, then a new key is allocated from a free queue counter that tracks available, unallocated queues. The counter then points to the actual row corresponding to the logical hash table entry.

Handling Hash Conflicts. Even after partitioning—for correctness reasons—the HAT must still deal with occasional hash conflicts. In a typical software-based hash table, a conflict is handled using a linked list to chain conflicted elements. In hardware, adding chaining support to each physical queue in the HAT would introduce undesired complexity and logic for a rarely occurring event. A key observation we make is that the conflicted elements should simply be *re-circulated* through the HAT rather than being inserted into the HAT's hash table. As shown in Figure 9 (bottom), the HAT reserves a special hash collision spill queue that stores *all* of the conflicted key-value pairs. The spill queue is treated like any other queue in the HAT and can spill into its own private partition in main memory. When the HAT finishes reading its current partition, it re-circulates the content of the the spill queue (and its partition) through the HAT, until all elements have been processed.

Sensitivity of Spills to Partitioning. Figure 10 shows how partitioning dramatically reduces the number of queue spills in a hardware-based `GroupBy`. For a 1M data set with 1% unique keys in the distribution, the amount of spillage under no partitioning can be 72 times the number of elements. This excessive spilling occurs because the hardware-based hash table is very small, filling up to only 64, then spilling all other remaining keys into the spill queues. On subsequent iterations, this process continues pathologically until the entire collection is processed. As can be seen, with a 2-pass partitioning step, the number of memory accesses is tripled (2 for the partitioning, 1 for the final hash table), but spills become negligible because the hash tables rarely conflict after the partitioning steps.

6.3 Supporting Other Essential Operators

We now discuss how the HAT primitives can be used to support the remainder of the essential LINQ operators. In the case of `Select`, `SelectMany`, and `Where`, the pre-cores are used to compute the user-defined functions that map inputs to outputs (for `Select`, `SelectMany`) or inputs to predicates (for `Where`). In these modes, the partition queues can be chained together to form one large logical buffer for spilling out to a single partition. In `Aggregate` mode, the HAT operates similarly to `Select`, except that values exiting the queues are accumulated into a register value rather than being spilled directly into memory. The accumulation function for `Aggregate` is supported by the post-cores shown in Figure 7.

The `Join` is the most complex operator to implement. The approach we take is to apply the `GroupBy` operator to two separate input partitions, breaking them into two sets of sub-partitions that can be joined together per-subgroup in a pair-wise fashion. In main memory, a table must be used to track which sub-partitions actually exist—we perform this operation using the partition allocator state machine. After the sub-partitions are created, the LINQits hardware is reconfigured to stream data from both partitions into the queues, upon which the post-cores are used to perform the actual join.

6.4 Generating pre- and post-cores

In our current FPGA-based implementation, the user-defined functions (i.e., pre- and post-cores) are implemented using a high-level synthesis tool called AutoESL [47]. Currently, our process for converting LINQ C# functions to AutoESL-compatible C is not automatic. Although the majority of the code translation is straightforward, we must specify pragmas that control the pipeline datapath widths and the desired initiation intervals for loops. In the near future, we plan to automatically translate C# expressions and methods into AutoESL-compatible C/C++ code; using a constrained hardware template also makes it easy for us to generate the pragmas automatically. It is worth noting that LINQits does not fundamentally require HLS—a simple thread-interleaved RISC processor is also a viable approach, which we plan to investigate in future work.

7. EVALUATION

Target Platform. Our study compares the performance, power, and energy efficiency of a LINQits FPGA prototype running against tuned software on the ZYNQ ZC702 platform [48] (described in Section 3). As depicted in Table 2, the ZYNQ processor approximates the capabilities of a hypothetical mobile system-on-chip augmented with reconfigurable logic. Although ZYNQ is not engineered for mobile devices, the combined FPGA and ARM cores in a single die enables accurate architectural comparisons and measurements in a technology-neutral environment.

