A Traversal Cache Framework for FPGA Acceleration of Pointer Data Structures: A Case Study on Barnes-Hut N-body Simulation

James Coole, John Wernsing, Greg Stitt
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL
{jcoole, wernsing, gstitt}@ufl.edu

***Abstract*—Numerous studies have shown that field-programmable gate arrays (FPGAs) often achieve large speedups compared to microprocessors. However, one significant limitation of FPGAs that has prevented their use on important applications is the requirement for regular memory access patterns. Traversal caches were previously introduced to improve the performance of FPGA implementations of algorithms with irregular memory access patterns, especially those traversing pointer-based data structures. However, a significant limitation of previous traversal caches is that speedup was limited to traversals repeated frequently over time, thus preventing speedup for algorithms without repetition, even if the similarity between traversals was large. This paper presents a new framework that extends traversal caches to enable performance improvements in such cases and provides additional improvements through reduced memory accesses and parallel processing of multiple traversals. Most importantly, we show that, for algorithms with highly similar traversals, the traversal cache framework achieves approximately linear kernel speedup with additional area, thus eliminating the memory bandwidth bottleneck commonly associated with FPGAs. We evaluate the framework using a Barnes-Hut n-body simulation case study, showing application speedups ranging from 12x to 13.5x on a Virtex4 LX100 with projected speedups as high as 40x on today's largest FPGAs.**

Keywords-FPGA, traversal cache, pointers, speedup

I. INTRODUCTION

Much previous work has shown that field-programmable gate arrays (FPGAs) can achieve order of magnitude speedups compared to microprocessors for many important embedded and scientific computing applications [3][4][9]. However, one limitation of FPGAs that has prevented widespread usage is the requirement for regular memory access patterns (i.e., sequential streaming of data from memory) due to the heavily-pipelined circuits common in FPGA implementations. Applications with irregular memory access patterns, such as pointer-based data structure traversals, achieve much lower memory bandwidth due to increased row-address-strobe (RAS) operations and memory indirection caused by pointer accesses. Consequently, this lower memory bandwidth results in many pipeline stalls, which waste large amounts of parallelism, often resulting in limited or even no FPGA speedup.

Previous work [12] partially addressed the inefficiency of irregular memory access patterns on FPGAs by using a traversal cache framework that identified repeated traversals of pointer-based data structures (which had irregular access patterns), reordered the repeated traversals into a sequential sequence of data, and then stored the reordered data into a traversal cache that efficiently streamed data to the FPGA. Although that framework improved FPGA performance, speedup was limited to repeated data structure traversals, which are not common in many important applications.

In this paper, we present an improved traversal cache framework that eliminates many limitations of previous work, increasing the number of applications amenable to FPGA speedup. The improved framework is motivated by the observation that although many applications do not have repeated traversals, many applications do exhibit a large percentage of similarity between traversals. To exploit this similarity, the improved traversal cache framework stores multiple traversals, represented by an application-specialized data structure, in a traversal cache. Unlike previous work that only improved performance of repeated traversals, the improved framework improves performance of similar traversals by enabling multiple traversals to be performed in parallel based on the amount of similarity. Furthermore, if an application does repeat traversals, the improved framework maintains all of the advantages of the previous approach.

Figure 1 provides an overview of traversal caches for an application that traverses a tree-based data structure. The previous approach, shown in Figure 1(a), does not achieve speedup due to a lack of repeated traversals (i.e., a 100% traversal cache miss rate). For any non-repeated traversal, the previous approach requires the microprocessor to compute the new traversal and update the traversal cache with the data included, often creating a bottleneck. On a cache miss, unlike previous approaches, the presented framework transfers a specialized representation of the data structure to the traversal cache, which enables the FPGA (as opposed to the microprocessor) to generate multiple traversals in parallel based on the amount of similarity. By transferring all the data necessary to compute any traversal, the improved approach effectively improves memory bandwidth by reusing data across multiple datapaths, thus enabling additional datapath replication (e.g., increased loop unrolling). In fact, the experimental results show that *the memory bandwidth bottleneck is almost completely eliminated for highly-similar traversals*, resulting in kernel speedup that increases approximately linearly with area. To deal with differences in traversals, the framework uses a

