# Traversal Caches: A First Step towards FPGA Acceleration of Pointer-Based Data Structures

Greg Stitt, Gaurav Chaudhari, James Coole
University of Florida
Department of Electrical and Computer Engineering
{gstitt, guchaudhari, jcoole}@ufl.edu

## ABSTRACT

Field-programmable gate arrays (FPGAs) often achieve order of magnitude speedups compared to microprocessors, but typically have been unable to improve the performance of applications with irregular memory access patterns, such as traversals of pointer-based data structures. Due to the common use of these data structures, the applicability and widespread success of FPGAs has been limited. In this paper, we introduce the *traversal cache* framework – a first step towards improving the performance of FPGA applications that utilize pointer-based data structures. The traversal cache is a local FPGA memory that stores repeated traversals of pointer-based data structures, allowing for these traversals to be efficiently streamed into the FPGA. Although the cache is generally limited to improving applications that exhibit repeated traversals, we show that many applications in fact have this characteristic. Furthermore, we show that few repetitions are needed to achieve performance improvements. We present experimental results showing that FPGA implementations using the traversal cache framework achieve speedups ranging from 7x to 29x compared to pointer-based software on a 3.2 GHz Xeon.

## Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems.*

## General Terms

Performance, Design.

## Keywords

Traversal cache, FPGA, pointers, synthesis, CAD, hardware/software partitioning.

## 1. INTRODUCTION

Field-programmable gate arrays (FPGAs) and other reconfigurable computing devices have been shown to achieve 10x to 100x speedups compared to state-of-the-art microprocessors for many applications [5][11]. FPGAs achieve such speedup by exploiting tremendous amounts of parallelism, ranging from the bit level up to the task level. FPGA designs often use heavily pipelined implementations to improve throughput, which greatly increases memory bandwidth requirements [14]. If data cannot be delivered at a sufficient rate, these pipelines frequently stall, often resulting in significant slowdown compared to microprocessors [12].


*CODES+ISSS'08*, October 19–24, 2008, Atlanta, Georgia, USA.


Due to the need for efficient data transfer, FPGAs have typically been unable to effectively implement code with irregular access patterns [6]. We define an irregular access pattern as any access that does not fetch data sequentially from memory, or patterns that cannot be buffered based on compile time analysis [10]. Although there are many types of problematic irregular access patterns, in this paper we focus on pointer-based patterns, which cause several performance problems for FPGA implementations. First, traversals of pointer-based structures, such as lists, require multiple memory accesses to fetch a single item of data. Second, data in consecutive nodes of pointer-based structures is rarely stored at consecutive memory locations, requiring inefficient non-sequential memory accesses that use many lengthy row address strobes (RAS), which typically take much longer than a column address strobe (CAS) [8]. Non-sequential accesses also prohibit use of specialized burst access modes [8]. Furthermore, many FPGAs access memory using DMA units that only support block transfers. Due to the common use of pointer-based data structures, inefficient FPGA performance for these structures has limited the widespread success of FPGAs.

In this paper, we present the *traversal cache* framework, which often eliminates the performance limitations caused by pointer-based data structures on FPGAs. The traversal cache framework is motivated by the observation that applications often traverse portions of a pointer-based data structure multiple times before either changing the data structure, or changing the way the structure is traversed. The traversal cache framework capitalizes on this characteristic by reordering data involved in repeated traversals and caching that data sequentially in a local FPGA

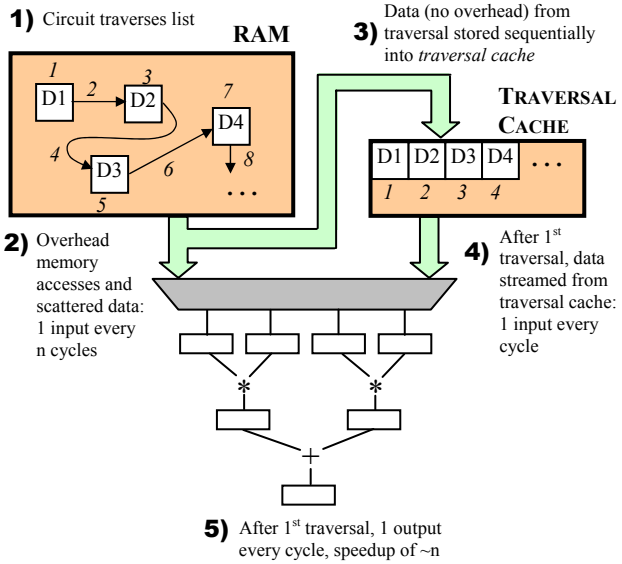


**Figure 1:** Conceptual idea of *traversal caches*, where repeated traversals of pointer-based structures are stored sequentially to improve memory bandwidth to custom circuits.