Power Methodology. The ZC702 platform exposes 9 voltage rails of independent sub-systems (e.g., DRAM, ARM, FPGA) that can be measured through three on-board software-managed voltage regulators [48]. We are interested in two levels of detail on our hardware platform: the core power characteristics and the system power characteristics. For the core power, we assume that unused components can be power- and clock-gated. This means that we ignore the on-chip power measurements of the FPGA and its subcomponents when focusing on the A9 subsystem and vice-versa. Out of the 9 voltage rails, we log the ZYNQ's power including the BlockRAMs, CPU system, and DDR I/O supply voltage, and ignore peripheral power measurements in our comparisons.

7.1 Applications

Software. Table 3 lists our workloads reflecting both compute- and memory-intensive uses of LINQ. For all soft-

Technology	28nm TSMC
Processing	2-core Cortex-A9 + FPGA Logic
Core Parameters	2-wide superscalar out-of-order
Peak Frequencies	667 MHz (A9), 200MHz (FPGA)
DRAM	1GB DDR3-533MHz
	Single-channel, 8.5GB/s peak
Voltages	1V, 1.5V (DDR3)
FPGA Capacity	53KLUTs, 106K Flip-flops
CPU-to-DRAM Latency	140ns
FPGA-to-DRAM Latency	140ns
CPU-to-FPGA Latency	200ns

Table 2: ZYNQ platform overview.

	Application Parameters
K-means	Single-prec, 4M 2D Points, 10 centers
Bscholes	Single-prec, 1M Options
GroupBy	64b words, 32b keys, 400K inputs, 10% uniq. keys
Join	64b words, 32b keys, 2×200K inputs, 10% uniq. keys
KeyCount	32b words, 32b keys, 4M inputs, 10% uniq. keys

Table 3: Software Benchmarks.

ware implementations, we evaluated both C# and native C implementations to separate the overheads of a managed run-time for C#. The K-means clustering algorithm [44] and Black-Scholes (Bscholes) [43] are examples of two compute-intensive, floating-point workloads. In C#, K-means and Black-Scholes are implemented using LINQ queries, similar to the examples described in Section 2; in C, we used existing tuned multithreaded implementations for both benchmarks [44, 43]. For the C-based versions of GroupBy and Join, we explored classical cache-conscious algorithms [41] and hash partitioning [27]. In our tuning efforts for the ARM cores, an implementation of the radix-sort clustering algorithm [30] gave us the best overall performance.

All software workloads are measured using cycle-level CPU timers while running on Debian Linux and averaged across multiple runs. In all C-based codes, we pre-allocated memory pages to optimize performance—our ARM core measurements exclude this overhead. We also compiled C-based codes using GCC-4.1 with ARM VFP floating point instructions enabled. To run C# applications, we cross-compiled the Linux-compatible Mono 2.10 runtime [2] using an ARM GCC 4.1 cross-compiler. Instead of relying on Mono's built-in LINQ provider, we leveraged an improved internal implementation of LINQ [38]. Due to synchronization bugs in Mono 2.10, we were unable to run any multi-threaded LINQ queries to completion—thus, we report only single-threaded results for C#. Join, GroupBy, and KeyCount in C are also single-threaded—in these instances, we conservatively report perfect performance scaling from 1 to 2 cores (and for the power measurements, we ran two concurrent single-threaded instances with reduced dataset sizes).

Hardware. Each application in Table 3 and associated queries were mapped to different instances of the LINQits hardware template (HAT) described in Section 5. The HAT is currently implemented in under 10K lines of Verilog (including test harnesses) and has been placed-and-routed at 100MHz on the ZYNQ ZC702 platform for all the applications. The HAT exposes many compile-time parameters (e.g., queue sizes, number of queues, datapath width, etc.), each configured according to a given application's needs, as shown in Table 4. Presently, our parameters are scaled manually to achieve high area utilization of the FPGA while meeting the datapath requirements based on the data types required by the queries.

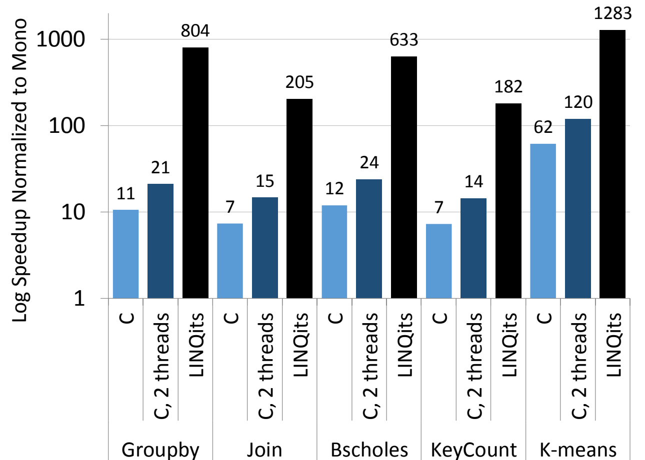

Figure 11: Speedup of optimized C (1 and 2-CPUs) and LINQits relative to C#.

