Spatial Memory Trace Prediction

Nadeen Yassir Gebara[a,b], Paolo Ienne[b], and Kermin Fleming[a]
[a]Intel Corporation, Hudson, MA, USA
[b]EPFL, Lausanne, Switzerland

Abstract

Heterogeneous FPGA systems among other spatial architectures are gaining popularity as candidates for accelerating Data Center and HPC applications. As the number of FLOPs supported on FPGAs increases, the rate at which data can be provided to computational units becomes the major bottleneck determining overall system performance. This heightens the importance of having a quick way to observe and understand the memory access patterns of applications on FPGAs prior to investing long periods of time and effort in writing RTL, especially for applications that will not observe any performance improvements due to bandwidth limitations. Furthermore, since the memory subsystem within FPGAs is programmable, a clear visibility of the spatial memory access patterns of applications prior to design implementation can guide FPGA programmers on the best way to configure and use the memory subsystem in an application specific manner while providing an estimate of bandwidth requirements. In this paper, we present a novel approach for obtaining memory traces of workloads targeting FPGA systems to provide insight on the memory access patterns of applications and guide FPGA programmers. Our algorithm displays an average prediction accuracy of 97.45% for read memory accesses when RTL optimization opportunities are fully exploited, and an accuracy of 90.71% for memory write accesses across the investigated benchmarks. By providing a fast and efficient way to obtain spatial memory reference traces of applications, our work not only guides FPGA programmers by providing more insight into the memory profiles of workload regions amenable to FPGA implementations, but also helps leverage the shortage of memory traces available for performing architectural studies and enhancing FPGA memory systems.

1 Introduction

Post-Dennard scaling, homogeneous processor systems can no longer efficiently meet the high computational demands of scaling and High Performance Computing (HPC) workloads at reasonable costs. Previously, the most common approach for keeping up with the increasing computational requirements of applications was increasing the number of processing cores. However, this approach is now limited by significant power requirements. Without such scaling, homogeneous processor systems can't efficiently drive the projected memory bandwidths of emerging memory technologies such as High Bandwidth Memory (HBM) which can mitigate the processing memory gap. FPGAs and Spatial Architectures are good candidates for providing emerging workloads with better performance per watt. However, determining which codes are amenable to acceleration on Spatial Architectures is still challenging especially for non-hardware experts.

The most accurate method to determine whether an application will perform well on an FPGA is to obtain the optimized hand-written Register Transfer Level (RTL) of the application, synthesize it and implement the obtained design on an FPGA, execute and measure the obtained performance. However, the process is not as straightforward since only a handful of applications have RTL implementations. The actual process involves writing the optimized RTL implementations first, and this requires a very good understanding of the application in terms of computation and memory access patterns, a clear assessment of the target FPGA platform to determine the best way to utilize the provided resources and schedule events, a certain level of hardware expertise to write optimal RTL implementations of the design, and a great deal of time.

An alternate approach that is more accessible to a broader community of software programmers is using available High Level Synthesis Tools (HLS) to translate the high level language (HLL) into RTL and synthesize the required hardware. However, this approach is limited since an understanding of the application and hardware is still required to take advantage of the optimization directives and pragmas provided by HLS tools. Also, Iterative Design Space Exploration (DSE) approaches are often used to ensure Quality of Synthesis of Results (QoR) **[1]**, and this can be time consuming considering the time required for hardware synthesis.

Confronted with these challenges, we seek a better way of determining the applicability of FPGAs. We observe that the performance of FPGAs will be limited by available memory bandwidth despite the fact that they can supply almost any level of computational requirements **[2]**. Since memory bandwidth is the bottleneck limiting the overall performance of applications on FPGAs, a visibility into the memory access patterns of an application on a target FPGA coupled with information on the bandwidth supported by that FPGA can be enough to bound the application's performance on the target system.

To circumvent the problems described earlier with obtaining optimized implementations of programs on FPGAs and then running the programs to observe their memory access patterns, we leverage the wide availability of applications targeting x86 platforms and design an algorithm that predicts the memory accesses of optimally written FPGA implementations of applications from their x86 binaries.

Our proposed algorithm applies a set of FPGA inspired filters to transform the initial traces of the benchmarks obtained on an x86 machine into their spatial equivalent traces. The filters exploit temporal locality across and within memory access operations in the original x86 binary. We further implement the designed algorithm within the Pin Dynamic Binary Instrumentation Framework **[3]**, which we call Spatial Memory Trace Predictor. Spatial Memory Trace Predictor produces a spatial memory reference trace of the application and logs it to a file. We define a memory reference trace as set of accesses to certain memory addresses in a program correct order. To validate the performance of the proposed algorithm as an FPGA memory trace predictor, we compare the memory trace predictions obtained using our tool for various CORAL **[4]** and PARSEC **[5]** HPC Benchmarks against those obtained when their equivalent Intel Proprietary RTL implementations were run on an FPGA simulator with a fixed storage capacity. We show that our proposed algorithm achieves an average experimental prediction accuracy of 97.45% for read memory accesses when compared against optimal RTL implementations, and an average prediction accuracy of 90.71% for write memory accesses.

We believe that the traces generated by our algorithm can be used for the following purposes:

1. Determine whether applications are amenable to FPGA acceleration or will limited by the supported bandwidth.
2. Provide insight on the memory profiles of the optimized implementations of various benchmarks on FPGAs which can be used to bound the achievable performance.
3. Guide FPGA programmers on how to make use of the configurable memory systems of FPGAs when implementing their designs especially for workloads with irregular memory access patterns.
4. Drive future research in the scope of enhancing the architectures of FPGA memory systems.

