A Study of Pointer-Chasing Performance on Shared-Memory Processor-FPGA Systems

Gabriel Weisz[1,2] Joseph Melber[1] Yu Wang[1]
Kermin Fleming[3] Eriko Nurvitadhi[3] and James C. Hoe[1]
[1] Computer Architecture Lab at Carnegie Mellon University, {gweisz,jmelber,yuw,jhoe}@cmu.edu
[2] Information Sciences Institute, University of Southern California, gweisz@isi.edu
[3] Intel Corporation, {kermin.fleming,eriko.nurvitadhi}@intel.com

ABSTRACT

The advent of FPGA acceleration platforms with direct coherent access to processor memory creates an opportunity for accelerating applications with irregular parallelism governed by large in-memory pointer-based data structures. This paper uses the simple reference behavior of a linked-list traversal as a proxy to study the performance potentials of accelerating these applications on shared-memory processor-FPGA systems. The linked-list traversal is parameterized by node layout in memory, per-node data payload size, payload dependence, and traversal concurrency to capture the main performance effects of different pointer-based data structures and algorithms. The paper explores the trade-offs over a wide range of implementation options available on shared-memory processor-FPGA architectures, including using tightly-coupled processor assistance. We make observations of the key effects on currently available systems including the Xilinx Zynq, the Intel QuickAssist QPI FPGA Platform, and the Convey HC-2. The key results show: (1) the FPGA fabric is least efficient when traversing a single list with non-sequential node layout and a small payload size; (2) processor assistance can help alleviate this shortcoming; and (3) when appropriate, a fabric-only approach that interleaves multiple linked list traversals is an effective way to maximize traversal performance.

1. INTRODUCTION

Motivations. There are now a growing number of FPGA-accelerated computing systems that support coherent shared memory between processors and FPGAs, including the ability for the FPGAs to directly read from and write to the processor's cache. Both Xilinx and Altera have this capability in their SoC products, which integrate processor cores and a reconfigurable fabric on the same die [1, 18]. The Convey HC-1 was an early commercial FPGA acceleration system that supported shared memory with the host processor [4]. More recently, Intel and IBM have respectively announced initial server products that integrate cache-coherent shared-memory processors and FPGAs at the system level [13, 2]. These new platforms promise to enable FPGA acceleration for a new class of applications with irregular parallelism.

Irregular parallel applications, including many data analytic and machine learning kernels in data center workloads, operate on very large, memory-resident, pointer-based data structures (i.e., lists, trees and graphs). For example, databases use tree-like structures to store the indices for fast searches and combining information from different database tables. Similarly, many machine learning algorithms within big data applications rely on graphs, which use pointers to represent the relationships between data items.

The parallelism and memory access patterns of these applications are dictated by the point-to relationships in the data structure, which can be irregular, and sometimes time-varying. This reliance on pointer-chasing imposes stringent requirements on memory latency (in addition to bandwidth) over a large main-memory footprint. As such, these applications are poorly matched for traditional add-on FPGA accelerator cards attached to the I/O bus, which can only operate on a limited window of locally buffered data at a time.

Pointer-Chasing. While the class of irregular parallel applications is broad and varied, a fundamental behavior is pointer-chasing. In pointer chasing, the computation is required to dereference a pointer to retrieve each node from memory, which contains both a data payload to be processed and a pointer (or pointers) to subsequent nodes. The exact computation on the payload and the determination of the next pointer to follow depend on the specific data structure and algorithm in use. In this paper, we ignore these differences and focus on only the basic effects of memory access latency and bandwidth on pointer chasing. It is our contention that the optimization of basic pointer-chasing performance ultimately determines the opportunities for FPGA acceleration of irregular parallel applications.

For this purpose, we fixed a simple reference behavior, namely a linked-list traversal. This reference behavior is parameterized by (1) node layout in memory (best- vs. worst-case in our experiments); (2) per node data payload size; (3) payload dependence (an artificial constraint that payload must be retrieved before following the next pointer); and (4) concurrent traversals (availability of multiple independent traversals). Taken together, these parameters abstractly capture the execution differences of different pointer-based algorithms and data-structures. The two execution requirements are that (1) the linked-list is initialized in DRAM and (2) the data payload must be delivered into the fabric

FPGA'16, February 21-23, 2016, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/2847263.2847269

in order.[1] All other details are open to interpretation and optimization by the implementation.

Implementation Considerations. The shared-memory FPGA computing systems under consideration may have multiple pathways between the fabric and main memory, and between the fabric and the processor. Because of this, the best approach for implementing even simple behaviors, such as linked-list traversals, is not obvious, even before taking into account the data-structure-specific parameters discussed above. In Section 4, we will discuss the range of implementation options and their considerations in the context of an abstract shared-memory processor-FPGA architecture. In Section 5, we will demonstrate concretely the manifestations of these effects on real shared-memory processor-FPGA systems available today including the Xilinx Zynq, the Intel QuickAssist QPI FPGA Platform, and the Convey HC-2. Our key results demonstrate that:

1. A fabric-only approach is least efficient when traversing a single list with non-sequential placement and small payload size.

2. Processor assistance can help alleviate this shortcoming.

3. When appropriate, interleaving multiple list traversals can be very effective in optimizing traversal performance by allowing pipelined memory operations.

A particular point of interest is our incorporation of a tightly-coupled processor in the solution space. Implementing the FPGA's memory subsystem in soft logic can dramatically lower its performance (due to lower frequency and a reduced cache capacity) relative to what can be achieved in hard logic. The processor-FPGA link technology in use and the placement of the FPGA in the system can also have a major impact on its memory access latency. The effects of these design choices are most prominent when using an in-fabric engine to traverse a linked list with non-sequential placement and small payload size. In this paper, we study a range of hybrid approaches that use processors (with an order of magnitude higher clock frequency and cache capacity) to assist in delivering the data payload to the fabric. Processor assistance also has the added benefit of greatly simplifying the implementation of complicated pointer-based data structures and algorithms found in realistic applications by expressing these behaviors in software. Our study points to this hybrid approach as a promising avenue for supporting irregular parallelism on future shared-memory processor-FPGA systems.