K-Means, Black-Scholes, and KeyCount have user-defined anonymous functions in LINQ. These functions are mapped to the pre- and post-core regions discussed in Section 5. As discussed in Section 5, to generate these cores, we converted the original C# LINQ functions into C code targeting Xilinx AutoESL [47], a commercial high-level synthesis (HLS) tool that compiles C/C++ into RTL. These C functions must be side-effect-free and cannot perform memory allocations or recursion. Table 4 shows the resource utilization of each of the cores generated by AutoESL for various applications.

Inputs. For all applications, we deliberately selected input parameters that would stress the capabilities of the LINQits HAT module. In K-means, for example, the arithmetic intensity scales with $O(nk)$, where n and k are the total number of points and centers, respectively. By selecting only 10 centers, the amount of work available for offloading is limited per data byte, which increases the overhead of launching the HAT for each iteration of K-means. Similarly, for GroupBy, Join, and KeyCount, we created random distributions of inputs bounded with a replication factor such as that only 10% of the total numbers were unique. For example, grouping 400K integers with 40K unique keys will produce 40K groups of 10 integers each—during the final GroupBy phase, the HAT would be run many times but will incur large overheads when generating small groups[3]. Due to DRAM capacity limitations on the ZYNQ evaluation board (1GB, up to 25% reserved by the OS), we scaled all input sizes to fit within off-chip memory but ensured that all data set sizes exceeded the last-level shared cache (512KB) of the ARM A9 cores to generate realistic traffic (see Table 3).

7.2 Performance Comparison

Figure 11 shows the performance of C (1 and 2 threads) and LINQits normalized to single-threaded C# performance. Our first observation is that running C# on a managed run-time (Mono) incurs a significant overhead rel-

[3]We also conducted experiments with less adversarial data sets with expected results (not reported in this paper).

	Configuration	Operators	Total Area	Pre-Core Area	Post-Core Area
GroupBy	64b data, 32b key, 256 logical queues, 4 banks, 1kB/queue, 4 pre-cores	Partition, GroupBy	21 KLUTs (39%) 90 BRAMs (62%)	1.1 KLUTs (2%)	-
Join	64b data, 32b key, 256 logical queues, 4 banks, 1kB/queue, 4 pre-cores	Partition, GroupBy, Join	27 KLUTs (51%) 94 BRAMs (67%)	2.2 KLUTs (4%)	-
Black-Scholes	256b data-in, 32b data-out, 4 logical queues, 2 banks, 1kB/queue, 2 pre-cores	Select	29 KLUTs (54%) 54 BRAMs (34%)	3.9 KLUTs (7%) 55 DSPs (24%)	-
KeyCount	32b data, 32b key, 1024 logical queues, 8 banks, 256B/queue, 8 pre-cores, 1 post-core	Partition, GroupByAccum	38 KLUTs (71%) 98 BRAMs (65%)	635 LUTs (1%)	747 LUTs (1%)
K-means	64b data, 32b key, 64 logical queues, 4 banks, 1kB/queue, 4 pre-cores, 1 post-core	GroupByAccum	28 KLUTs (51%) 90 BRAMs (60%)	1.7 KLUTs (3%) 14 DSPs (6%) 1 BRAM (1%)	6.7KLUTs (13%) 8 DSPs (4%)

Table 4: Area of the LINQits hardware template configured to implement different applications.