The rest of the paper is organized as follows. In Section 2 we present the basic differences between Spatial and Von-Neumann Architectures along with a brief motivational example to clarify the problem we are targeting. We then describe our proposed algorithm in Section 3. In Section 4, we present our experimental method and results, and evaluate the performance of our algorithm in predicting spatial memory traces.

2 Spatial Architectures Versus Von-Neumann Machines

FPGAs are becoming more widely considered as accelerators for many applications as their fine-grained Spatial Architecture appears capable of delivering higher performance than the rigid Von-Neumann Architecture through application-specific configuration of memory and computational resources. Control logic in FPGAs is typically implemented in the form of finite state machines (FSMs) allowing efficient mapping of algorithms onto available logic blocks.This permits the execution of multiple independent operations on data within the same pipeline stage as well as maintaining a producer consumer relationship across successive pipeline stages. By maintaining a producer consumer relationship across pipeline stages, the need for performing memory accesses between pipeline stages is alleviated and pipeline parallelism can be further exploited.

Von-Neumann Architectures on the other hand comprise of three distinct building blocks: a central processing unit, memory, and input/output devices connected together by a system bus. They have a fixed data path pipeline that dictates instruction execution. Instructions are first read and decoded, and operations are then performed on data according to the instructions. Unlike FPGAs which can stream data and have configurable embedded memory blocks, Von-Neumann architectures rely on registers for data transfer. Since register files are limited in size due to performance and architectural reasons, Von-Neumann machines are not as flexible as Spatial Architectures and can't expand to accommodate working sets resulting in the need for accessing memory more frequently to obtain data. Teubner et al. show how this flexibility in accommodating working sets provided by Spatial Architectures can be exploited to significantly improve the performance of the frequent item calculation operation **[6]** over software based implementations.

In addition to the structural differences between the architectures, their programming models also differ. Programs running on Von-Neumann machines typically involve function and library calls to provide a programmer friendly environment. Changes in control flow on Von-Neumann machines reflect as stack accesses to keep track of the value of the current state of a program. This is commonly referred to as stack discipline and is crucial for maintaining correct program execution on Von-Neumann Architectures. Since control is implemented as finite state machines within the programmable logic of Spatial Architectures, FPGAs do not need to perform any memory accesses due to changes in control flow.

2.1 Motivational Example

The code snippet displayed in **Figure 2** shows a simple binary search algorithm implemented in C. The input array is sorted with value i stored at index position i for i $\subseteq \{0, 14\}$, and gets queried successively for the values $\{10\}$, $\{9\}$, and $\{2\}$. When querying those terms, the sequence of memory accesses shown in **Figure 2-(a)** is to be expected. However, upon instrumenting the code's binary

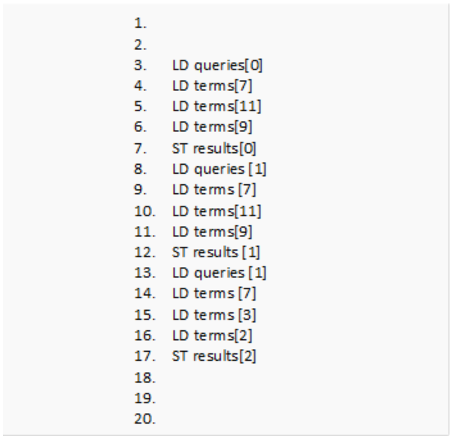
```c
void binary_search(int nqueries,int nterms,uint64_t *
    queries_array, uint64_t * input_array,uint_64t*
    result_array){
for (int index = 0; index < nqueries; index++)
 {
 uint64_t query, cur;
 int min=0, mid, max=nterms-1;
 query=queries_array[index];
 do{    mid = (min + max) / 2;
        cur = input_array[mid];
        if (cur < query) {
            min = mid+1;
        } else if (cur > query) {
            max = mid-1;
        } else {
            max = mid;
        }
   } while ((max> min) && (cur!=query));
  result_array[index] = max;
 }
}
void main() {
//Initializations
 int nqueries=3;  int nterms=15;
 uint64_t queries[nqueries]={10,9,2};
 uint64_t results[nqueries];
 uint64_t terms[nterms];
 for(int i=0;i<nterms;i++)
 terms[i]=i;

 //Function Call
 binary_search(nqueries,nterms,queries,terms,results);
}
```

Figure 1 Binary Search Algorithm: This figure displays the source code for a naive implementation of the Binary Search Algorithm in C.