Paper Outline. Following this introduction, Section 2 reviews background material on shared-memory processor-FPGA systems and irregular parallel applications. Section 3 defines the reference linked-list traversal behavior. Section 4 discusses general design considerations. Section 5 demonstrates the key effects observed on select shared-memory processor-FPGA systems currently available. Section 6 offers recommendations on the design of future systems with the goal of accelerating irregular parallel applications. Finally, Section 7 concludes.

[1]For the purposes of this study, we are not concerned with justifying why the data need to be processed in the fabric. Other works have shown FPGAs offer raw performance and power efficiency that make them attractive for these types of applications [3, 10, 7, 16, 14, 9].

2. BACKGROUND

Shared-Memory Processor-FPGA Systems. As early as 2010, FPGA vendors have created "System-on-Chip" devices that integrate hard-logic processor cores, a reconfigurable fabric, and shared memory into the same chip [1, 18]. While this approach allows tight integration between the processor cores and reconfigurable fabric, the existing product lines have targeted the embedded market in terms of the included cores, fabric capacity, and DRAM interfaces. The Convey HC-1 was an early server-class FPGA acceleration system that supported shared memory with the host processor [4]. Intel and IBM have recently announced server-class products that integrate FPGAs and processors at the board level using proprietary cache-coherent interconnects [13, 2]. These later systems support best-of-breed processors and FPGAs, but incur latency and bandwidth overheads by using inter-package board-level links and relying on soft logic to implement the memory interfaces on the FPGAs. Because of these overheads, the design choices and performance level of today's commercially available products should not be taken as representative of what future shared-memory processor-FPGA architectures could or should be. Optimistically, these products are indicative of a growing interest in FPGAs as first-class computing substrates, which will hopefully lead to still significant evolution in step with the emergence of "killer" apps.

Irregular Parallel Applications. A major premise of this paper is that shared-memory processor-FPGA systems can be effective for accelerating irregular parallel application kernels common to data center workloads. Machine learning algorithms are at the heart of search and many image- and speech-recognition tasks that make up big data applications. Graphs are the core data structure used by these algorithms [6, 8], and use pointers to represent relationships between information. Databases store information in a number of tables. They use indexing structures to allow fast retrieval of information across these tables. These operations commonly require chasing pointers within a tree-like structure (e.g., for efficient range-based searches [5]).

A common theme in the execution of irregular parallel applications is pointer-chasing over a large in-memory data structure [15]. Traditional add-on FPGA accelerator cards are attached to the I/O bus, and can only efficiently access small amounts of data that has been bulk-copied to the DRAM installed on the card. These cards are thus limited to working on a limited window of locally buffered data. Prior work that accelerated irregular parallel application kernels on add-on FPGA accelerator cards first "regularized" the task into contiguous chunks of data, which were then handed off to the FPGA accelerator one by one for processing. This was achieved either by exploiting next-level batch-processing parallelism between tasks (e.g., [16, 14]) or by pre-processing to create parallelizable task partitions through scheduling and data reorganization (e.g., [12]).

On a shared-memory processor-FPGA system, the fabric can directly access the full data set, in some cases with the benefit of virtual address translation. In prior work, Umuroglu, et al., studied a breadth-first graph traversal using a Xilinx Zynq SoC FPGA that shares memory between

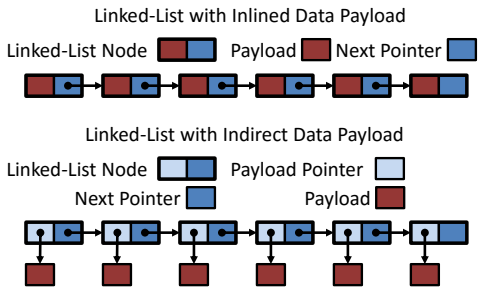

Figure 1: Two possible implementations of the linked-list struct with an inlined data payload and an indirect data payload.

the ARM cores and fabric [17]. Their work assigned different phases of the breadth-first traversal processing to the ARM core and the fabric respectively. Hurkat, et al. studied FPGA acceleration of a machine learning algorithm that took advantage of Convey's special high-throughput interface to a large pool of memory [9].

3. LINKED-LIST TRAVERSAL

Regardless of the exact data structure and algorithm, irregular parallel algorithms repeatedly follow a pointer to the next node in order to find the associated data payload, and then perform the associated computation. In this paper, we ignore the computation, and instead focus on the pointer-chasing backbone of a dynamic execution instance and the fetching of the data payload along the way. This is what we attempt to capture in the simple reference behavior of a linked-list traversal. Establishing a simple reference behavior enables an exploration of the many possible optimizations of pointer-chasing mechansims on real systems. Despite its simplicity, the parameterizations defined by this reference behavior allow us to capture the most salient execution characteristics of many different pointer-based data structures and algorithms.

Linked Lists. A generic singly linked list is a sequence of nodes. Logically, each node contains a "next" pointer and a data payload; the next pointer points to the next node in the sequence. In this study, a linked-list traversal is parameterized in four dimensions: (1) node layout in memory; (2) per node data payload size; (3) payload dependence; and (4) concurrent traversals.

- **Layout:** In practice, the placement of the nodes in memory depends on the data structure, the algorithm, and the memory allocator. In our study, we consider only the best case and worst case data layouts. For the best case data layouts, the linked-list nodes are laid out sequentially in memory to make optimal use of the spatial locality optimizations in standard memory subsystems. In the best-case studies, sequential prefetching is allowed as an optimization. For the worst case data layouts, the linked-list nodes are laid out in large strides (16 KBytes) to defeat cache block and DRAM row buffer reuse. In the worst-case studies, we explicitly disallow prefetching based on this known stride as an optimization. Real world linked lists would fall somewhere between these two extremes.

- **Payload Size:** We considered data payload size as a parameter (varying from 4 to 1K bytes in our studies). The payload size should correspond to how much of the payload needs to be examined in the traversal of a real-world pointer-based data structure or algorithm (and not their full declared payload size).