memory, as is illustrated in Figure 1 for a simple linked list. When executing a traversal for the first time, the FPGA traverses the list (with assistance from a microprocessor) and passes the data to the datapath while simultaneously storing the data sequentially into a traversal cache, allowing all future repeated traversals to avoid overhead memory accesses and lengthy RAS operations. Traversal caches improve the memory bandwidth of pointer-based data structure traversals, yielding speedup as high as 29x compared to pointer-based code running on a 3.2 GHz Xeon.

Of course, traversal caches are limited to improving the performance of repeated traversals of a data structure. If the data structure changes, or if the algorithm traverses the structure differently, the framework must invalidate the traversal cache and read the new traversal from memory. Clearly, the overall performance improvements from traversal caches depend on how frequently the traversal cache is invalidated, which we refer to as the *invalidation rate*. We have observed that many applications do in fact have a low invalidation rate, and are therefore amenable to the traversal cache framework. For example, an n-body simulation may traverse a quad tree structure hundreds or even thousands of times in the same way.

The paper is formatted as follows. Section 2 discusses previous work. Section 3 gives the details of the traversal cache framework. Section 4 presents experimental results.

# 2. PREVIOUS WORK

Previous work has investigated hardware synthesis techniques for code utilizing pointers. In [18], Semeria integrated alias analysis techniques into a high-level synthesis tool flow to help resolve aliases at compile time, thus enabling further optimization and utilization of multiple memories. These techniques were later extended in [19] to support dynamic memory allocation by integrating a memory manager into the synthesized circuit. The traversal cache framework is a complementary approach that targets hardware/software codesign by not restricting the hardware to using a separate memory management unit – a situation that may not be practical or efficient for all FPGA accelerators.

Diniz [6] utilized FPGAs to create smart memory engines capable of reorganizing data from pointer-based data structures to improve data locality and cache performance. The traversal cache framework has a similar goal, but does not reorder data in main memory, and thus does not have the alias restrictions of [6]. Furthermore, the traversal cache framework can be applied to any FPGA accelerator, including those that access memory via DMA.

Impulse [2] introduced a memory controller that remaps physical addresses to improve cache performance and memory bandwidth. Impulse remapped data using specialized languages and operating system support. The traversal cache framework does not have these restrictions, and only requires use of a specific library.

Specialized cache architectures and memory allocation techniques have also been introduced to better handle pointer operations. Collins et al. [4] introduced the pointer cache to efficiently handle chained pointer traversals by prefetching data based on pointer transitions. Weinberg [20] eliminates pointer-based memory accesses at runtime by caching previous evaluation results. Hu [13] predicts memory behavior and utilizes a time-based victim cache to improve hit rate. Chilimbi et al. [3] discuss manual programming practices that can improve data locality of pointer structures, in addition to automatic data layout optimizations that are integrated into garbage collection. Calder [1] considers cache-conscious data placement for heap and stack objects. The traversal cache framework provides similar optimizations for FPGAs, which commonly have direct access to memory and therefore do not benefit from traditional cache optimization.

Smart buffers [10] are a data caching scheme for FPGAs that prevent reused data from being read multiple times from memory. Smart buffers greatly improve memory bandwidth and FPGA performance, but do not support pointer-based data structures.

Numerous compiler optimizations [8][9][17] modify data layout at compile time based on memory access patterns. Traversal caches improve on these previous approaches by supporting pointer-based data structures.

# 3. TRAVERSAL CACHE FRAMEWORK

In this section, we discuss the functionality required to enable the traversal cache framework, which includes the system architecture, hardware/software communication, and traversal-cache software library.