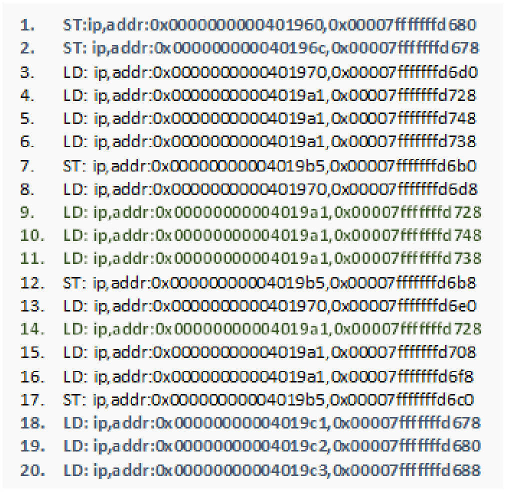
to display the sequence of memory accesses made during its execution on an x86 machine, the memory reference trace displayed in **Figure 2-(b)** was obtained. An excess of two store memory accesses and three load accesses can be observed when comparing the obtained trace with the anticipated sequence of memory accesses. Those excess accesses result from stack discipline associated with the Von-Neumann programming model. The first two memory accesses result from storing the values in the rbp and rbx registers on the stack because the x86 calling convention dictates that the called function is expected to preserve the values in those registers. The three additional load instructions at the end of the trace restore the values previously held in those registers from the stack and read the value of the program counter (PC) or instruction pointer (IP) stored by the calling routine to continue execution in program order and maintain program correctness.

Upon using the implemented Pin-Tool that relies on our proposed algorithm for predicting spatial memory reference traces on the same code's x86 binary, the sequence of accesses shown in **Figure 2-(c)** was obtained. The obtained trace is not only free of the overhead memory accesses native to the Von-Neumann programming model, but from memory accesses that would no longer be necessary on FPGAs if their Spatial Architectures were appropriately exploited by the programmer. A detailed inspection of the obtained trace shows that for the given algorithm, repeated accesses to the memory addresses corresponding to the terms {7}, {9}, and {11} are not expected on FPGAs. These terms correspond to the roots of the binary tree and are accessed for all searches. The algorithm implies



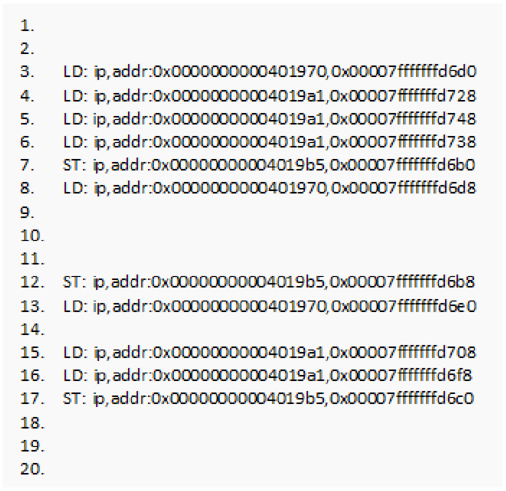
```
1.
2.
3.   LD queries[0]
4.   LD terms[7]
5.   LD terms[11]
6.   LD terms[9]
7.   ST results[0]
8.   LD queries [1]
9.   LD terms [7]
10.  LD terms[11]
11.  LD terms[9]
12.  ST results [1]
13.  LD queries [1]
14.  LD terms [7]
15.  LD terms [3]
16.  LD terms[2]
17.  ST results[2]
18.
19.
20.
```

(a) Expected Memory Trace from Inspection.



```
1.   ST:ip,addr:0x0000000000401960,0x00007fffffffd680
2.   ST: ip,addr:0x000000000040196c,0x00007fffffffd678
3.   LD: ip,addr:0x0000000000401970,0x00007fffffffd6d0
4.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd728
5.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd748
6.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd738
7.   ST: ip,addr:0x00000000004019b5,0x00007fffffffd6b0
8.   LD: ip,addr:0x0000000000401970,0x00007fffffffd6d8
9.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd728
10.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd748
11.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd738
12.  ST: ip,addr:0x00000000004019b5,0x00007fffffffd6b8
13.  LD: ip,addr:0x0000000000401970,0x00007fffffffd6e0
14.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd728
15.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd708
16.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd6f8
17.  ST: ip,addr:0x00000000004019b5,0x00007fffffffd6c0
18.  LD: ip,addr:0x00000000004019c1,0x00007fffffffd678
19.  LD: ip,addr:0x00000000004019c2,0x00007fffffffd680
20.  LD: ip,addr:0x00000000004019c3,0x00007fffffffd688
```

(b) Memory Trace Obtained on an x86 Machine.



```
1.
2.
3.   LD: ip,addr:0x0000000000401970,0x00007fffffffd6d0
4.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd728
5.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd748
6.   LD: ip,addr:0x00000000004019a1,0x00007fffffffd738
7.   ST: ip,addr:0x00000000004019b5,0x00007fffffffd6b0
8.   LD: ip,addr:0x0000000000401970,0x00007fffffffd6d8
9.
10.
11.
12.  ST: ip,addr:0x00000000004019b5,0x00007fffffffd6b8
13.  LD: ip,addr:0x0000000000401970,0x00007fffffffd6e0
14.
15.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd708
16.  LD: ip,addr:0x00000000004019a1,0x00007fffffffd6f8
17.  ST: ip,addr:0x00000000004019b5,0x00007fffffffd6c0
18.
19.
20.
```

(c) Memory Trace on a Spatial Machine.

Figure 2 Memory Reference Traces of Binary Search Algorithm: This figure displays overhead memory accesses due to the Von-Neumann Machine Model and contrasts the memory trace of Binary Search Algorithm on a Von-Neumann Machine with the trace on a Spatial Architecture.

the existence of an application-specific cache to hold the terms. This is consistent with the availability of embedded storage structures such as BRAMs in Spatial Architectures which can be exploited in an application-specific manner to improve the performance of applications by avoiding long latency off-chip memory accesses. By observing the obtained trace, similar insights can be obtained on FPGA memory related optimizations for other programs without requiring a detailed knowledge of the benchmark on the programmer's part.