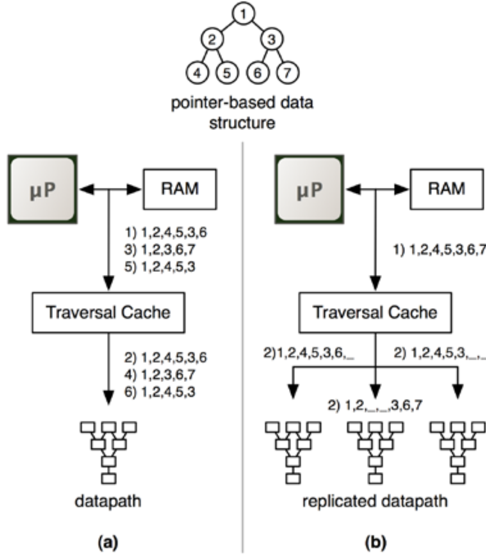

Figure 1. Traversal cache overview. (a) Previous approaches often lack speedup due to unrepeated traversals. (b) The improved traversal cache framework exploits similarity between traversals, enabling datapath replication by reducing memory accesses and delivering appropriate membership data to each datapath.

membership function to exclude inappropriate data from reaching each corresponding datapath. For the example in Figure 1(a), the previous approach would have required 16 memory accesses and 16 traversal cache accesses, limiting the FPGA to process a single traversal at a time. The improved approach, shown in Figure 1(b), requires only 7 memory accesses and 7 traversal cache accesses, which enables the FPGA to process 3 parallel traversals, resulting in a speedup of approximately 3x.

Note that the framework is not intended to automatically speedup an application; instead, a designer must provide the framework with necessary application-specific information. In this paper, we evaluate the framework using a Barnes-Hut n-body simulation [6] and discuss how the framework could be used for other applications. Barnes-Hut is a motivational example for the traversal cache framework because for n-body simulations on FPGAs, Barnes-Hut is typically avoided due to irregular memory access patterns resulting from the use of a quad-tree. Instead, FPGA implementations often use a straightforward $O(N^2)$ implementation that does not require a tree data structure. Although this straightforward implementation enables highly parallel FPGA execution, software execution of $O(N\log N)$ versions of Barnes-Hut eliminates FPGA speedup for large problem sizes. The traversal cache framework enables highly parallel FPGA execution of the $O(N\log N)$ versions of the Barnes-Hut algorithm, with application speedup ranging from 12x to 13.5x on a Virtex4 LX100 and projected speedups as high as 40x on today's largest FPGAs. For a simulation involving 1e5 particles, previous approaches using the $O(N^2)$ algorithm would have to achieve a 500x speedup to match Barnes-Hut implemented in software, or a 6800x speedup to match the performance of our approach.

The paper is formatted as follows. Section II discusses previous work. Section III describes the traversal cache framework. Section IV presents experimental results for an implementation of the Barnes-Hut n-body algorithm using the framework.

II. Previous Work

In addition to the prior traversal cache work discussed in the introduction, there are several other areas of research related to traversal caches. Smart buffers [8] are specialized FPGA data caches, similar to traversal caches, which store reused data for sliding window operations. Smart buffers prevent reused data from being read multiple times from memory, which improves bandwidth and increases datapath replication. The traversal cache framework discussed in this paper is a superset of the functionality provided by smart buffers, also supporting pointer-based data structures and irregular memory access patterns.

Smart memory engines [5] use FPGAs to dynamically reorganize main memory to improve locality of pointer-based data structures. Traversal caches also reorganize data, but do so locally in the traversal cache as opposed to modifying main memory, which avoids alias limitations.

Numerous high-level synthesis techniques have focused on alias analysis to enable optimization of pointer-based code [11]. Previous compiler and architecture studies have focused on improving locality of pointer-based data structures with data alignment and numerous other optimizations [1][2][7][10]. These techniques are complementary to traversal caches.

The Impulse memory controller [13] improves memory performance of irregular patterns by mapping irregular virtual patterns into regular physical patterns. Traversal caches perform the same optimization, but do not require a specific memory controller or operating system support.