- **Payload Dependence:** This third parameter is an artificial constraint that the payload must be retrieved before the next pointer in the linked list can be followed. This is to model the effect of payload data dependence when traversing a real-world pointer-based data structure. For example, the behavior of traversing a sorted binary tree to produce a sorted sequence would be captured by a linked-list traversal without payload dependence; the behavior of searching a sorted binary tree (requiring examining the value at a node before descending to the next node) would have to be captured by a linked-list traversal with payload dependence.

- **Concurrent Traversals:** The final parameter allows for the possibility of benefiting from concurrency when performing multiple independent traversals. For example, two branches of a sorted binary tree could be traversed concurrently to have their sorted sequences concatenated afterwards. This degree of freedom is extremely helpful in overcoming the effects of high memory access latency.

Design Freedom. We impose no other restrictions on the application developer in order to maximize flexibility in optimizing the reference behavior for the platform. For example, we do not stipulate the pointer size, which presumably should be chosen to be natural to the platform. Furthermore, we do not stipulate that the node struct contains the actual payload field—instead each node struct may contain a pointer to a payload held separately from the node (see inlined vs. indirect payload in Figure 1). For indirect payloads, the nodes and the data payloads are separately laid out sequentially or strided, corresponding to best and worst-case data layout. As will be discussed in the next sections, the implementation for a given platform has full freedom to make use of the available mechanisms to optimize the linked-list traversal performance under different parameterization settings.

Performance Metric. For our study, we require that a sufficiently large linked list (16K nodes in our experiments) is initialized in main memory. We require the data payload to be delivered into the FPGA fabric in order. We are interested in the steady-state rate of traversing this linked list by pointer chasing. In the design of this study, we do not focus on the start-up cost, which could be significant for pointer-based data structures and algorithms that involve only short pointer-chasing sequences. In cases where short sequences need to be serialized, their behavior over repeated traversals effectively matches that of traversing long sequences. In cases where short sequences are independent and can be traversed concurrently, efficient implementations are addressed by the hardware interleaved concurrent traversals that we present later in the paper.

Linked Lists as a Proxy for More Complex Data Structures. Although singly-linked lists are simpler than other pointer-based data structures such as trees and graphs, the linked-list traversals demonstated in this paper can be used as a proxy for algorithms that traverse trees and graphs. For example, searching for an item in a sorted tree reduces

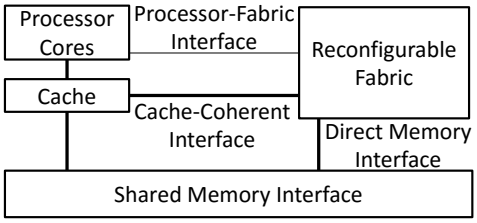

Figure 2: Model of the datapath in a shared-memory processor-FPGA system

to a payload-dependent traversal, as discussed above. Similarly, a graph traversal can be reduced to following the first edge of the current vertex that connects to an unvisited vertex, and intiating a new (possibly parallel) traversal for each additional edge of the current vertex that connects to an unvisited vertex.

4. SYSTEM-LEVEL CONSIDERATIONS

In this section, we discuss at a high-level the range of options and their implications when supporting pointer-chasing on shared-memory processor-FPGA systems. The discussion in this section is not tied to the reference linked-list traversal, but is generalized to a broad range of pointer-based data structures and algorithms. The next section will offer observations from concrete implementations of the parameterized reference linked-list traversal on real platforms.

4.1 Platform Options

Figure 2 shows a generic shared-memory processor-FPGA system architecture, highlighting the major interface options connecting the processor cores, the memory, and the reconfigurable fabric. This figure does not differentiate between die-level and system-level integration of the fabric and processor cores.

The processor cores will typically be connected to the shared main memory through a coherent cache hierarchy.[2] The processor cores can directly interact as masters with the soft-logic blocks on the fabric through a memory-mapped Programmed I/O (PIO) style interface. Not shown in the figures is the option for a programmable DMA copy engine (master) to push or pull data from the fabric (slave) through this interface.

The main feature of a shared-memory processor-FPGA system is of course the ability for the fabric itself to act as the master in reading and writing the shared memory, with some systems even supporting virtual address translation. Through shared memory, it becomes possible for the processor and fabric to interact through unbounded, diverse means. The processor cores and fabric may not have symmetric access to the shared memory, resulting in possibly large differences in their experienced bandwidth and latency. The fabric's access to shared memory may be cache-coherent or non-cache-coherent:

1. The fabric's memory interface feature a cache-coherent link, allowing the processor cores and fabric to automatically see a coherent view of memory. Furthermore, in some systems, cache-coherent accesses from the fabric may be serviced from the processor cache instead of main memory. Data requests that hit in the processor cache will incur lower latency and achieve higher bandwidth. A cache-coherent shared memory allows the processor cores and fabric to interfere constructively when accessing shared memory. This feature is available on the Xilinx Zynq and the Intel QuickAssist QPI FPGA Platform. On some systems, it is further possible for the fabric to construct its own private coherent cache in soft-logic. This is the case for the Intel QuickAssist QPI FPGA Platform but not for the Xilinx Zynq.

2. The fabric may also have a non-cache-coherent interface to the shared memory in addition to the coherent interface. A possible motivation for offering a non-coherent interface, beside implementation simplicity, is higher bandwidth and lower latency for accessing data residing in main memory. A major downside, especially if the processor cores and the fabric are interacting in a fine-grained fashion, is the cost for the processor core to explicitly ensure coherence through costly cache flushes or through precise discipline in issuing memory address sequences.

In a given system, some variations of all or a subset of the interface options in Figure 2 may be found. The available interface options will offer different tradeoffs in latency, bandwidth, and invocation overhead. Some interface options might be replicated for higher aggregate bandwidth or to allow multiple outstanding transactions. All in all, there are significant complexities in considering all of the interface options and their exact designs to optimize something as simple as linked-list traversal on a real system.