3 Predicting Spatial Memory Traces from x86 Binaries

Our proposed method for obtaining spatial memory traces from x86 Binaries stems primarily from the observation that there is a great abundance of benchmarks targeting x86 Architectures. Relying on the observations summarized in **Table 1**, we further classify the set of memory accesses that aren't expected in spatial traces into the following categories:

C 1: Memory Accesses resulting from the instructions push, pop, call, branch and return.

C 2: Overhead Stack Accesses resulting from Storing and Restoring local variables when a change in the control flow of a program is encountered.

C 3: Read only Memory Accesses to constant values and loop counters.

C 4: Overhead Memory Accesses resulting from the fact that Spatial Architectures don't need to access memory to obtain values recently used or produced.

Our proposed algorithm passes each instruction through a set of filters to determine whether the instruction belongs to any of the listed categories and is to be excluded from the predicted spatial memory trace as can be seen in **Figure 3**. The first filter relies on Opcode-based filtering and is responsible for excluding all memory accesses that are push, pop, return, and branch instructions. The second filter relies on Program Counter (PC) based filtering and is responsible for detecting accesses belonging to either C-2 or C-3. The third filter uses Reuse-distance based filtering and is primarily responsible for detecting and eliminating accesses belonging to C-4.

3.1 Opcode-based Filtering

Since push, pop, return, call, and branch instructions have distinct op-codes within the x86 ISA, these accesses could be easily determined using routines provided by Pin's Inspection API. More specifically, the Pin framework provides functions that return instructions' op-codes as well as higher level abstractions that check whether a certain instruction belongs to a larger subcategory of instructions. Since there are various types of branch and call instructions with unique op-codes, the abstraction routines provided were used to detect and filter out all instructions that are

Figure 3 Spatial Memory Trace Prediction Algorithm.

```
Input: x86 Program Binary
Output: Text file with predicted Spatial Memory Trace
 1: for all Instructions ∈ Program do
      Read Memory Accesses
 2:   if Instruction is a Memory Read Access then
        Apply Opcode-based Filtering
 3:     if pop or return Instruction then
 4:       Eliminate from Trace {C-1}
 5:     end if
        Apply PC-based Filtering
 6:     if Current Mem_Address@PC == Previous Mem_Address@PC then
 7:       Eliminate current and previous access from Trace {C-2 or C-3}
 8:       Flag Address Accessed as stack/constant
 9:     end if
        Apply Reuse-based Filtering
10:     if Reuse(Mem_Address) <= Reuse Distance then
11:       Eliminate current access from Trace {C-2 or C-4}
12:       Set Address Accessed as stack/constant
13:     end if
14:     Log Access Meta-data
15:   end if
      Write Memory Accesses
16:   if Instruction is a Memory Write Access then
        Apply Opcode-based Filtering
17:     if push or branch or call Instruction then
18:       Eliminate current access from Trace {C-1}
19:     end if
20:     if Mem_Address flagged as stack/constant then
21:       Eliminate current access and previous write access to that address from Trace {C-2}
22:     end if
23:     Log Access Meta-data
24:   end if
25: end for
26: Write non-eliminated Accesses to Text File
```

either branches, calls, or return instructions. Push and pop instructions on the other hand don't have multiple types and op-codes within the x86 ISA. Since Pin provides no direct routines to identify such instructions, we used Pin's op-code inspection function to inspect the op-codes of instructions and match them against the those of push and pop instructions. Upon obtaining a match, the memory accesses are filtered out of the predicted trace since push and pop memory accesses shouldn't exist in spatial memory traces.

3.2 PC-based Filtering

Though the Pin inspection API provides routines that identify stack accesses, not all stack accesses are necessarily overhead accesses. As a result, we take a different approach and use PC-based filtering to detect overhead stack memory accesses as well as memory accesses belonging to C-3.

This technique checks whether consecutive read memory accesses are made to the same memory address at a particular PC. Since FPGAs are great at exploiting loop parallelism, compute intensive applications with multiple iterations are often accelerated using FPGAs. This kind of reuse is common across loop iterations, and such values would be cached in registers within FPGAs. Similarly, assuming a function call exists within the kernel, values

Table 1 Implications of Machine Architectures on their Memory Accesses.

Von-Neumann Architectures	Spatial Architectures	Implications on Memory Accesses
Control Driven → Stack Discipline required to make sure program state and order is maintained.	Data Driven → No Stack Discipline is required.	***I1.*** Von Neumann Architectures have overhead Memory Accesses resulting from stack discipline upon entering and exiting the function of interest as well as storing and restoring local variables when a change in the control flow of a program is encountered.
$LimitedRegisterFile$ → Frequent memory accesses required for reading/writing data.	Presence of embedded storage such as SRAMs and configurable channels between memory blocks and processing units → Less Frequent Memory Accesses Required.	***I2.*** Reads of constant values and loop iterators do not appear in FPGA based memory Traces. ***I3.*** Spatial Architectures don't need to access memory to obtain values recently used or produced.

will be stored and restored from the same stack address at a fixed PC. Therefore, by checking for repeated memory references at fixed PCs, overhead load operations resulting from reads of constant values and stack accesses can be detected and appropriately excluded from the generated spatial trace. To correctly detect overhead write memory accesses to the stack, we further flag the memory addresses suspected to be stack/constant reads. When a write access is then performed to a flagged address, the write access is filtered out as belonging to C-2.