III. Traversal Cache Framework

A. System Architecture

The system architecture consists of a microprocessor with access to an FPGA accelerator, as illustrated in Figure 2. The FPGA implements pipelined datapaths that accelerate computation-intensive kernels of an application. The microprocessor and FPGA can be tightly coupled, resident on the same board or device, or loosely coupled, as over a system bus or network. Separate input and output memories

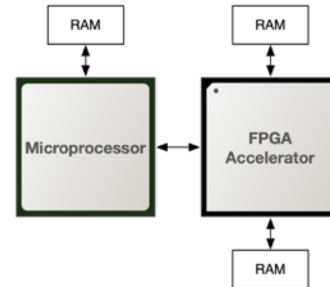

Figure 2. System architecture used by the traversal cache framework.

are assumed for streaming data through the pipelines, accessible by both the FPGA and microprocessor. Although the framework can support any system meeting these requirements, we currently implement the framework using a Nallatech H101-PCIXM FPGA accelerator attached over PCI-X to a 3.2 GHz Intel Xeon.

B. Software

The microprocessor executes the software portion of the application until it encounters an FPGA-implemented region. As in previous work [12], software populates the traversal cache before the FPGA can be used. To allow the traversal cache to process multiple traversals simultaneously, the software portion of the framework uses designer-specified code to serialize the entire data structure, in the order it will be traversed in common for all traversals. This is, in a sense, a generalization of the previous approach which provided the data for a single traversal at a time, in the order of the traversal. Figure 1(b) gives an example of a serialization of three similar traversals from Figure 1(a). The serialization logic is currently manually specified because the serialized encoding must be specialized for different data structures and traversal orders. Automation is a topic for future work.

Software also places in a section of the traversal cache any data outside the data structure necessary to compute each traversal, which we refer to as *traversal inputs.* For example, in an implementation of the Barnes-Hut n-body algorithm (discussed in Section IV), the forces on a number of particles are computed by traversing a tree, and those particles are the traversal inputs. The performance of the accelerator is maximized by the traversal cache when the traversal inputs are ordered to maximize the similarity between the traversals for adjacent inputs. The current framework also relies on the software to maintain the traversal cache's consistency by updating the cache's contents to reflect changes made to the data structure. In a tightly coupled system architecture, however, the framework could potentially be modified to support cache coherence through snarfing or snooping.

Once the traversal cache is populated, software notifies the accelerator to begin processing data from the cache and is later notified by the accelerator when it is finished. Depending on the application, the processor might be able to do other work while waiting on the accelerator. For example, in Barnes-Hut, the output of the accelerator can be streamed to the microprocessor as it is computed, allowing preparation for the next time step's calculations to occur in parallel.

C. FPGA Framework

The FPGA portion of the traversal cache framework is shown in Figure 3. The *traversal generator* computes the traversals for batches of traversal inputs at a time until all inputs have been used. For each batch, the number of memory accesses is minimized by exploring the data structure in a single pass, covering as little of the data structure as is necessary. The traversals for each input within the batch are computed in parallel and the elements included in each input's traversal are streamed out from the generator as soon as their membership in each traversal is computed.

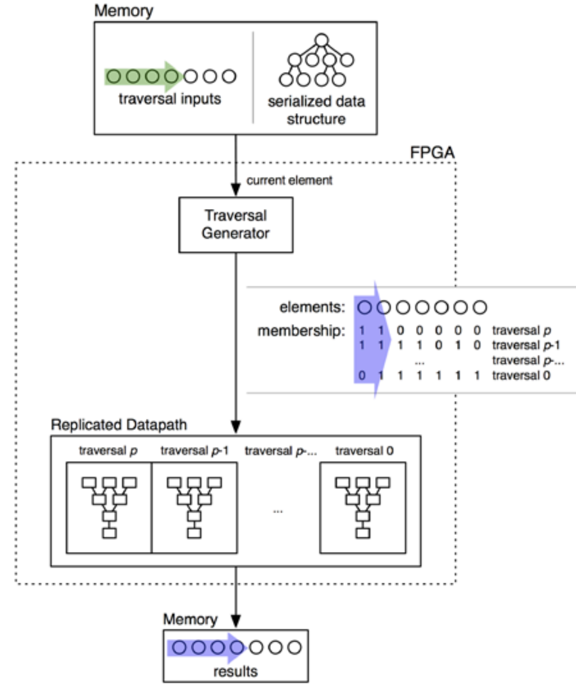

Figure 3. FPGA portion of the traversal cache framework.

The accelerator's datapath can be replicated *p* times, with each replication handling a separate traversal stream produced by the traversal generator. In addition to a single common bus containing the current data structure element, e.g., the current Barnes-Hut tree node, *data_valid* and *wait* signals are provided for each datapath, informing the datapath when elements included in that traversal are available and allowing the datapaths to stall the generator, respectively. If necessary, any intermediate datapath products that might be computed in the generator kernels can also be brought out for the datapaths.