4.2 Performance Model

We offer a simple performance model to assist in reasoning about the pointer-chasing behavior that we are trying to optimize. In general, for a pointer-chasing traversal that touches n nodes of a pointer-based data structure and $s_{payload}$ bytes of data payload per node, ignoring the computation time, the average traversal time per node can be stated as:

$$\begin{aligned} T_{per-node} = T_{management}/n + (L_{node} + \\ BW_{node}^{-1} \times s_{node} + \\ L_{payload} + \\ BW_{payload}^{-1} \times s_{payload}) \end{aligned} \quad (1)$$

In the equation, $T_{management}$ includes all of the time involved in setting up and tearing down a traversal. This time is of lesser concern as it is amortized by n which we assume to be large in the reference behavior (Section 3). L_{node} and $L_{payload}$ represent the latencies in passing data across the interface providing nodes and payload data, respectively. BW_{node}^{-1} and $BW_{payload}^{-1}$ represent the corresponding incremental per-byte times for passing data. Equation 1 separates the time to read the node struct (for the pointer mainly) from the time to read the data payload using different L and BW^{-1} values. This allows for cases where an optimized implementation takes different paths for fetching the node struct vs. the payload. For the case of small inlined data payload fetched together with the pointer, $L_{payload}$ and $BW_{payload}^{-1}$ become 0.

[2]At this level of discussion, the precise organization of L1 vs. L2 and private vs. shared processor core caches is not important. We also omit details such as the possibility of mapping SRAM scratchpads or other memory-mapped devices into the shared address space.

Equation 1 assumes that the path traversed in a data structure depends on the data payload values (e.g., searching a sorted binary tree). In the cases where the traversed path is independent of the data payload values (e.g., depth-first-visit of a binary tree), it becomes possible to decouple the pointer-chasing sequence from the data payload fetch sequence (e.g., where one agent is performing pointer chasing to generate a stream of data payload pointers to be fetched by a second agent). Taking advantage of this decoupling by overlapping the two fetch sequences results in a traversal time is determined by the slower of the two sequences. The average traversal time per node when there is no payload dependence can be approximated as:

$$T_{per-node} = T_{management}/n + MAX\Big((L_{node} + BW_{node}^{-1} \times s_{node}), (L_{payload} + BW_{payload}^{-1} \times s_{payload})\Big) \quad (2)$$

The appropriateness of Equation 1 vs. 2 is captured by the payload dependence parameter in the reference linked-list traversal behavior.

Lastly, if the data structure or algorithm permits interleaved traversals of multiple data structures or independent parts of the same data structure, the effective latencies are reduced through latency hiding effects. The degree of the performance improvement will depend on the degree of interleaving and their indirect effects (e.g., increased row buffer conflicts or cache thrashing).

As a final note, the latency and bandwidth in the above first-order models are to be taken as averages. The instantaneous latency and bandwidth of an interface will in general not be constant and will be context specific. For example, both DRAM and caches will exhibit much shorter effective latency during sequential memory references than random or strided references. This effect is exercised by the best-case vs. worst-case node layout parameter of the reference linked-list traversal behavior.

4.3 Implementation Approaches

Software-Only Traversal. Although this work is predicated on delivering the linked-list payload to the fabric, we begin by discussing pointer-chasing in software using processor cores. The most obvious benefit of pointer-chasing from the processor cores is the ease of implementation afforded by software. This is particularly important for complicated algorithms and data structures. Furthermore, hard-logic processor cores benefit from a much higher clock frequency and a more sophisticated and higher capacity memory hierarchy. For example, in the current Intel QuickAssist QPI FPGA Platform, the Xeon processor enjoys a much more powerful and higher-performing memory subsystem than what is achievable from the FPGA. The Xeon processor presents a hard-to-beat design point if we focus only on the pointer chasing aspect of the traversal (ignoring the FPGA's potential advantages in payload memory accesses and payload processing). On the other hand, the disadvantage of a software-only traversal is its inability to customize at the datapath level for performance or efficiency.

Fabric-Only Traversal. The starting points in this study are hardware traversal engines built as soft-logic in the fabric. The optimal design of such traversal engines would include the logic to traverse the pointer-based data structure, and would also include an accompanying co-designed memory interface and subsystem, subjected to an extreme degree of customization. We summarize below the most profitable opportunities we encountered in our study:

1. For data structures with an indirect payload, a hardware traversal engine may separately issue the pointer and the payload memory fetches to the coherent and non-coherent interfaces, respectively, if both are available. This approach reserves the cache capacity for any available (cache-block granularity) spatial and temporal localities in the traversal of the nodes. The non-cache-coherent interface keeps the payload fetches (without temporal locality) from polluting the cache and may even be able to offer higher payload fetch bandwidth.

2. If it is known that the pointer and payload memory fetches exhibit good spatial locality (such as modeled by the best-case sequential layout in the reference behavior), a hardware traversal engine could sequentially prefetch ahead of the current node's memory location in case the following locations are needed soon. If the memory interface operates at a cache block granularity, some degree of spatial prefetching happens unavoidably; the hardware traversal engine need only add logic to recognize when subsequent nodes to be visited fall within the current cache block. In general, a hardware engine should take advantage of specific knowledge of the pointer-based data-structure and the traversal algorithm in speculative run-ahead-type optimizations.

3. When the path traversed through the pointer-based data structure is independent of the data payload values, a hardware traversal engine should decouple and overlap the pointer chasing and the data payload memory access sequences. This concurrency leads to improved behavior modeled by Equation 2 rather than Equation 1.

4. When multiple concurrent traversals are allowed, a hardware traversal engine should be much more able than software to achieve the high degree of interleaving necessary to fully hide the memory latency and saturate the memory bandwidth. When allowed, interleaving multiple traversals is a very powerful optimization technique for overcoming the effects of memory latency, and works well in general over the remaining parameters (layout, payload size, and dependence).

Hybrid Traversal. The most challenging scenario for a fabric-only traversal is traversing a single data structure with a small data payload size and prefetch-unfriendly node layout in memory. This challenge singularly accentuates the impact of memory read latency. Our pointer chasing study on shared-memory FPGAs was in fact motivated by this scenario and by the prospect of incorporating the processor cores in a hybrid solution. A basic approach would be for the processor core to traverse the pointer-based data structure, fetch and prepare the data payload to be streamed into the fabric using any one of the interface options. Though simple, this approach (which we call hybrid-push) is very

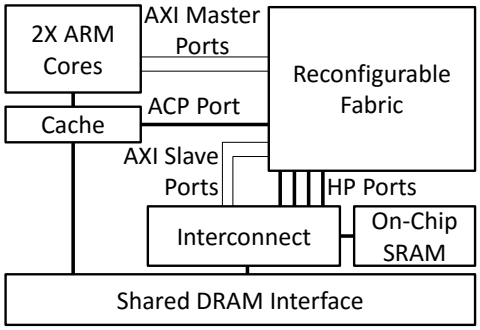

Figure 3: Datapaths between memory, the ARM cores, and the fabric in the Zynq

effective for traversals with small payload sizes. But the inefficiency of multiple movements of the data payload grows with payload size.