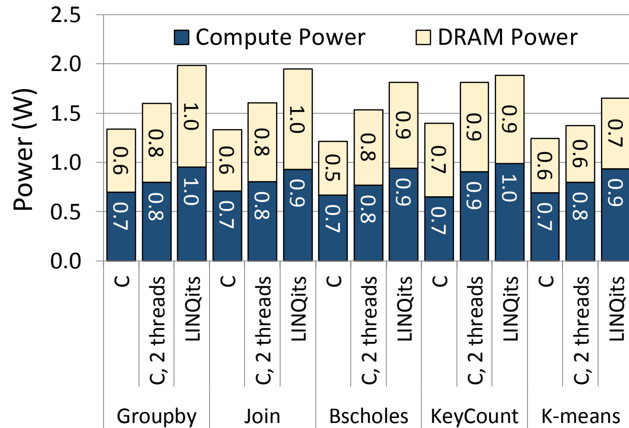

Figure 12: Power of optimized C (1 and 2-CPUs) vs. LINQits.

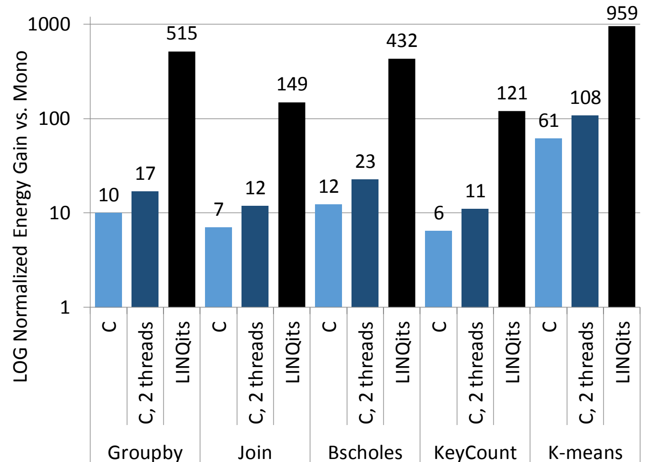

Figure 13: Energy of optimized C (1 and 2-CPUs) vs. LINQits.

ative to native C (about one order of magnitude). To verify that this was not an issue caused by the ARM-based cross-compiler, we also observed a similar degradation in performance when running Mono against C on a desktop Xeon-based processor.

The rightmost set of bars in Figure 11 show LINQits achieving nearly 2 to 3 orders of magnitude better performance relative to single-threaded C#. Relative to C with 2 threads, LINQits achieves between 10.7 and 38.1 times better performance. These gains are possible due to the HAT's ability to exploit parallelism and pipelining between multiple stages in hardware (i.e., queue insertion, queue spilling, pre- and post-core computations, etc), and the ability to coalesce memory accesses efficiently to DRAM.

Overall, for the applications we studied, LINQits achieves between 10.7 and 38.1 times better performance than optimized multithreaded C code running on the ARM cores.

7.3 Power and Energy

Figure 12 shows the system- and core-level power consumption of LINQits compared to the C-based implementations. On the 2-threaded C-based runs, the dual ARM cores (including caches) consume about 700mW of power. In the single-threaded runs, we were unable to disable the unused core, which explains the incremental difference of 100-200mW between 1- and 2-threaded runs. Figure 12 also shows the DDR3 power consumption, which alone accounts for nearly 50% of total power.

In general, the LINQits HAT running on the FPGA consumes higher power than both the ARM cores combined. LINQits, by virtue of running faster than the processors, also increases memory traffic to DRAM, resulting in higher power dissipation. A more important metric than power or speedup alone, however, is the relative energy reduction using LINQits. Figure 13 shows how LINQits achieves anywhere from 8.9 to 30.6 times energy reduction relative to multithreaded C. This result is somewhat surprising given that a significant percentage (around 50%) of the overall power is attributed to DRAM, yet LINQits is still able to improve energy efficiency by an order of magnitude. Intuitively, by speeding up the computation by an order-of-magnitude, LINQits reduces the amount of static energy wasted between idling periods of the ZYNQ's DRAM subsystem as well as maximizing efficient use of data (versus a cache controller that may unnecessarily fetch unused data). In a hypothetical mobile system with other components such as screens and radios, increasing performance would reduce the response time dramatically, lowering overall energy consumed from idling in the system.

For the applications we studied, LINQits achieves between 8.9 and 30.6 times energy reduction relative to optimized multithreaded C on the ARM cores.