3.3 Reuse-based Filtering

Reuse-distance is a common terminology in the field of compiler optimization and a common metric for predicting cache performance. Within those contexts, reuse distance is defined as the number of memory accesses to unique addresses between two successive accesses to the same address. We take a different approach and define reuse distance as the number of unique ***read*** memory accesses between two successive accesses to the same memory address. We introduce this metric to mimic the ability of Spatial Architectures to store data locally and feed data directly between processing units. Since we assume limited storage throughout the scope of this work, we expected the reuse-distance to be representative of the amount of embedded memory within the FPGA and their ability to support application specific memory hierarchies **[7]**. For example, when targeting an FPGA with an embedded memory capacity of 4 Kbytes to perform single precision computations, the reuse distance is set to 1024 since effectively 1024 values can be held within the FPGA. To verify our hypothesis, a sweep of various reuse distances was made to predict the memory accesses performed by binary search since the theoretical bound on the number of accesses obtained on FPGAs was known. As expected, the number of predicted memory accesses converged to the bound when the reuse distance became closer to the embedded storage capacity. Also. since the value of the reuse distance is typically larger than the number of load memory accesses that happen between stack accesses to store/restore register values when a change in program flow is encountered, the reuse distance additionally helps in detecting and filtering out overhead stack read accesses belonging to C-2 within a loop iteration.

4 Experimental Results and Evaluation

4.1 Experimental Setup

We implemented our proposed algorithm for obtaining spatial memory traces in a Pin-Tool. To evaluate the performance of our approach, we used a collection of well known CORAL, PARSEC, and other HPC benchmarks summarized in **Table 2**. We compiled the benchmarks with O2 optimization using g++ version 4.8.2 compiler to obtain x86 binaries on an Intel (R) Xeon(R) CPU. We opted for an optimization level of O2 since a lower optimization level would unnecessarily hinder performance whereas an optimization level of O3 caused the number of memory accesses on the x86 Machine to appear less than they actually were. Since our algorithm acts as a filter that reduces the number of memory accesses to those expected on a Spatial Architecture, the starting x86 memory trace of a deterministic benchmark should posses at least as many memory accesses as the spatial trace. When an optimization level of O3 was used, the x86 ISA specific memory instruction move unaligned double quad word (MOVDQU) was used in some applications causing the number of memory accesses performed to appear less than those performed on the Spatial Architecture resulting in asymmetric traces. Therefore, to fairly and easily assess the performance of our prediction algorithm in terms of the predicted number of memory word load and store operations, an optimization level of O2 was preferred. We obtained Intel Proprietary hand-written RTL implementations of the benchmarks optimized for a smaller sized FPGAs with an embedded memory capacity of approximately 32 KBytes. We ran the benchmarks on an FPGA simulator configured with an embedded storage capacity of 32 KBytes to obtain reference memory traces against which we could evaluate the performance of our algorithm. For practical reasons we only display the baseline number of memory accesses performed on each architecture in **Figures 4-(a)** and **4-(b)** and not the detailed traces. After establishing a reference for comparison, we configured the reuse-distance parameter of the implemented Pin-tool appropriately to represent 32 KBytes of embedded storage for the various benchmarks and used it on the compiled x86 binaries of the benchmarks. Since the processor and the FPGA simulator have different address spaces, some additional instrumentation

Table 2 Benchmarks used and their Description. **(R)**: Regular Access Patterns. **(I)**: Semi-Regular/ Irregular Access Patterns.

Benchmark	Description
MiniGhost (**R**)	A Mantevo Suite miniapp that implements a difference stencil across a homogenous three dimensional domain.
Blocked DGEMM (**R**)	A simple blocked dense-matrix multiply benchmark.
Stream-Triad (**R**)	A simple synthetic benchmark that measures sustainable memory bandwidth (MB/s) and a corresponding computation rate of a simple vector kernel.
BlackScholes (**R**)	An Intel RMS benchmark that calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. The benchmark is data parallel with structured memory accesses.
Graph500 (**I**)	A CORAL benchmark that implements scalable breadth first search of a large undirected graph.
SPMV (**I**)	A benchmark that multiplies a sparse matrix by a source vector. The sparse matrix is stored in CSR format.
XS (**I**)	A CORAL benchmark that evaluates the performance of the memory subsystem. It performs look-ups for terms within an array using binary search.
MiniMD (**I**)	A light-weight molecular dynamics application from the Mantevo suite. The main kernel computes the Lennard Jones force of each atom.
SPMDM (**I**)	A benchmark that multiplies a sparse matrix by a dense matrix. The sparse matrix is stored in CSR format.
Stencil (**R**)	A simple one dimensional implementation of a weighted average computation.
Haccmk (**R**)	A CORAL benchmark that ccomputes the short force of particles.

and analysis was involved to map the addresses obtained to variables. Also, since spatial traces tend to reorder memory operations with respect to each other as long as their dependencies are resolved, we don't expect our tool's output trace to have the same global ordering of memory accesses as that obtained on the FPGA simulator.