The structure of the traversal generator is shown in Figure 4. Generator logic is to some extent specific to the data structure and algorithm, and therefore must currently be designer-specified. In all cases, however, the generator consists of a *generator kernel* specific to the application replicated for each of the *p* parallel traversals. Each generator kernel decides independently whether the current data structure element is included in the traversal for its input and

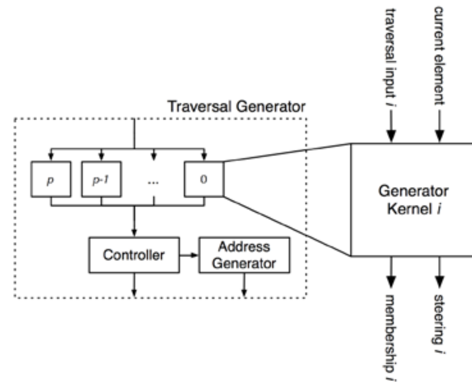

Figure 4. Traversal Generator.

guides further exploration of the data structure, for example by skipping nodes in a tree that it knows will not be included. Inclusion information is passed out of the generator as *data_valid* signals to the corresponding datapath. For example, in our Barnes-Hut implementation, the *data_valid* signal is generated by logic testing whether to accumulate the force due to the current tree node. In this case, since generating this signal also involves calculating data needed by the force calculation performed in the datapaths (the force direction among others), those signals are also brought out from the generator kernel for use by its datapath.

For generator kernels that guide exploration of the data structure, the generator is also responsible for exploring as little of the data structure as possible, while still meeting the requirements of all its kernels. In the Barnes-Hut implementation, for example, if all but one kernel can skip a portion of the tree, the generator must still explore the section for the objecting kernel while blocking that data from reaching the other kernels. Because the kernel logic is itself pipelined, the generator must continue to fetch elements from the traversal cache, predicting that they will be used, to avoid stalling excessively. Thus, kernels can continue to receive elements that they have chosen to ignore for as long as the pipeline latency. Kernel outputs for these elements are invalid and must be ignored by the generator logic. For the same reason, the kernels themselves must either be stateless (i.e. no dependence on past data) or must be capable of reverting their state when it's determined that previously received elements were invalid.

In the simplest case, the number of traversals handled by the framework in parallel is limited to p, the number of replicated datapath-generator kernel pairs, which is limited by the resources available on the FPGA. The number of parallel traversals can be increased without increasing datapath replication by breaking them up into p-sized subsets: for each entity in the data structure, traversals within each set are computed in parallel and sets are processed sequentially by swapping out the traversal inputs used by the generator kernels. The amount of datapath replication p and the number of p-sized sets of traversals computed together s are the framework's architectural parameters. The total number of traversals considered in a batch is given by $p \times s$.

D. Limitations

The maximum speedup using our approach is achieved for algorithms with independent traversals in the same order, e.g. depth-first traversal, across a data structure. In the case that an algorithm's traversals are dependent on previous traversals, the framework cannot process traversals in parallel (p=s=1). For algorithms whose traversals occur in different orders across a data structure, the framework is equivalent to the previous approach [12], requiring software intervention between processing traversals.

The potential for speedup using this approach increases with the similarity between successive traversals. Because not all applications exhibit high similarity between traversals, the framework does not always improve performance. Low similarity between traversals results in disagreement among generator kernels about which regions of the data structure can be ignored. Since the generator must satisfy the needs of all the kernels to maintain correctness, it must include regions needed by any single kernel, stalling any other kernels (and datapaths) that don't need those elements, reducing parallelism. Since element accesses are grouped for traversals computed in parallel, it can also be shown that the total number of accesses is minimized when similarity is maximized. However, because efforts to maximize the similarity between traversals would also benefit pure software implementations due to caches on main memory, implementations using the framework might benefit from existing work along those lines.

In this paper we assume the data structure fits in the accelerator's memory. In situations where the size of memory is a limitation, software could load only a part of the data structure into memory, updating the cache with the missing portions when they are reached during each traversal. Though this would limit the system performance, implementations not using a traversal cache would also be limited by the size of available memory, and possibly sooner since the ordering assumed by the framework allows the data structure's serialized form to often be smaller than the original form in memory. More detailed analysis of the effects of cache size is left as future work.