## 3.1 System Architecture

The system architecture used by the traversal cache framework is illustrated in Figure 2(a). The architecture consists of two main components: the microprocessor and the FPGA accelerator. The microprocessor executes all software regions of the code, and assists in fetching traversals anytime the traversal cache is empty or invalidated. The framework supports any type of microprocessor but currently uses a 3.2 GHz Xeon. The FPGA accelerator implements pipelined datapaths for computation-intensive kernels, while using the traversal cache to efficiently handle pointer-based data structures.

The system architecture utilizes three memories. The microprocessor uses one memory and the FPGA uses two memories to enable simultaneous streaming of inputs and outputs. Although not shown in the figure, the framework allows data to be copied between the microprocessor memory and FPGA memories. Communication details are discussed in the following section.

The architectural details of the FPGA accelerator are shown in Figure 2(b). The controller interprets messages from the microprocessor, and enables the address generators when the microprocessor activates the accelerator. The address generators control the input and output memories in order to read or store a data structure traversal. The address generators currently support sequential memory accesses, or linear accesses with a specified stride. The datapath is a pipeline customized for each specific kernel. The invalidation logic determines runtime traversal cache invalidations. More details are given in section 3.3.

The input memory hierarchy used by the accelerator consists of three possible data sources. The accelerator uses a RAM for normal array traversals, for which data can efficiently be streamed into the datapath by the address generators. Although the framework currently utilizes external SRAM and on-chip block RAM, any type of memory is supported. In addition to the RAM, the input memory hierarchy uses the traversal cache to store pointer-based data structure traversals in sequential order. The assisted data buffer stores data from the software assisted data stream as it arrives from the microprocessor, while simultaneously being stored into the traversal cache.

The output memory hierarchy is similar to the input memory hierarchy, consisting of a separate RAM, and possibly a separate traversal cache. The output traversal cache enables efficient

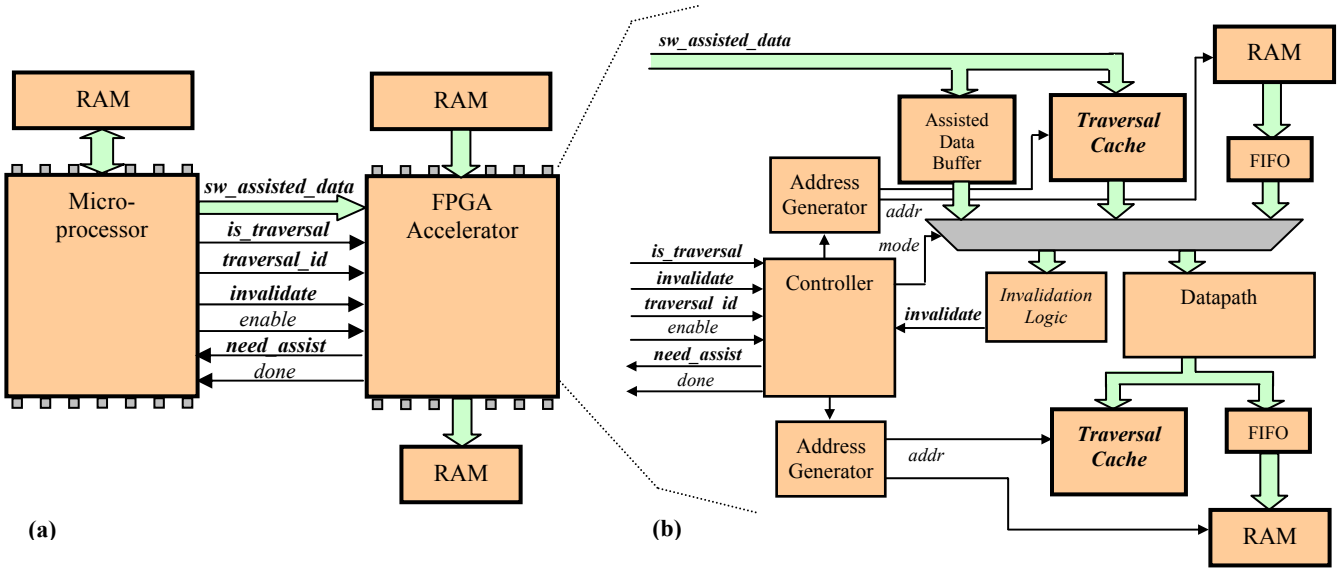


**Figure 2:** (a) The *traversal cache* framework, which initially utilizes a microprocessor to assist with pointer-based data structure traversals. During an assisted traversal, the (b) FPGA accelerator stores the traversal sequentially into the traversal cache, allowing future executions of the same traversal to efficiently stream data into the datapath.

streaming of outputs for code that writes to a pointer-based data structure.