When the data structure can be traversed independently of the data payload values (or can be traversed using only a very small portion of the payload and minimal processing), a better approach is for the processor core to traverse the pointer-based data structure and stream pointers to the data payload into the fabric. The fabric in turn fetches the data payload directly from shared memory. This approach (we call hybrid-pull) not only benefits from more efficient use of memory bandwidth but it also benefits from overlapping the pointer-chasing and the data payload memory accesses (Equation 2). An important benefit of both hybrid approaches is the ease of development coming from handling complicated data structures and algorithms in software.

5. REAL SYSTEM EFFECTS

In this section we offer a detailed look at how the high-level considerations discussed in the preceding section play out in real systems that include the Xilinx Zynq, the Intel QuickAssist QPI FPGA Platform, and the Convey HC-2EX. By focusing exclusively on the reference linked-list traversal behavior, we are able to make a very thorough examination of the implementation space. It is important to note that we are not evaluating the suitability of these current systems for supporting pointer chasing; we certainly make no attempt to compare them. Our goal with these studies is to convey to the readers the real effects and complexity that comes from the different mechanisms and implementation options. We devote most of this section to our extensive design study on Xilinx Zynq because it provides the most diverse range of implementation options.

5.1 Xilinx Zynq

5.1.1 Platform Description

For the Xilinx Zynq study, we worked with the ZC706 evaluation board (containing a XC7Z045 SoC FPGA). Figure 3 provides a high-level view of the Zynq datapath, showing 2 ARM cores and a reconfigurable fabric. The Zynq architecture provides the full selection of interface options discussed in Section 4.1.

The reconfigurable fabric on the Zynq supports high bandwidth DRAM accesses through (a) four 64-bit non-cache-coherent "High Performance" (**HP**) ports, and (b) one 64-bit cache coherent "Accelerator Coherency Port" (**ACP**) port. The Zynq fabric can also access on-chip SRAM (referred to as **OCM**) that is shared with the ARM cores through these ports. The ARM cores interact with the fabric using programmed I/O (**PIO**) through two 32-bit memory-mapped AXI master ports. Included in the ARM system is a built-in 8-channel **DMA** engine that can copy data between any source and destination regions in the global addresss space.

Table 1: Experimental parameters.

Reference Behavior Parameters	**Values**
Node Layout	best case (sequential), worst case (16-KByte strided)
Payload Size	2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024 bytes
Payload Dependence	Yes, No
Traversal Concurrency	1, 2, 4, 8, 16, 32, 64, 128 ways

Implementation Options	**Values**
Payload Location	inlined, indirect
Traversal Approach	software-only, fabric-only, hybrid-push, hybrid-pull
Fabric Memory Path	HP, ACP
Fabric Memory Fetch Size	8, 16, 32 Bytes
Core-to-Fabric Path	PIO, DMA
Core-to-Fabric DMA Staging	in DRAM, in OCM

5.1.2 Implementations

We conducted a nearly exhaustive design study of the parameterized reference linked-list traversal behavior over the full combination of implementation options available on the Zynq. Table 1 summarizes the behavior parameters (presented in Section 3) and the Zynq implementation options discussed below.

The in-fabric portion of the traversal engines are developed in Bluespec System Verilog [11]. We tested shared memory accesses from the fabric using both the coherent ACP port and the non-coherent HP ports. The traversal engine supports a compile-time configurable data block size (tested at at 8, 16 and 32 bytes) for fetching linked-list node structs. This is to enable sequential prefetching (allowed by the best-case node layout scenario). The traversal engine supports issuing a new memory request one cycle after a data block is delivered to it, and can interleave multiple traversals at run time, allowing a single engine to support the parallel traversals described below. We used ISE 14.7 to compile the Verilog emitted by Bluespec. The synthesized traversal engines ran at 200 MHz. None of our engines utilized more than 10% of the XC7Z045.

We utilized a single 667 MHz ARM core in "bare-metal" mode for the software components of the implementations. The ARM core sends payload (in hybrid-push) or payload pointers (in hybrid-pull) to the fabric through the AXI master port by either PIO or DMA. When using DMA, we stage data by copying one or more payloads (or pointers) into one of several contiguous buffers in DRAM or OCM. Staging data achieves much higher bandwidth than PIO, and amor-

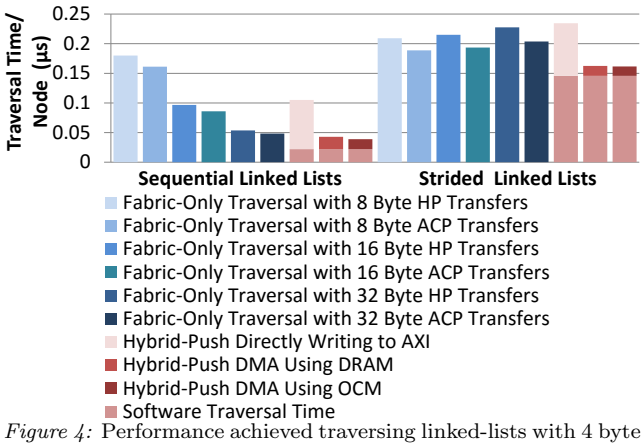

Figure 4: Performance achieved traversing linked-lists with 4 byte payload inlined within the linked-list nodes.

tizes the cost of initiating DMA transfers.[3] We built the software component with Xilinx SDK 14.7 with gcc -O3.

5.1.3 Observations

Below we present the most illustrative results from our comprehensive design study cases.