7.4 Discussion

The results presented in this section confirm that the divide-and-conquer approach of LINQits combined with

Workload	Computation time (ms)				Power (mW)								Energy (mJ)							
					Computation				DRAM				Computation				DRAM			
	Mono	C	C 2threads	LINQits	Mono	C	C 2threads	LINQits	Mono	C	C 2threads	LINQits	Mono	C	C 2threads	LINQits	Mono	C	C 2threads	LINQits
Groupby	7580	716	358	9.4	707	696	800	952	559	641	796	1032	5359	498	286	9	4237	459	285	10
Join	8813	1193	596	43	699	712	803	723	566	622	772	1021	6160	849	479	31	4988	742	460	44
Blackscholes	17610	1471	735	28	698	670	768	939	548	545	554	875	12292	986	564	26	9650	802	407	25
Symbolcount	7922	1110	555	44	693	651	719	978	564	749	905	898	5490	723	399	43	4468	831	502	40
K-means	150360	2441	1255	117	687	690	800	934	547	553	571	719	103297	1684	1004	109	82247	1350	717	84

Table 5: Summary of all performance, power, and energy results.

hardware-accelerated user-defined functions can offer a significant benefit to future hypothetical mobile devices incorporating reconfigurable logic. Although we do not present ASIC results in this paper, it is a reasonable conjecture that a hardened HAT engine could significantly reduce the core power consumption relative to the FPGA by another factor of 10-100 times [28]. Without the flexibility of reconfigurable logic, the ASIC would require software-programmable engines for the pre- and post-cores, which we are planning to explore. Based solely on the results of Figure 12 and Figure 13, using an ASIC would offer a substantial gain in overall energy efficiency if: (1) the FPGA is unable to saturate off-chip memory bandwidth, and (2) off-chip DRAM power consumption scales down commensurately in the long term. We leave this investigation to future work. Table 5 offers a summary of all the performance, power, and energy results collected in our experiments.

8. FUTURE WORK

In the future, there is an entire body of work related to re-thinking the hardware-software stack for specialized hardware. For example, we plan to explore new software runtimes that optimize the scheduling and placement of LINQ queries across HATs and software. The current LINQits HAT is most effective when the startup costs are amortized across medium or large data set sizes; if amortization is impossible for a given query, then software should dynamically avoid offloading. Furthermore, there is opportunity within the runtime to pre-characterize (e.g., sampling) input collections prior to execution in hardware to determine the optimal number of levels in multi-pass partitioning for queries such as Join or GroupBy. As mentioned in Section 7, the HAT currently requires manual configuration of hardware parameters adjusted to match the application and platform. In the future, we plan to pursue methods for automatic parameter selection. Another area of future work is to augment or replace the application-specific cores within LINQits with run-time programmable, multithreaded microcode engines.

Our work also points the way to multiple improvements that can be made to an SoC that integrates FPGA fabric. One critical area of difficulty is the memory management aspects of data structures shared between the CPU and accelerator. In order to break the reliance of the FPGA on the processor for data management, adding a TLB would enable the FPGA to translate memory addresses and observe memory protection. LINQits is really a data movement engine and as such could benefit from higher memory bandwidth and larger memory capacity to dive deeper into the world of "big data". Improved external memory characteristics would cause a commensurate requirement for more internal memory capacity and bandwidth, potentially requiring a dedicated network for scalability. Finally, for finer-grained interactions between the cores and FPGA, the communication latency could be reduced, via shared message registers or other mechanisms.

9. RELATED WORK

Our work intersects broadly with three major areas: database accelerators, high level synthesis, and hardware accelerators. We provide some context by comparing our work to SQL accelerators because LINQ is a close cousin to SQL with the addition of user-defined functions. Prior work has focused on both custom and commodity hardware, like GPUs. We also perform high-level source translation to hardware for the user-defined functions and touch on high-level synthesis. Finally, researchers have been exploring ways to incorporate existing or new specialized hardware into systems at various granularities. To provide context, we discuss the mapping granularity of software to specialized hardware and focus on specialized hardware for bundles of instructions or units of coarser granularity.