To assess the performance of the proposed algorithm, we calculate the Average Experimental Prediction Accuracy **(AEPA)** over the benchmarks. This is done separately for read and write memory accesses. The Experimental Prediction Accuracy **(EPA)** is defined in **Equation 1**. In some cases,we discover that the obtained RTL implementations of the algorithms are not optimal for the given inputs, and the number of memory read accesses performed on the FPGA would be less if additional memory system optimizations such as implementing caches were performed. We therefore report two average experimental prediction accuracy values for read accesses; one that uses the number of memory accesses obtained using the RTL implementations as the target number of accesses, and one that uses the expected optimal number of accesses if further optimization opportunities suggested by the algorithm are considered.

tr = target number of read accesses

tw = target number of write accesses

pr = Algorithm's predicted number of read accesses

pw =Algorithm's predicted number of write accesses

$$
\begin{aligned}
EPE_{Reads} &= \frac{|tr - pr|}{tr} \times 100 \\
EPE_{Writes} &= \frac{|tw - pw|}{tw} \times 100
\end{aligned}
\tag{1}
$$

4.2 Spatial Memory Trace Prediction Algorithm Performance Results

As can be seen in **Figure 4-(d)**, our proposed algorithm performs mostly well in predicting the number of write memory accesses on the investigated benchmarks. Though the results obtained for DGEMM and Graph500 might appear contradictory, the mismatch was not a result of our filtering algorithm.

In DGEMM, the mismatch results because the algorithm of the compiled executable includes multiple write initializations which are not included in the hand-written RTL version. Since these initializations are considered as a part of the program, the algorithm does not filter them out of the trace to maintain intended program behavior.

In the case of Graph500, the order in which the neighbouring nodes are visited in each implementation varies resulting in non-deterministic graph traversal. This is supported by **Figures 4-(a) and 4-(b)** showing that the baseline number of both read and write memory accesses on the x86 machine were actually lower than those of the FPGA.

To avoid long execution times, we only keep track of the previous and current addresses accessed by a memory operation at a particular PC. Since we rely on PC-based filtering to flag memory addresses that correspond to potential stack/constant values, these addresses are flagged the second time a read memory access happens at a particular PC. Since write accesses to those addresses are eliminated once the address is flagged as can be seen in **Figure 3**, this means that only the write accesses of loop iterations 3 and 2 are eliminated. Stack write accesses that occurred in iteration 1 will still be present in the trace. Such accesses can be further eliminated by making a final static pass over the generated trace and eliminating the write accesses from iteration 1. However, this approach results in a performance overhead. Since the number of overhead write accesses is negligible particularly when dealing with large scale benchmarks, we choose to optimize the performance at the expense of perfect prediction.

The algorithm's performance in predicting the number of read accesses is less uniform and greatly dependent on the memory access patterns of the benchmarks as well as the size of the working set. We classify the benchmarks into groups based on their access patterns and evaluate the algorithm's performance in predicting the number of read memory accesses. We then analyze the obtained results on a per category basis in the following sections. As can be seen in **Table 3**, the proposed algorithm demonstrates an average experimental read prediction accuracy of 97.45% when the RTL implementations determined to be sub-optimal are further optimized, and an average prediction accuracy of 67.31% when the baseline RTL implementations are used as the target. It also achieves an average write prediction accuracy of 90.71%.

4.3 Benchmarks with Regular Memory Access Patterns

As can be seen in **Figure 4-(c)**, with the exception of Haccmk our algorithm either matches the target FPGA number of read accesses or performs an overestimate for the benchmarks with a tag **R** indicating regular memory accesses in **Table 2**.

BlackScholes provided an example of a benchmark with most overhead accesses resulting from overhead x86 stack accesses as well as reading constants from memory. We analyzed the traces obtained from our tool and the memory read accesses matched those obtained on the FPGA simulator.

Stream-Triad was investigated despite the fact that the baseline x86 and FPGA number of accesses were a match for completeness and demonstrated that the algorithm doesn't result in the elimination of accesses needed for correct program execution. Similarly, the traces were a match. The predicted number of read accesses for the Stencil Benchmark slightly overestimates the target number of read accesses. Upon analyzing the traces, we determined that the generated trace did in fact match the FPGA reference trace, however there were additional read accesses resulting from loading the stencil elements for the first time. Despite the mismatch, this is still reasonable since it is not uncommon to initialize the values of the stencil array from memory. Nonetheless, we consider this as a prediction overhead in assessing our algorithm.

The number of read accesses predicted for DGEMM exceeded the reference number of accesses by approximately 3%. The memory access patterns of DGEMM are well-known and this allows RTL designers to write blocked optimized implementation of DGEMM that minimize the number of required memory accesses. In theory the optimal implementation for Blocked DGEMM in terms of memory accesses divides the working set into tiles that fit within faster memory. Since the input matrices are of dimensions $n \times n$ with $n = 256$ and the fast memory here is the amount of embedded storage, the optimal tile size b is 32. Theoretically, the number of memory read memory accesses required to perform the multiplication is $2 \times \frac{n^3}{b}$ to access elements of the input matrices and n^2 accesses to load the grid sum values. Performing this calculation results in 1,114,112 memory read accesses. This was in fact the number of baseline FPGA read memory accesses indicating that the hand written reference RTL was optimized and valid as a reference.

MiniGhost is a great algorithm for Spatial Architectures due to the deterministic nature of its memory access patterns and the low precision computations involved. It also has a lot of memory access locality which can be exploited to improve performance. Unlike the hand written RTL, the C code is not implemented in a blocked fashion to exploit temporal locality of accesses. The benchmark consists of 3 nested loops that span a grid with dimensions $64 \times 64 \times 64$. With such an input size, the reuse distance is not able to capture the locality present and therefore our algorithm overestimates the target value by 24 %. A fair comparison is a blocked implementation of the Minighost benchmark. A similar observation was made when a non-blocked implementation of the DGEMM benchmark with a matrix size of 256×256 was used. Nonetheless the algorithm is still able to capture a lot of the locality in accesses with a prediction error below 25%.