Although given the name "cache", the framework allows the traversal cache to be implemented in several ways. If the stored traversal is small enough, the traversal cache may be stored in on-chip block RAM. For systems with an integrated CPU, the traversal cache may potentially be implemented using scratchpad memory. In the general case, the framework implements the traversal cache using some form of external memory, such as SRAM or SDRAM. Although utilizing an external memory may seem like an inefficient use of resources, many FPGA accelerator boards have numerous memories [7][16] that may otherwise be unutilized. Alternatively, the traversal cache could share the same memory with other data needed by the accelerator.

We point out that although we present a specific system architecture in this paper, the framework supports numerous other architectural possibilities. For example, the microprocessor and FPGA may be tightly coupled, existing on either the same chip or the same board, or loosely coupled, where the microprocessor and FPGA communicate using a PCI-X bus or network. The microprocessor may be implemented on an FPGA as either a soft core [21], or as a hard core [22]. Another alternative is the use of a shared memory between the microprocessor and FPGA, which requires tighter coupling, but avoids the situation of having to copy data before the accelerator can start – a situation that may limit speedup.

## 3.2 Hardware/Software Communication

Communication between the microprocessor and FPGA consists of the signals shown in Figure 2(a). For simplicity, we do not show signals for copying data to/from the microprocessor memory into the local FPGA memories. However, that communication is of course included by the implementations used in the experiments. The bold signals are unique to the traversal cache framework, while all other signals are used for normal synchronization.

When the microprocessor reaches a kernel that requires a pointer-based data structure traversal, the microprocessor asserts the *is_traversal* control signal to inform the accelerator that a traversal is being performed. If the traversal cache is invalid, meaning that the traversal is not stored in the traversal cache, then the accelerator responds by asserting the *need_assist* signal. If the accelerator requires assistance, the microprocessor first enables the accelerator using the *go* signal, and then performs the traversal in software, fetching the data that is needed, and passing that data to the accelerators in the software assisted data stream (*sw_assisted_data*). The accelerator datapath processes this data as it arrives, often requiring many stall cycles, which is one reason pointer structures have previously been inefficient. However, as the data arrives, the accelerator stores the data in sequential order into the traversal cache so that future repetitions of this traversal can be handled more efficiently. After sending the entire traversal to the accelerator, the microprocessor deasserts the *is_traversal* signal and waits for the accelerator to complete. After processing the entire traversal, the accelerator asserts the *done* signal, which allows the microprocessor to resume software execution.

In the case that a pointer-based traversal is not needed, the microprocessor simply copies the needed data to the FPGA memory, and then asserts the *go* signal, in which case the accelerator fetches data from RAM instead of from the traversal cache.

The microprocessor uses the control signal *invalidate* to inform the FPGA that the data in the traversal cache is invalid and should no longer be used. Invalidations may result from a number of situations that are described in section 3.3.

In the case of multiple traversal caches, the framework utilizes a traversal identifier signal (*traversal_id*) to specify which traversal cache should be used for the current traversal. This identifier allows for multiple traversals to be utilized in the case that an application repeatedly traverses different data structures, or the same data structure in different ways.

The framework currently implements all control and synchronization signals using memory mapped registers inside the FPGA. The microprocessor writes the software-assisted data stream directly into the assisted data buffer, which is either a SRAM or block RAM depending on the size. All communication occurs over a PCI-X interface. The FPGA uses additional logic

not shown in the figure to route data from the PCI-X bus into the appropriate location in the FPGA. We point out that the framework is independent of the communication architecture, potentially supporting any underlying architecture that can implement the discussed control signals.