9.1 Custom Database Machines

The idea of using custom hardware to accelerate queries is not new and was introduced by DeWitt in 1978 [11]. The `DIRECT` and `GAMMA` [13] multiprocessors employed specialized processors dedicated to database operations. More recently, various research and commercial efforts have used FPGAs to target acceleration of database queries in the datacenter [22, 34, 33, 37]. For example, IBM's Netezza appliance [22] uses FPGAs to filter and compress data between I/O and memory; in this setting, the FPGAs only target simple Select and Where queries while delegating more complex queries such as `GroupBy` and `Join` to the CPU. LINQits handles both simple and complex queries and incorporates support for user-defined functions in hardware.

The `Avalanche` and `Glacier` [33, 34] projects explore synthesis of database queries for FPGAs while exploiting runtime reconfigurability. The Glacier compiler focuses on acceleration of streaming algebra operators that process aggregate data in a single-pass arriving from the network or disks. Glacier does not support multi-pass partitioning to handle complex operators such as `GroupBy` or `Join`. Furthermore, Glacier requires a custom bitstream for each query, whereas the HAT in LINQits supports multiple operators in a single design that can be configured at runtime without reloading a bitstream.

9.2 GPGPUs

A growing body of work examines the use of GPGPUs to accelerate in-memory databases and query languages (e.g., [17, 19, 20, 25, 3, 46, 38]). These solutions run on commercial off-the-shelf GPUs and must amortize the communication costs over a slow PCI express channel. Some re-

ported GPU results exhibit a high degree of variance (e.g., 2-7X speedup in [20], 3-8X speedup in [25]) on complex queries such as Joins; furthermore, these studies do not compare relative energy consumption, a first-class constraint in future systems. Unlike LINQits, many of the GPU approaches focus on conventional database queries that operate on tables with simple data types such as integer tuples. LINQits, on the other hand, can be easily extended to handle complex data types such as variable-length strings or dynamic objects. An area of future work for us is to pursue a hybrid solution on a system-on-chip, where the GPU is used to process simple data-parallel queries such as `Select` and `Where`, while the FPGA and/or ASIC handles more complex queries such as `GroupBy` or `Join`.

9.3 High-Level Synthesis

Commercial high-level synthesis tools such as Xilinx AutoESL [47] compile high-level languages in C or C++ into synthesizable RTL. Although the user perceives a sequential programming abstraction, significant effort is needed to generate high-quality results. Furthermore, many tools cannot handle dynamic memory allocation or cope with large dataset sizes that exceed available on-chip storage.

In contrast, LINQits is not intended to target arbitrary programming language constructs, and instead, narrows the playing field to well-defined communication and computation patterns (i.e., LINQ queries). LINQits can leverage existing high-level synthesis tools in the generation of pre- and post-cores, although this is not fundamentally required (i.e., a multithreaded processor could also be just as effective).

9.4 Instruction-level Accelerators

Mapping basic blocks or super blocks of instructions to specialized hardware provides the finest granularity of specialization. These groups of instructions can be mapped on to specialized processor cores, as is the case with C-Cores [42]. These C-cores lack the generality to be reused for other purposes, which is appropriate for designs that have plentiful transistor and communication latency budgets. This approach is similar to fixed-function accelerators in mobile SoCs on the market now. On the other end of the spectrum are coarse-grain reconfigurable arrays (CGRAs) that compose data flow engines from smaller building blocks, like those used by DYSER [4]. The network that contains the coarse-grain units provides the flexibility to map many different program segments or data flow graphs onto the same building blocks. This latter example provides more flexibility and the underlying silicon area can have higher utilization. This approach is limited by the communication resources and related data flow scheduling problem. In either case, these approaches use the general purpose processor's memory system, including caches to service memory requests. These approaches are also restricted by being tightly coupled to imperative languages. That is—the accelerators are implementing the instructions and not providing an opportunity to optimize the algorithm or implementation.

9.5 Method-level Accelerators

Instead of targeting basic blocks or instruction level acceleration, others have taken the approach of accelerating methods in high level languages. In this scenario, the high level language provides rapid software development. Later, programmers with intimate hardware knowledge can provide method replacements that are specific to the underlying hardware, in most case GPUs or many-core processors. This provides an opportunity to optimize the algorithm and target specific hardware, as has been demonstrated by SEJITS, Copperhead and auto-tuning [26, 7, 45]. Thus far, researchers have targeted existing hardware and have not applied this methodology to specialized hardware. This also has the advantage of overloading methods in popular programming languages and is very suitable for accelerating library code. In high performance computing, the math kernel library (MKL) and CUDA are examples of this [23, 36]. Finally, given an API, the library developer is free to implement the algorithm however they prefer. The method declares the computation's intent instead of explicitly describing how it should be done.