Single Linked-List with a 4-Byte Payload. Traversing a single linked-list with small payload (4-Bytes, inlined) presents the most difficult scenario. Figure 4 reports the per-node traversal time achieved under several implementation options. Results for both best-case and worst-case node layouts are reported. The fabric-only traversals reach a steady state for linked list with under 10 nodes. The hybrid-push traversals that use DMA require linked lists with 20 to 30 nodes in order to reach a steady state.

The first 6 bars in the series correspond to the fabric-only traversal engines using the HP or ACP interface to issue 8, 16 and 32 byte transfers. We see that using the ACP interface results in a slightly faster traversal time. For the best-case sequential layout, we see an improvement in traversal time as we increase the transfer size. This is due to spatial prefetching effects; the same is not observed for the worst-case strided layouts. In cases where prefetching is ineffective, the performance of traversing a single linked list with a small payload is entirely dictated by the platform's memory latency performance. What may be counterintuitive is that the Zynq, which does not have the DRAM bandwidth or capacity of the server systems, has the best memory latency performance due to its tight integration and hardwired memory path.

The next 3 bars in the series in Figure 4 correspond to hybrid-push using PIO, DMA with DRAM staging, and DMA with OCM staging. These bars are stacked to show how much time is spent by pointer chasing in the ARM core and how much time is spent in pushing data payload to the fabric. Hybrid-pull is not reported, as it does not make sense in this context since the ARM core would have to send the same size data (4-byte pointer or 4-byte payload per node).

We are encouraged to find a real-life example of the hybrid approach improving over the fabric-only approach on small payload scenarios. DMA from DRAM or OCM performed roughly equally;[4] and hybrid-push PIO performance is much worse because the ARM core actually stalls on each write (for approximately 100ns in our measurements) until it receives the bus response. As should be expected, software pointer-chasing is also much slower for the worst-case node layout, but hybrid-push still did better than the fabric-only implementations.

[3]DMA initiation costs are high enough that performing a DMA transfer for each linked list node is slower than PIO.

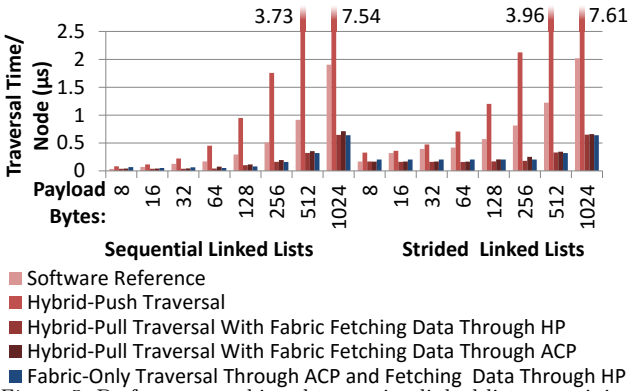

Figure 5: Performance achieved traversing linked-lists containing pointers to a larger data payload.

Single Linked-List with Varying Payload. Figure 5 reports the per-node traversal time of a single linked-list with payload sizes ranging from 8 Bytes to 1 KBytes. Again, the results for both best-case and worst-case layouts are reported. For this set of results, the linked-list nodes contain pointers to the payload (indirect payload). Five bars are shown for each payload size. The first bar, provided as a reference only, is the time of a software-only traversal. This software-only traversal touches all payload data but does not send payload data to the fabric (as required by the reference behavior). In many cases, especially for large payload sizes, the software-only traversal is in fact slower than sending the payload data to the fabric. This serves to provide another justification for delivering the payload into the fabric besides the assumed FPGA acceleration of processing.

The next 4 bars correspond to hybrid-push, hybrid-pull using HP for payload fetch, hybrid-pull using ACP for payload fetch, and a highly optimized fabric-only implementation that uses ACP for fetching the node structs as 32-byte blocks and HP for fetching the indirect payload.

We can see the advantage of hybrid-pull over fabric-only on the smaller payload sizes. Keep in mind that hybrid-pull is only valid for traversals that are independent of the payload values. As the payload size increases, hybrid-pull's advantage over fabric-only diminishes as the traversal time become dominated by the payload fetch time through the HP ports. Both hybrid-pull and fabric-only are able to reach 98% of the peak bandwidth of a 64-bit wide HP interface at 200 MHz.

When payload sizes exceed 4 bytes, hybrid-push is always slower than fabric-only. But for smaller payload sizes and payload dependent traversals, hybrid-push may still be a valid option due to its ease of development.

Interleaved Concurrent Traversals. For this final set of Zynq results, we return to linked lists with small 4-byte

[4]The DMA engine transferred data from the processor cache rather than memory, masking the performance difference between OCM and DRAM.

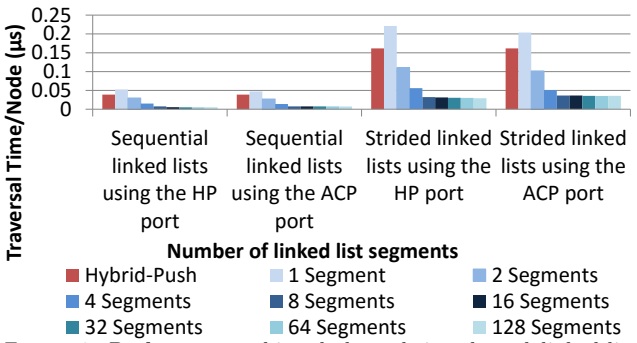

Figure 6: Performance achieved through interleaved linked-list traversal, including the best "Hybrid-Push" results from Figure 4 for reference.

inlined payloads, but achieve better performance through parallel memory accesses. We break the 16,384 element linked list into shorter linked list segments. These linked list segments are traversed in a multi-threaded fashion using a single traversal engine, which interleaves the memory requests by the different threads. Figure 6 reports the per-node traversal time when interleaving 1 to 128 concurrent traversals. Results are presented for the best-case and worst case node layouts, retrieving data using HP and ACP interfaces. The per-node traversal time of DMA hybrid-push from OCM is included as the first bar as a reference.