## 3.3 Software Library

To utilize the traversal cache, the accelerator requires assistance from a software library running on the microprocessor. The library is responsible for specifying when a traversal occurs, detecting invalidations of a traversal, and passing data to the accelerator when the traversal cache is empty or invalidated. We currently implement this functionality using a library of wrapper functions around standard data structures, but plan to automate this process as part of a high-level synthesis and hardware/software partitioning tool, which could theoretically allow a user of the framework to utilize any library.

To specify a traversal, the wrapper functions send the appropriate control signals to the FPGA, wait to see if assistance is needed, and then begin fetching the data used by the traversal. The library uses a different wrapper function for each type of traversal, such as an in-order tree traversal, a depth first search of a graph, etc.

The most challenging task required by the software is to detect traversal invalidations. Currently, the framework invalidates the cache anytime the data stored in the data structure is changed, and also when the traversal changes (e.g. in-order to post-order). Detecting changes to the data structure is a challenging problem, due to aliasing issues that may exist in the code. To avoid these issues, the framework requires that any changes made to the data structure be made through the use of the wrapper functions. This requirement guarantees that the traversal cache will be invalidated for any modification to the data structure. Furthermore, this requirement does not restrict the use of aliases outside the wrappers because those aliases cannot modify the structure.

The software library detects changes in the ordering of a traversal in the following way. In the simple case that the ordering of the traversal is known at compile time, the code simply uses a different wrapper function for each type of traversal. However, not all traversals follow a path through the data structure that is known at compile time. Binary search is an example of this problem, where the traversal of the data structure depends on the values in both the data structure and the input value. For cases where the software library cannot determine invalidation until runtime, the accelerator utilizes invalidation logic to dynamically check for invalid traversals. This runtime analysis requires extra data to be stored in the traversal cache to allow the accelerator to determine if the actual traversal differs from the stored traversal. If the accelerator invalidates the traversal cache at runtime, the accelerator requests assistance from the microprocessor to fetch the correct traversal and then restarts execution.

## 3.4 Limitations

The main limitation of the traversal cache framework is that not all applications using pointer-based data structures are amenable to speedup. To achieve speedup, the application typically must have repeated traversals. However, as shown in the experiments section, few repetitions are typically needed before the traversal cache achieves improvements. In fact, for some computational-intensive applications, speedup can be obtained even if the traversal cache is invalidated for every traversal.

Another limitation is that the traversal cache must be manually created and the specified software library must be used. Ideally, a hardware/software partitioning tool could partition the application automatically, high-level synthesis could determine the appropriate size and amount of traversal caches, and also modify the software source code appropriately for use with any data structure library. These issues are outside the scope of this paper, but we plan to introduce synthesis techniques for traversal caches as part of future work.

# 4. EXPERIMENTS

## 4.1 Experimental Setup

To evaluate the traversal caches, we implemented the framework on the Nallatech H101-PCIXM [16]. This board consists of a Xilinx Virtex 4 LX100 FPGA, in addition to 4 SRAM banks. We mapped the framework onto this target architecture in the following way. The microprocessor is a 3.2 GHz Xeon, and the FPGA is the LX100 on the H101-PCXM. All communication between the microprocessor and FPGA, which includes the software-assisted data stream, control signals, and synchronization signals are sent over a PCI-X bus. Control and synchronization signals are read from and written to memory mapped registers inside of the LX100 using memory map nodes provided by Nallatech. The software assisted data stream is written into one of the SRAM banks, or alternatively block RAM on the LX100 if the traversal is small enough. To test realistic traversal sizes, all of our tests utilize the SRAM. Utilizing block RAM could potentially improve the reported results due to increased memory bandwidth. The traversal cache is also implemented by one SRAM bank. Although traversal caches can potentially be used for outputs, in this paper the tested examples do not require an output cache, and instead write outputs directly to an SRAM bank. The invalidation logic was also not required for any of the tested examples because we could determine all possible invalidations at compile time. The selected examples are representative of many FPGA-amenable kernels, which implies many examples may not need the invalidation logic. We plan to determine of the effects of the invalidation logic as future work.