9.6 Language-level Accelerators

The highest level abstraction for specialized hardware has been developed in the context of Domain Specific Languages (DSLs) that can target many-core CPU or GPU systems. In this case, frameworks like Delite [5], provide operators for domain specific experts to build their own DSL. Predefined Delite operators used in the new DSL definition are accelerated for "free" because the operators already target existing hardware. Others can then use these DSLs for their applications and leverage this. If pre-defined operators are not used in the DSL, someone will have to map these new operators on to the specialized hardware. In general, new languages, including DSLs, suffer from an adoption problem, limiting their utility.

LINQ is an example of a DSL that associates SQL-like operations with a data collection. Unlike new DSLs, LINQ is an established DSL that can be embedded in many managed-language frameworks, like Java and C#. In this work, we target the LINQ operators for specialized hardware as opposed to trying to write optimized code for the LINQ operator that targets many-core or GPUs. LINQ provides a declarative description of the computation and allows optimization at many levels, e.g., the query plan down to individual operators.

9.7 Hardware-Based Templates

Finally, the idea of hardware-based templates has been explored for both many-core processing and map-reduce patterns. The MARC project, for example, targets OpenCL and similar abstractions to a many-core template that can be re-configured for an FPGA or implemented on an ASIC [29]. Researchers have also proposed implementing a Map-Reduce computational pattern in the FPGA [40].

10. CONCLUSION

The LINQits framework represents a first step towards domain-specific declarative language acceleration using specialized logic. The heart of LINQits is a parameterized hardware template that employs divide-and-conquer approaches to virtualize limited on-chip resources while coalescing memory accesses for better locality. Our empirical results from running on a real single-chip heterogeneous platform confirms that custom hardware can significantly enhance data- and compute-intensive tasks on energy-constrained mobile processors. Compared to optimized, multithreaded C code, LINQits is 10.7 to 38.1 times faster and 8.9 to 30.6 times more energy-efficient than low-power, embedded cores.

Acknowledgements

We thank Chris Rossbach, Yuan Yu, and Jon Currey for their help with the Dandelion compiler. We thank our anonymous reviewers, Doug Burger, Babak Falsafi, and Onur Koçberber for their invaluable feedback.