IV. Case Study: N-Body Simulation

To evaluate the traversal cache framework, we performed a Barnes-Hut n-body simulation case study. The critical step in n-body simulation involves computing the resultant physical force (e.g. gravitational or electrostatic forces) on each of N bodies due to the other bodies. The Barnes-Hut algorithm [6] approximates the solution by recursively subdividing space into a quad-tree or oct-tree, with internal nodes representing an average of the leaf bodies (e.g. center of mass or electric multipoles). For concentrated collections of distant objects, set by a threshold ratio *theta*, the force is computed due to these average nodes instead of individual bodies. Depending on the value chosen for *theta*, Barnes-Hut has a complexity ranging from $O(N \log N)$ to $O(N^2)$, with $O(N \log N)$ for *theta*=0.5 being common in practice [6].

Our implementation of Barnes-Hut computes classical gravitational forces in two dimensions. The quad-tree traversal logic was implemented as a generator supporting generic depth-first search over an n-ary tree, with traversal kernels computing traversal membership and steering the search according to the algorithm and a programmable *theta*. Similarity between traversals is increased by loosely ordering the bodies by spatial locality in the universe, which is available inexpensively as the order of the leaves in the fully constructed quad-tree. Our software implementations also use this ordering, which we found provides a speedup of ~30% due to cache issues, as mentioned in Section III.D. Our hardware implementation constructs the next time step's quad-tree in parallel with the accelerator by streaming in accelerator outputs as they become available.

A. Experimental Setup

To evaluate the traversal cache, we implemented the framework in VHDL, including custom generator kernels

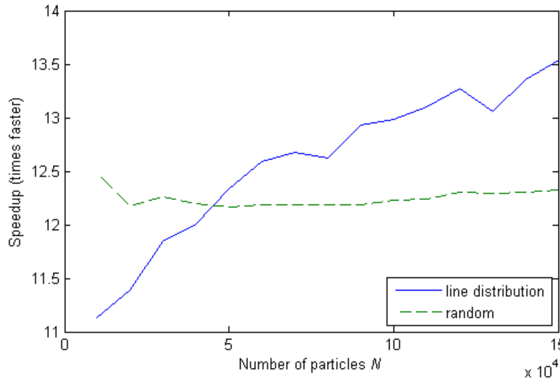

Figure 5. N-body application speedup achieved by the traversal cache framework on a Virtex 4 LX100 compared to a 3.2 GHz Xeon for various numbers of particles and representative distributions.

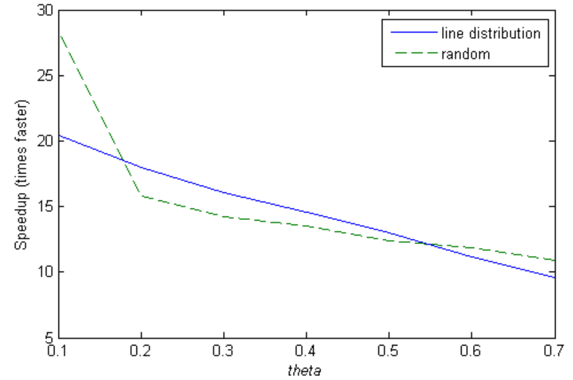

Figure 6. For lower values of theta (i.e., less approximation in Barnes-Hut), the traversal cache framework achieves larger application speedup. Higher values of *theta* than shown are not common in practice.

and datapaths for Barnes-Hut, using generics for the number of traversals computed in parallel p and the number of p-sized subsets computed sequentially s in each pass. The framework was configured to use a single SDRAM input memory. Cycle counts were extracted from an HDL test bench that used a memory model of the SDRAM available on the Nallatech H101-PCIXM, accessed through Nallatech's SDRAM controller. Timing was measured and verified using a hardware implementation on the Nallatech H101-PCIXM FPGA accelerator card.

The HDL simulation and timing data were combined to create a model of the framework's behavior for arbitrary input data. From this model, we developed a C-based simulator capable of giving accurate timing for real input data. Because synthesizing each test case would have been very time consuming, this C-based simulator enabled rapid exploration of the framework's design space, and also allowed considering configurations that wouldn't fit on the H101's Xilinx Virtex 4 LX100 FPGA.