To investigate effects of invalidation rate on performance, we do not base the invalidation rate on a specific input stream, and instead manually test different invalidation rates. This approach allows us to test different input possibilities, ranging from the worst case to the best case. For each example, we test invalidation rates of 1 (invalidate every traversal), 5 (invalidate every 5 traversals), 10, 20, 40, and 80.

We evaluated the framework using the following benchmarks, which we developed. For each benchmark, we describe the pointer-based data structure and justify tested invalidation rates.

*Search* scans a linked list of 1 million 16-bit integers and determines the number of occurrences of a specified value. The FPGA implementation performs 16, 16-bit comparisons and 15 additions every cycle. The implementation consisted of 3933 lines of VHDL and 1409 lines of C code. The invalidation rate for *search* is likely to be low for any application that searches a data structure multiple times without changing, such as a database application.

*Audio* performs convolution of an input signal consisting of 16-bit audio samples with a 64 sample impulse response. The data structure used by audio is a linked list of audio streams, which may likely occur in a digital audio workstation or a video game. Repeated traversals are likely since the actual audio stored in these applications does not change frequently. The circuit implementation performs 64 multiplications and 63 additions

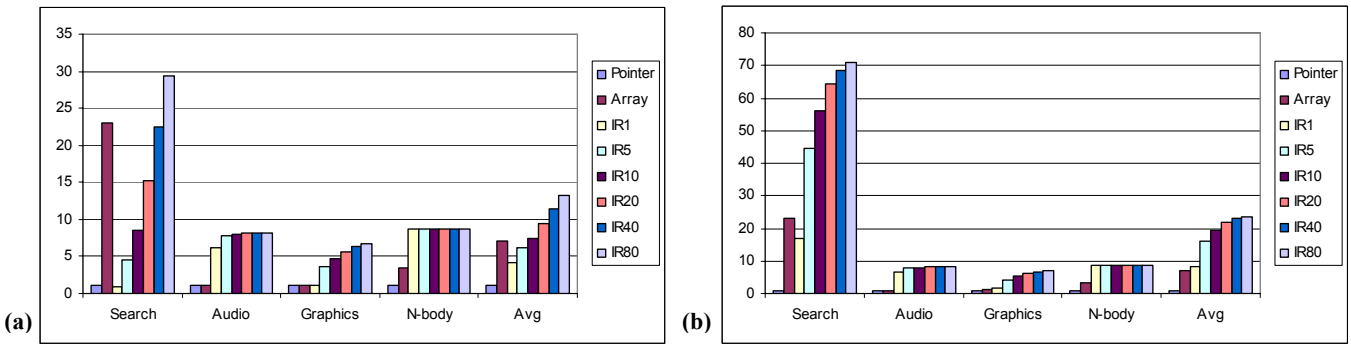


**Figure 3:** (a) Speedups obtained by the traversal cache framework for invalidation rates (IR) ranging from 1 to 80, compared to pointer-based software (*pointer*) running on a 3.2 GHz Xeon. Performance of array-based software (*array*) is also shown. The results show that even high invalidation rates (low IR values) typically achieve large speedup. (b) The same comparisons after recoding the traversal cache implementations to use arrays instead of pointer-based data structures. The results show increased speedup for *search*, but almost identical speedup for the other examples, which suggests traversal caches often completely hide the overhead of pointer-based structures.

every cycle, and required 4496 lines of VHDL and 1399 lines of C code.

*N-body* determines the forces exerted on 100,000 particles for 80 time steps. The data structure used by *n-body* is a quad tree, which recursively divides the 2-dimensional space that contains the particles into small subspaces. *N-body* performs 8 multiplications, 2 divides, 2 additions, 2 subtractions, 1 square root, and 2 accumulations every cycle. All operations are floating point except for the accumulations. We are implementing a pipelined floating point accumulator, but currently use a large fixed point accumulator with 32 integer bits and 24 fraction bits. *N-body* required 4865 lines of VHDL and 1439 lines of C code. The VHDL utilized a floating point multiplier, divider, square root, adder, subtracter, float-to-fixed, and fixed-to-float component from Xilinx CORE Generator. The quad tree structure is normally used by the Barnes-Hut algorithm to reduce the complexity of n-body simulations from $O(n^2)$ to $O(n*\lg n)$. However, for our experiments, the circuit implementation searches the entire quad tree to avoid inflating the benefits of traversal caches compared to an array implementation. Therefore, actual execution times using a quad tree, with or without traversal caches, is likely better than reported.