11. REFERENCES

[1] "Big Data Definition," mike2.openmethodology.org/wiki/Big_Data_Definition.
[2] "Mono Platform," www.mono-project.com.
[3] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *GPGPU'10*.
[4] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, Integration and Implementation of the DySER Hardware Accelerator into OpenSPARC," in *HPCA'12*.
[5] K. Brown, A. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A Heterogeneous Parallel Framework for Domain-Specific Languages," in *PACT'11*.
[6] M. Budiu, J. Shotton, D. G. Murray, and M. Finocchio, "Parallelizing the Training of the Kinect Body Parts Labeling Algorithm," in *Big Learning: Algorithms, Systems and Tools for Learning at Scale*, Sierra Nevada, Spain, December 16-17 2011.
[7] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an Embedded Data Parallel Language," in *PPoPP'11*.
[8] Chipworks, Inc. Inside the Apple iPad 4–A6X a very new beast! www.chipworks.com/blog/recentteardowns/2012/11/01/inside-the-apple-ipad-4-a6x-to-be-revealed/.
[9] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" in *MICRO'10*.
[10] R. H. Dennard, F. H. Gaensslen, H.-n. Yu, V. Leo Rideovt, E. Bassous, and A. R. Leblanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *Solid-State Circuits Newsletter, IEEE*, vol. 12, no. 1, pp. 38 –50, winter 2007.
[11] D. J. DeWitt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," in *ISCA'78*.
[12] D. J. DeWitt and R. H. Gerber, "Multiprocessor Hash-Based Join Algorithms," in *VLDB'85*.
[13] D. J. Dewitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "Gamma - A High Performance Dataflow Database Machine," in *VLDB'86*.
[14] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *ISCA'11*.
[15] Gartner, "The Mobile Scenario: Understanding Mobile Trends Through 2017," gartner.com/it/page.jsp?id=2227215, Nov 2012.
[16] J. R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Management," May 1981.
[17] N. K. Govindaraju and D. Manocha, "Efficient Relational Database Management Using Graphics Processors," in *DaMoN'05*.
[18] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward Dark Silicon in Servers," *IEEE Micro*, vol. 31, no. 4, pp. 6–15, Jul. 2011.
[19] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational Query Coprocessing on Graphics Processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, Dec. 2009.
[20] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," in *SIGMOD'08*.
[21] Herb Sutter, "Elements of Modern C++ Style," herbsutter.com/elements-of-modern-c-style, Oct 2010.
[22] IBM, Inc. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics.
[23] Intel, Inc. Intel Math Kernel Library. http://www.intel.com/software/products/mkl.
[24] M. Isard *et al.*, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proc. EuroSys*, 2007.
[25] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU Join Processing Revisited," in *DaMoN'12*.
[26] S. Kamil, D. Coetzee, S. Beamer, H. Cook, E. Gonina, J. Harper, J. Morlan, and A. Fox, "Portable Parallel Performance from Sequential, Productive, Embedded Domain-Specific Languages," *SIGPLAN Not.*, vol. 47, no. 8, pp. 303–304, Feb. 2012.
[27] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs," *Proc. VLDB Endow.*
[28] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," in *FPGA'06*.
[29] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring Many-core Design Templates for FPGAs and ASICs," *Int. J. Reconfig. Comput.*, vol. 2012, pp. 8:8–8:8, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/439141
[30] S. Manegold, P. Boncz, and M. Kersten, "Optimizing Main-Memory Join on Modern Hardware," *IEEE Trans. on Knowl. and Data Eng.*, vol. 14, no. 4, pp. 709–730, Jul. 2002.
[31] F. McSherry, Y. Yu, M. Budiu, M. Isard, and D. Fetterly, *Scaling Up Machine Learning*. Cambridge University Press, 2011, ch. Large-Scale Machine Learning using DryadLINQ.
[32] Microsoft, Inc., "LINQ (Language-Integrated Query)," msdn.microsoft.com/en-us/library/bb397926.aspx.
[33] R. Mueller, J. Teubner, and G. Alonso, "Glacier: A Query-to-Hardware Compiler," in *SIGMOD'10*.
[34] ——, "Streams on Wires: A Query Compiler for FPGAs," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 229–240, Aug. 2009.
[35] NVIDIA, Inc. www.nvidia.com.
[36] ——. www.nvidia.com/object/cuda_home_new.html.
[37] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "An Implementation of Handshake Join on FPGA," in *ICNC'11*.
[38] C. J. Rossbach, Y. Yu, J. Currey, and J.-P. Martin, "Dandelion: A Compiler and Runtime for Distibuted Heterogeneous Systems," *Technical Report: MSR-TR-2013-44, Microsoft Research Silicon Valley*, 2013.
[39] Samsung, Inc. www.samsung.com/global/business/semiconductor/minisite/Exynos/index.html.
[40] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA," in *FPGA'10*.
[41] A. Shatdal, C. Kant, and J. F. Naughton, "Cache Conscious Algorithms for Relational Query Processing," in *VLDB'94*.
[42] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *ASPLOS'10*.
[43] Victor Podlozhnyuk, NVIDIA Inc. (2007) Black-Scholes Option Pricing.
[44] Wei-keng Liao. [Online]. Available: users.eecs.northwestern.edu/~wkliao/Kmeans
[45] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," in *SC'07*.
[46] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation," in *MICRO'12*.
[47] Xilinx, Inc. Vivado High-Level Synthesis. www.xilinx.com/tools/autoesl.htm.
[48] ——, "ZC702 Evaluation Board for the Zynq-7000 XC7Z020. All Programmable SoC User Guide, October 8, 2012."
[49] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey, F. McSherry, and K. Achan, "Some Sample Programs Written in DryadLINQ," Microsoft Research, Tech. Rep. MSR-TR-2009-182, December 2009.