Haccmk displays the algorithm's performance when the entire working set fits in memory resources within the FPGA. Though the prediction error of Haccmk was determined to be 100%, further analysis of the obtained traces revealed that the algorithm's prediction was in fact optimal, and the RTL implementation didn't exploit the FPGA's full potential.The handwritten RTL was a generic implementation of Haccmk which wasn't optimized for the given input set and the target FPGA. Haccmk was setup to run for 3 repetitions with inner loop counts of 400, 420, and 440 respectively. Each loop iteration three values are read from three distinct input arrays holding the values of each dimension of a 3-D force vector. An additional memory access might be performed to a fourth array storing the mass associated with each force entry, however accesses to this array are control determined. At the end of the first repetition, 420 input values are read from each array resulting in a total of 1260 read accesses. Since each element is 4 bytes, the first 400 elements of the force arrays can be kept in local BRAM or a cache within the FPGA. The next iteration, there is no longer any need to access the first 400 elements in memory since they are still available from the previous iteration, and the newly accessed 20 elements are additionally loaded. The same is true for the third iteration. Our obtained trace indicates that the first 440 elements of

Table 3 Summary of Algorithm's Evaluation Results.

Benchmark	Baseline Number of x86 Read Accesses	Target Number of FPGA Read Accesses	Target Number of FPGA Read Accesses with Optimizations	Algorithm's Predicted Number of Read Accesses	Experimental Prediction Accuracy for Read Accesses (%)	Experimental Prediction Accuracy for Read Accesses with Optimizations (%)	Baseline Number of x86 Write Accesses	Target Number of FPGA Write Accesses	Algorithm's Predicted Number of Write Accesses	Experimental Prediction Accuracy for Write Accesses (%)
MiniGhost	4849969	663552	663552	823680	75.87	75.87	558442	262144	262144	100.00
Blocked DGEMM	33554439	1114112	1114112	1146880	97.06	97.06	16842758	65536	2195464	0.00
Stream-Triad	2001	2000	2000	2000	100.00	100.00	1000	1000	1000	100.00
BlackScholes	243728	43008	43008	43008	100.00	100.00	172046	28672	28678	99.98
Graph500	2195185	2202251	2143573	2143573	97.33	100.00	130730	130889	128559	98.22
SPMV	2557292	2557288	1790193	1790194	70.01	100.00	32768	32768	32768	100.00
XS	36830	31542	12288	12244	38.82	99.65	2048	2048	2048	100.00
MiniMD	4573219	4530176	1390254	1390256	30.69	100.00	43022	43008	43008	100.00
SPMDM	2697090	2696704	838088	838090	31.08	100.00	32771	32768	32771	99.99
Stencil	203601	4096	4096	4121	99.39	99.39	4072	4072	4072	100.00
Haccmk	9625883	1382540	1730	1730	0.13	100.00	3722895	2943	2953	99.66
Average Experimental Prediction Accuracy					**67.31**	**97.45**				**90.71**

the force vectors should be stored on-chip or will be kept locally if the implemented RTL included a cache. It also indicated that repeated accesses to values in the mass array will be eliminated by implementing a cache.

4.4 Benchmarks with Semi-Regular or Irregular Access Patterns

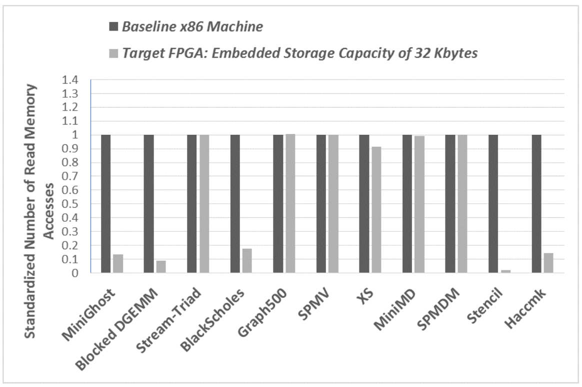
As can be seen in **Figure 4-(c)**, our algorithm underestimates the target FPGA number of read accesses for the benchmarks with a tag **I** indicating semi-regular or irregular memory accesses in **Table 2**. When the predicted number of read memory accesses was less than that of the reference, we analyzed the traces generated to establish that program correctness was maintained and verify whether the obtained results are justifiable. Due to the irregular access patterns, optimizing the hand written kernel is challenging particularly when knowledge of the input set can't be assumed.Since the prediction tool instruments the code dynamically, all memory accesses are visible, and the tool returns the expected number of read accesses optimized for both the benchmark and the input set.

Our algorithm predicts a substantially lower number of read memory accesses for XS than that of the hand written RTL. The investigated XS benchmark performs binary search on an input array of size 80,000 to find 2048 terms. Upon comparing the predicted memory trace against the reference hand-written version, we discovered that the reference FPGA code implemented binary search naively instead of applying the optimization demonstrated in **Section 2**. In theory, the average expected number of memory accesses for Binary Search is $O(\log n)$ with n being the size of the queried array. With an embedded storage capacity of 32 KBytes and a word size of 8 Bytes, an optimized implementation would keep the hottest 4096 terms on chip. If we consider storing the terms in a binary search tree, then this would mean that all values from the root up to the twelvth level of the tree are kept on-chip. Therefore, on average, instead of having to perform fifteen comparisons, only three would be required. This results in an expectation of a total of 12,288 memory accesses; 4096 accesses would be required to load the hottest terms into local BRAMs, 2048 accesses to read the input array, and 2048×3 additional accesses on average for comparisons with values not held on-chip.