In all cases, the traversal time improves predictably as the number of concurrent traversals is increased up to 8 (reaching 93% of the HP interface's peak bandwidth). The effect is most dramatic in the worst-case data layout scenarios where all the references have to pay the full latency to DRAM. Issuing multiple outstanding reads from different traversals effectively keeps the traversal engine busy. For more than 16 traversals, the traversal time improves slowly because the HP bandwidth begins to saturate.

5.2 Intel QuickAssist QPI FPGA Platform

The Intel Heterogeneous Architecture Research Platform program has made the Intel QuickAssist QPI FPGA Platform [13] available for academic research use. The currently available Intel QuickAssist QPI FPGA Platform is a pre-production system that pairs a server-class multicore Xeon processor with an Altera Stratix V FPGA using Intel's cache-coherent QPI interconnect. The QPI interface extends shared memory to the FPGA, providing coherent access to DRAM attached to the processor as well as the processor's last-level cache. The Intel-provided soft-logic infrastructural IPs include the QPI interface, a 64-kilobyte cache, and the support for address translation. In our own benchmarking, we have seen the FPGA achieve approximately 6 GByte/sec of read bandwidth and about 350ns read latency over QPI to DRAM. Experimental results on this hardware platform reflect experiments performed at CMU. Results in this publication were generated using pre-production hardware and software, and may not reflect the performance of production or future systems.

Figure 7 reports the per-node traversal time for linked-list traversals with indirect payload of 8-1K bytes. As before, we report the traversal time under both best and worst case node layouts. There are four bars for each payload size in each layout case. The first bar, is a software-only reference traversal by the Xeon processor. This software-only traversal touches all payload data, but does not send payload data to the fabric.

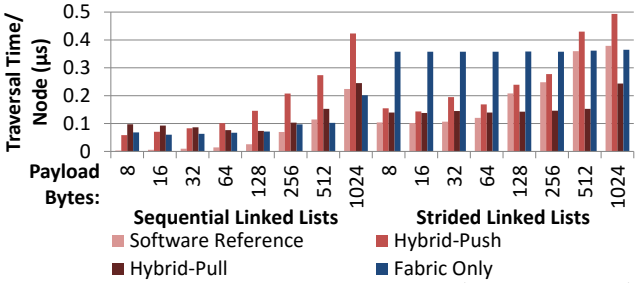

Figure 7: Traversal results on the Intel QuickAssist QPI FPGA Platform with indirect payload.

The next two bars are hybrid-push and hybrid-pull results, where the processor and fabric share the traversal effort to provide data to the fabric. Over increasing payload sizes, the traversal times stay about the same until the traversals switch from latency bound to bandwidth bound. As these cases are primarily latency bound, the processor's comparatively low-latency access to DRAM supports a hybrid traversal approach. However, without other interface options between the processor and the FPGA, all processor-FPGA interactions are implemented through shared memory using loads and stores. Thus, for hybrid-push, the processor copies the payload into a circular buffer in shared memory to be read by the FPGA. The hybrid-pull approach copies pointers to the payload into the buffer, and the FPGA fetches the data itself. The hybrid-push approach generally is slower than hybrid-pull, but can perform well with very small sequential payloads, where it can take advantage of the Xeon's superior cache, higher clock speed, and lower memory access latency.

The fourth bar shows fabric-only traversals. Fabric-only traversals were faster for sequential traversals due to spatial prefetching effects (the same observed in Figure 4). Each 64-byte cache line packs 8 sequential linked list nodes into each cache line. Furthermore, minimal latency between the hardware components fetching the linked list nodes and the hardware components fetching the payloads contributed to the speedup in sequential lists, as compared to the hybrid-pull approach. Overall performance in sequential traversals was still latency limited (by the latency of fetching payload data) until the payload size reaches 256 bytes. As before, fabric-only traversals of strided linked lists were the slowest. However, we achieved similar performance improvements to those in Section 5.1 by parallelizing the traversal: breaking up the strided linked lists into up to 8 segments and traversing these segments in parallel yielded linear performance improvements, and surpassed the performance of the hybrid traversals. We believe that future systems could improve the performance of hybrid approaches by providing low-overhead messaging and data transfer channels that reduce the synchronization overheads that applications incur when performing fine-grained communication.

5.3 Convey HC-2EX

Convey's HC-1 FPGA computing system featured shared memory with Intel host processors. The current generation HC-2 system replaces hardware shared memory with an FPGA system on a PCI-E bus containing a large capacity DRAM memory system, emulating a shared memory

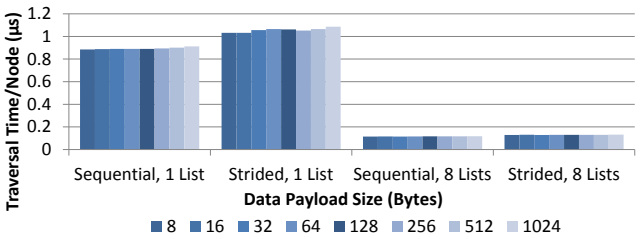

Figure 8: Parallel traversal results on the Convey HC-2EX with indirect payload.

abstraction in software. Due to this hardware implementation, our design study on the HC-2 focuses on the fabric-only approach. Even without shared memory, the HC-2, with its high-performing, high capacity DRAM subsystem, is still relevant to the acceleration of pointer-based irregular parallel applications.

We worked with a Convey HC-2EX [4] that includes proprietary "scatter-gather" memory modules. These unique memory modules allow the Convey machine to achieve peak memory bandwidth even on non-sequential accesses, up to 20GB/sec for each of its 4 user-logic FPGAs. Convey's memory subsystem is implemented in soft-logic using additional dedicated FPGAs. At first glance, Convey's emphasis on supporting irregular memory access patterns should be beneficial to pointer-chasing irregular parallel applications. For parallel traversal experiments on the Convey, we attached a separate Convey-optimized traversal engine to each of the user-visible memory ports. We do not show software or hybrid traversal results on Convey because the processor must go through the PCI-E interconnect to accesses the FPGA-attached DRAM.