B. Performance Results

Our traversal cache Barnes-Hut implementation was compared to software running on a 3.2GHz Xeon. The framework was tested with the maximum parallelism supported by the LX100 with no sequencing within a batch (s=1, p=25). Sequencing is discussed in the next section. Comparisons were made for two data sets representing extremes with respect to the similarity of traversals: bodies distributed randomly in space (minimum similarity of 83% for *theta*=0.5) and bodies uniformly distributed in a straight line (maximum similarity of 90% for *theta*=0.5). Multiple problem sizes N and values of *theta* were also tested. The results are presented in Figures 5 and 6.

Figure 5 shows the application speedup of the framework relative to software for different problem sizes and *theta*=0.5. For the problem sizes tested, a speedup of between 11x and 13.5x was achieved, depending on the data set. Execution times were lower for the high-similarity (line) data set for both hardware and software. The increasing speedup of the high-similarity data set is due to the task-level parallelism made possible by the framework; since the accelerator's run time was smaller for this data set than the time required to construct the next quad-tree, incremental increases in the accelerator's run time were hidden, resulting in a speedup increasing with N up to a plateau. The variability in the performance curves is primarily due to irregular growth in the size of the quad-tree.

The effect of the algorithm's precision parameter *theta* is shown in Figure 6 for a simulation with 1e5 bodies. The framework achieves higher application speedup for smaller values of *theta*, where the algorithm's complexity approaches $O(N^2)$. Traversals include more elements for these instances of the problem, resulting in a greater reduction in total memory accesses due to the traversal cache and, consequently, a greater speedup over software. When compared against experimental results in [12] the data also suggest a large speedup over previous traversal caches for the $O(N^2)$ case. Note that comparing speedups of different *theta* values is different than comparing execution times. As *theta* increases, the complexity and execution time of the algorithm decreases, which contributes to the decreasing speedup. Furthermore, previous n-body implementations are typically unable to achieve speedup for *theta* > 0.

C. Effect of Framework Parameters

The traversal cache framework can be configured for an application through the parameters p and s, as discussed in Section III.C. In this section we explore the effect of these parameters for our traversal cache implementation of Barnes-Hut. Simulation data is provided in Figures 7 and 8 below for a system of 1e6 bodies with *theta*=0.5. In order to focus on the behavior of the framework itself, the data in these figures deals only with execution times and speedups for the force calculation kernel, referred to as the *kernel speedup*, which the portion of Barnes-Hut implemented on the accelerator. The results for the Barnes-Hut application as a whole are discussed after the figures.

Figure 7 demonstrates that the kernel speedup provided by the framework increases with p by an amount determined by the similarity between traversals and grows nearly linearly for the high-similarity (line distribution) case. The traversal cache is able to achieve this speedup *without increased demands on physical memory bandwidth by increasing effective memory bandwidth*, requiring only a single access for any single data structure element within a batch of traversals. Since p is limited by the size of the device, labels were added showing the amount of unrolling achievable on some common FPGAs.

Figure 8 shows the effect of s, the number of p-sized groups of traversals computed in sequence within a batch,

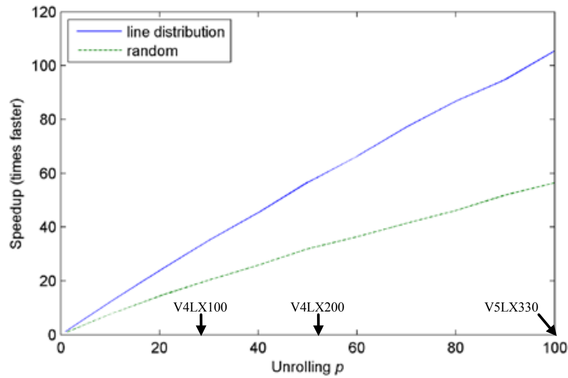

Figure 7. N-body kernel speedup obtained by the traversal cache framework for different amounts of datapath replication *p*. *Note that for high similarity, speedup increases almost linearly due to the framework exploiting similarity between traversals to eliminate the memory bandwidth bottleneck*.