*Graphics* performs 3-dimensional vertex transformations by multiplying 4x4 transformation matrices with 4x1 vertex matrices. *Graphics* uses a list of vertex arrays, where each node of the list represents an object to be rendered. The implementation performs 16 multiplications and 28 additions every cycle. All operations are floating point. The implementation used 4166 lines of VHDL and 1435 lines of C code, in addition to a floating point multiplier and adder from Xilinx CORE Generator.

For each benchmark, we manually performed hardware/software partitioning and then created a custom accelerator for the single most computationally-intensive kernel, using the traversal cache framework. We specified the entire framework using VHDL, and then specified the remainder of the FPGA environment using Nallatech DIMEtalk [15], which included the PCI-X interface, SRAM memory controllers, and memory map interfaces. We synthesized the resulting VHDL using Xilinx ISE 8.1.

We executed each example at the maximum possible clock frequency obtained after placement and routing, which ranged from 115 MHz for *graphics* to 135 MHz for *search*.

For all experiments, we compare traversal cache performance to software running on a 3.2 GHz Xeon. We compiled each benchmark using gcc 3.4.6 with –O3 optimizations to ensure that the baseline in the comparisons was as fast as possible.

## 4.2 Speedup Compared to Software

This section presents performance advantages of the traversal cache framework compared to software running on a 3.2 GHz Xeon. Figure 3(a) shows speedups obtained by the traversal cache compared to pointer-based software (shown as *pointer* in the figure). The figure includes speedups for each invalidation rate (abbreviated *IR*). The figure also shows the performance of software using sequential array traversals (shown as *array* in the figure) instead of pointer-based data structure traversals.

For the *search* example, only the highest invalidation rate (IR1) was slower than the pointer-based software. All other invalidation rates achieved large speedup compared to pointer-based software, reaching as high as 29x for IR80. Array-based software performance was better than the traversal cache framework for all invalidation rates under 40. For the IR80 case, the traversal cache was 1.3x faster than the software array implementation.

For *audio*, all invalidation rates, including IR1, were faster than both the pointer-based software implementation and the array implementation. Speedup ranged from 6.2x to 8.2x. The reason for the increased speedup at higher invalidation rates was because *audio* performed more computation for each piece of data, minimizing the effects of transferring the data to the FPGA.

*Graphics* achieved similar results, always outperforming the pointer and array software implementations, with speedup ranging from 1.2x to 6.7x.

For *n-body*, the traversal cache framework also outperformed software for all invalidation rates. The main difference for this example was the lack of speedup increase for lower invalidation rates. This difference resulted from a significant amount of computation for each piece of data sent to the FPGA, which completely dominated the data transfer times. A similar result can be seen for the *audio* example, where the slope of speedup increase is less than the other examples. Speedup for *n-body* was 8.7x for all invalidation rates compared to pointer-based software, and 2.5x compared to array-based software.

## 4.3 Speedup after Recoding

To determine how much improvement could be obtained by using arrays within the traversal cache framework instead of pointer-based data structures, in this section we report the speedups assuming a designer were to recode the benchmarks to use arrays.

Figure 3(b) illustrates the speedup, again compared to pointer-based software, after recoding. For *search*, speedup more than

doubled, reaching 70x for IR80. The increased speedup resulted from the significantly slower pointer-based software, which was more than 20x slower than the array-based software. We believe this performance difference is due to page faults caused by the large list size. All other examples achieved almost identical performances after being implemented with arrays. This surprising result implies that for certain examples, traversal caches make pointer-based code just as efficient on FPGAs as array-based code – a significant achievement considering the traditionally bad performance that has resulted from pointer-based structures.

# 5. CONCLUSIONS

In this paper, we introduced the traversal cache framework. By caching repeated traversals of pointer-based data structures, traversal caches deliver data to custom pipelined datapaths implemented in FPGAs faster than previously possible, resulting in speedups as high as 29x compared to software execution on a 3.2 GHz Xeon. Furthermore, for several examples, performance was almost identical after recoding to eliminate pointers, which implies that traversal caches may often completely hide the overhead of pointer-based structures. Although some applications with high invalidation rates may not be amenable to traversal caches, there are numerous applications that benefit from this framework. Furthermore, we showed that even for applications with high invalidation rates, traversal caches can achieve significant speedup.