Figure 8 reports the per-node traversal time for linked-lists with indirect payloads of 8 to 1K bytes. The left two clusters are for traversing a single linked-list under best-case and worst-case node layout, respectively. In this case, a single traversal engine utilizes 2 of the 16 memory crossbar ports available to one user-logic FPGA, one for fetching the node structs and one for fetching the payload data. The right two clusters are for traversing eight linked-lists concurrently using 8 different traversal engines and all 16 memory crossbar ports, still from one user-logic FPGA.

The Convey's performance is barely sensitive to node layout in memory, due to its unique support for non-sequential memory accesses. The Convey's per node traversal time is also not very sensitive to payload size. This is in part explained by Convey's very high memory bandwidth, but it also points to memory latency as the dominant effect in the single linked list traversal results. It may be surprising that the Convey's traversal time on a single linked-list is rather unexceptional, only on par with the Zynq. Recall that the Convey's memory subsystem stands out for its DRAM bandwidth. Even if Convey did not trade longer latency for better bandwidth optimizations, they would incur an unavoidable latency penalty by implementing their memory subsystem in soft logic. In the latency dominated traversals of a single linked list, the Convey can have no advantage over the hardwired memory subsystem in the Zynq. However, the Convey has plenty of bandwidth to sustain multiple current traversals without interference, as seen in the improvements in the 8-way traversal results. Even the 8-way traversal results do not even come close to saturating the bandwidth available on the Convey platform.

6. DISCUSSIONS

In this section we attempt to extrapolate from our study of pointer chasing.

Irregular Parallel Applications. Irregular parallel applications present an ideal target for shared-memory processor-FPGA acceleration. The demands of more regular applications could largely be satisfied by simpler solutions of memory bandwidth and capacity improvements. It is an irregular parallel application—with the requirement for irregular access over a large memory footprint and the opportunity for tightly coupled processor-FPGA collaboration—that really calls for the full potential of shared-memory processor-FPGA architectures.

Hardwired Memory Subsystem. With the growing use of FPGAs for compute acceleration, it is not inconceivable that an FPGA designed with computing in mind should come with a hardwired memory subsystem. We saw in the Zynq example that a modest commitment of die area to hardwiring the memory subsystem can yield large gains in performance and power efficiency over soft logic. To go one step further, there should be an entire compute-oriented FPGA memory architecture that addresses computing needs, ranging from memory hierarchy and cache-coherence nuts-and-bolts, to memory virtualization and protection, to what is the best presentation of memory to the fabric for use by spatial computing kernels.

Memory Performance. With hardwiring of the memory subsystem, there also needs to be a decision on the level of memory performance desired for FPGAs. With the exception of Convey, FPGAs and FPGA acceleration systems have generally been under-powered in their DRAM bytes-per-second relative to their potential for ops-per-second. Additional memory performance can go toward both unleashing computing performance and increasing application generality. Our study pointed especially to the importance of latency performance for irregular parallel applications. This can be achieved by both direct (faster, shorter links) and indirect (caches and prefetchers) means.

Memory Parallelism. Our study also repeatedly pointed to memory parallelism as a powerful technique to overcome memory latency. Memory parallelism should play an important role in shared-memory processor-FPGA system design. This can manifest at all levels, ranging from memory system design, to accelerator development tools, to algorithms. For our study, a tool that automated the process of creating parallelized traversal engines (possibly with annotations by the application developer indicating valid parallelization opportunities) would have made it much easier to achieve the best possible performance on data structure traversals.

Processor-FPGA Interactions. One motivation of the current work was to show that tightly-coupled processor-FPGA interactions in shared memory architectures can be used to improve performance (hybrid-push, hybrid-pull). Although not demonstrated by this study, there clearly are large dividends in eliciting software assistance to handle complex or non-critical tasks. Both hardware and software infrastructural support to simplify and speed up processor-FPGA interactions warrant increased attention in future shared-memory processor-FPGA systems. Although shared memory is elegant in its generality, there is room for special interfaces such as fast, short messaging FIFOs. The need for

performance and efficiency when accellerating in hardware can tilt the balance toward efficiency rather than elegance.

7. CONCLUSION

The conceptual simplicity of linked lists can belie their significance and its complexity. In this paper, we used a reference linked list traversal behavior as a proxy to study the potential of shared-memory processor-FPGA systems in accelerating irregular parallel applications that rely on pointer-based data structures and algorithms. We examined a broad range of implementation considerations and options both at a high level and through concrete implementations on real systems. We point to memory latency as the dominant performance factor when traversing a single linked list, and observe that interleaving multiple concurrent traversals can overcome stalls caused by memory latency. We find that incorporating tightly-coupled processor assistance can also be an effective approach.

8. ACKNOWLEDGEMENTS

CMU authors are supported in part by NSF CCF-1320725 and by Intel. The authors thank Altera, Xilinx, Intel, and Bluespec for tools and hardware donated to CMU.

9. REFERENCES

[1] Altera, Inc. Arria 5 Device Overview, January 2015. AV-51001.

[2] Bruce Wile. Coherent Accelerator Processor Interface(CAPI) for POWER8 Systems, September 2014.

[3] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 245–254, 2013.

[4] Convey Computer Corporation. Convey Personality Development Kit Reference Manual, April 2012. Version 5.2.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[7] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:45–52, 2012.

[8] Geoffrey Hinton, Li Deng, Dong Yu, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath George Dahl, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, November 2012.

[9] Skand Hurkat, Jungwook Choi, Eriko Nurvitadhi, José F. Martínez, and Rob A. Rutenbar. Fast Hierarchical Implementation of Sequential Tree-reweighted Belief Propagation for Probabilistic Inference. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications*, FPL '15, September 2015.

[10] Maysam Lavasani, Hari Angepat, and Derek Chiou. An FPGA-based In-Line Accelerator for Memcached. *Computer Architecture Letters*, 13(2):57–60, July 2014.

[11] Rishiyur Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[12] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 25–28, May 2014.

[13] N. Oliver, R.R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Yaping Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A reconfigurable computing system based on a cache-coherent fabric. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 80–85, Nov 2011.

[14] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware, February 2015.

[15] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[16] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[17] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform. In *Proceedings of the 25th International Conference on Field Programmable Logic and Applications*, FPL '15, September 2015.

[18] Xilinx, Inc. Zynq-7000 All Programmable SoC Overview, October 2014. v1.7.