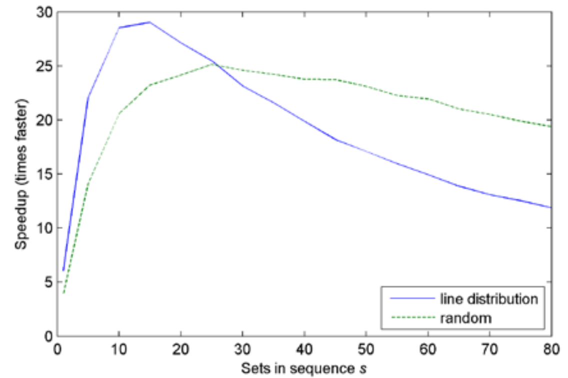

Figure 8. N-body kernel speedup obtained by the traversal cache framework with *p*=5 for different amounts of sequential processing *s*.

for Barnes-Hut force calculation implemented with 5 datapaths (*p*=5). The simulation demonstrates that additional kernel speedup can be obtained after maximum unrolling by increasing *s*, up to a maximum. This maximum speedup is also shown to be limited by the similarity between traversals, with higher similarities allowing a greater speedup for lower values of *s*. Since the optimum amount of sequential processing *s* depends on the amount of similarity, which, as is the case for Barnes-Hut, is likely not constant or known *a priori*, these results suggest that traversal caches might benefit from an adaptive implementation that adjusts *s* according to an observed or predicted amount of similarity.

The full application speedup is limited by the execution time for the software parts of the application. For our implementation, application speedup was limited to 13-40x, due to the time required to construct the quad-tree. Since the framework requires a serialized form of the data structure, however, speedup for the application is ultimately limited by the time required to generate and send this serialization. For our implementation, this step alone would have limited application speedup to 30-110x.

V. CONCLUSIONS

In this paper, we introduced a traversal cache framework that enables efficient FPGA execution of some applications with irregular memory access patterns. As opposed to previous work that was limited to repeated traversals, the framework in this paper extends traversal caches to handle numerous similar traversals in parallel. For an n-body case study, the framework was shown to achieve speedups ranging from 12x to 13.5x compared to software. More importantly, by exploiting similarity between traversals, the framework can eliminate the common memory bandwidth bottleneck, leading to linear increases in speedup with additional area. Future work includes automating many of the current designer specified portions of the framework.

REFERENCES

[1] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious dataplacement," in ASPLOS-VIII: Proceedings of the 8th international conference on architectural support for programming languages and operating systems. 1998, pp. 139–149.

[2] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Making pointer-based data structures cache conscious," Computer, vol. 33, no. 12, pp. 67–74, 2000.

[3] S. Craven and P. Athanas, "Examining the viability of FPGA supercomputing," EURASIP J. Embedded Syst., vol. 2007, no. 1, pp. 13–20, 2007.

[4] A. DeHon, "The density advantage of configurable computing," Computer, vol. 33, no. 4, pp. 41–49, 2000.

[5] P. Diniz and J. Park, "Data search and reorganization using FPGAs: application to spatial pointer-based data structures," in FCCM '03: Proceedings of the 11th annual IEEE symposium on field-programmable custom computing machines, 2003, pp. 207–217.

[6] A. Y. Grama, V. Kumar, and A. Sameh, "Scalable parallel formulations of the barnes-hut method for n-body simulations," in Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on supercomputing. 1994, pp. 439–448.

[7] P. Grun, N. Dutt, and A. Nicolau, "Access pattern based local memory customization for low power embedded systems," in DATE '01: Proceedings of the conference on design, automation and test in Europe. 2001, pp. 778–784.

[8] Z. Guo, B. Buyukkurt, and W. Najjar, "Input data reuse in compiling window operations onto reconfigurable hardware," in LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on languages, compilers, and tools for embedded systems. 2004, pp. 249–256.

[9] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on field programmable gate arrays. 2004, pp. 162–170.

[10] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," ACM Trans. Des. Autom. Electron. Syst., vol. 6, no. 2, pp. 149–206, 2001.

[11] L. Semeria, K. Sato, and G. De Micheli, "Synthesis of hardware models in c with pointers and complex data structures," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 6, pp. 743–756, Dec 2001.

[12] G. Stitt, G. Chaudhari, and J. Coole, "Traversal caches: a first step towards FPGA acceleration of pointer-based data structures," in CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on hardware/doftware codesign and system synthesis. 2008, pp. 61–66.

[13] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The impulse memory controller," IEEE Transactions on Computers, vol. 50, no. 11, pp. 1117–1132, 2001.