# 6. ACKNOWLEDGMENTS

The authors thankfully acknowledge equipment donations from Nallatech and Xilinx.

# 7. REFERENCES

[1] Calder, B., Krintz, C., John, S. and Austin, T. 1998. Cache-conscious data placement. Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems, pp. 139-149.

[2] Carter, J., Hsieh, W., Stoller, L., Swanson, M., Zhang, L., Brunvand, E., Davis, A., Kuo, C., Kuramkote, R., Parker, M., Schaelicke, L., and Tateyama, T. 1999. Impulse: building a smarter memory controller. Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), pp. 70-79.

[3] Chilimbi, T.M., Hill, M.D., and Larus, J.R. 2000. Making pointer-based data structures cache conscious. Computer, Vol 33, Issue 12, December 2000, pp. 67-74.

[4] Collins, J., Sair, S., Calder, B., and Tullsen, D. 2002. Pointer cache assisted prefetching. Proceedings of the International Symposium on Microarchitecture (MICRO), pp. 62-73.

[5] DeHon, A. 2000. The density advantage of configurable computing. Computer, Vol. 33, Issue 4, April 2000, pp 41-49.

[6] Diniz, P. and Park, J. 2003. Data search and reorganization using FPGAs: application to spatial pointer-based data structures. Proceedings of the Symposium on FPGAs for Custom Computing Machines (FCCM), pp. 207-217.

[7] GiDEL. GiDEL PROC Boards, 2008. http://www.gidel.com/PROCBoards.htm.

[8] Grun, P., Dutt, N., and Nicolau, A. 2001. Access pattern based local memory customization for low power embedded systems. Proceedings of Conference on Design, Automation, and Test in Europe (DATE), pp. 778-784.

[9] Grun, P., Dutt, N., and Nicolau, A. 2000. Memory aware compilation through accurate timing extraction. Proceedings of the International Design Automation Conference (DAC), pp. 316-321.

[10] Guo, Z., Buyukkurt, A. B., and Najjar, W. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. Proceedings of the ACM Symposium On Languages, Compilers and Tools for Embedded Systems (LCTES).

[11] Guo, Z., Najjar, W., Vahid, F., and Vissers, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pp. 162-170.

[12] Holland, B., Nagarajan, K., Conger, C., Jacobs, A., and George, A. 2007. RAT: a methodology for predicting performance in application design migration to FPGAs. Proceedings of the Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA), pp 1-10.

[13] Hu, Z., Kaxiras, S., and Martonosi, M. 2002. Timekeeping in the memory system: prediction and optimizing memory behavior. Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 209-220.

[14] Jacob, J. and Chow, P. 1999. Memory interfacing and instruction specification for reconfigurable processors. Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 145-154.

[15] Nallatech Inc. DIMEtalk 3, 2008. http://www.nallatech.com/?node_id=1.2.2&id=19.

[16] Nallatech Inc. Nallatech PCIXM FPGA accelerator card, 2008. http://www.nallatech.com/?node_id=1.2.2&id=41.

[17] Panda, P.R., Catthoor, F., Dutt, N., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandercappelle, A., and Kjeldsberg, P.G. 2001. Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 6, Issue 2, 2001, pp. 149-206.

[18] Semeria, L. and De Micheli, G. 1998. SpC: synthesis of pointers in C. Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 340-346.

[19] Semeria, L., Sato, K., and De Micheli, G. 2001. Synthesis of hardware models in C with pointers and complex data structures. IEEE Transactions of Very Large Scale Integration Systems (TVLSI), Vol. 9, Issue 6, Decemeber 2001, pp. 743-756.

[20] Weinberg, N. and Nagle, D. 1998. Dynamic elimination of pointer-expressions. Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 142-147.

[21] Xilinx Inc. MicroBlaze, 2008. http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm.

[22] Xilinx Inc. Virtex IV FX devices, 2008. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.