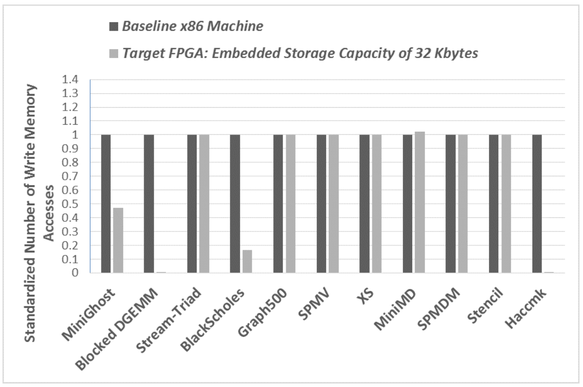
Similarly, the output trace obtained for SPMV suggested potential optimizations to the hand-written implementation. The investigated SPMV benchmark stores the sparse matrix in multiple arrays in CSR format. An array keeps row pointers, a second array stores the values of all the non-zero terms of the sparse matrix, a third array stores all the column indices of the nonzero values, and a fourth array stores the actual source vector to be used for the multiplication. The output trace of the tool contained all initial accesses to the three vectors, however, it lacked multiple repeated accesses to the same element in the source vector array. This indicated the potential for exploiting locality in accesses to the source vector array by storing it on chip. Otherwise, an RTL implementation that locally stores the most accessed entries of the source vector on chip would be optimal. However, our algorithm fails to filter out an overhead stack access resulting from reading the base address of the result array.

The results of SPMDM were similar to those obtained for SPMV. However, since the size of the sparse matrix used was much smaller than that used in the SPMV benchmark, the algorithm was able to detect additional optimization opportunities through detecting access locality, and a noticeable reduction in the number of memory read accesses is observed. Assuming the hand-written RTL included a cache, then the results predicted by our algorithm would have been obtained on the FPGA. Nonetheless, two overhead stack accesses resulting from reading parameter values passed through the stack were present in the predicted trace.

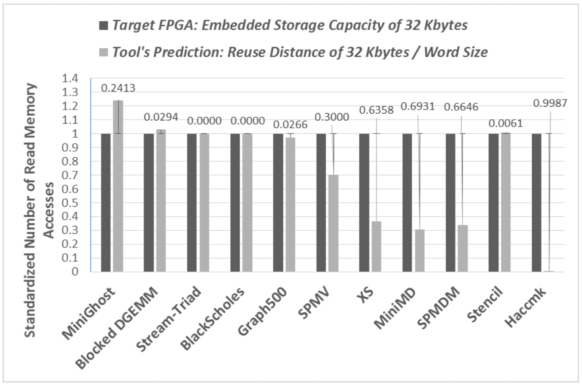
The obtained trace for MiniMD also suggested potential for improvement. The written code partitions the molecules into groups to allow job division and speed up execution. However this comes at the expense of redundant memory accesses to atoms made by each group. Our algorithm detects the nodes which are mostly clustered and eliminates a significant fraction of repeated accesses to them.



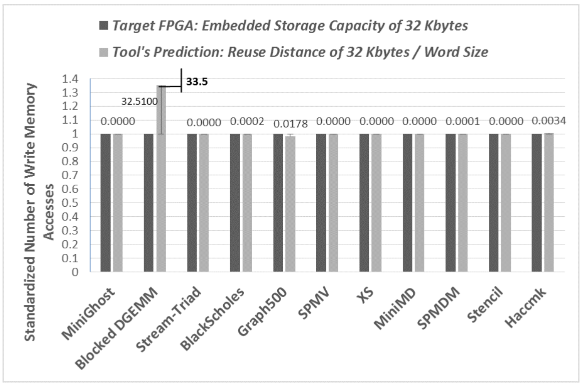
(a) Number of Memory Reads in the Target FPGA relative to the Memory Reads of the Baseline x86 Machine.



(b) Number of Memory Writes in the Target FPGA relative to the Memory Writes of the Baseline x86 Machine.



(c) Number of Memory Reads Predicted by the Algorithm relative to the Memory Reads in the Target FPGA.



(d) Number of Memory Writes Predicted by the Algorithm relative to the Memory Writes in the Target FPGA.

Figure 4 Obtained Number Memory Read and Memory Write Accesses.

5 Related Work

Predicting the performance of applications on various architectures is a very important problem, and a lot of work has been done within this scope over the past decades. Simulation-based performance prediction is the most accurate technique for performance prediction, and there are currently many simulators available in academics and industry. SimpleScalar **[8]**, Simics **[9]**, and SimOS **[10]** are only a few examples. However, as hardware architectures are becoming less homogeneous, and applications are increasing in scale, detailed simulation is no longer as attractive as it used to be. It is too slow especially within the context of large scale HPC applications. Carlson e.t.al. use this observation to create a performance projection tool that balances detailed cycle accurate simulation with one Instruction Per Cycle (IPC) simulation to provide a more reasonable compromise between simulation run-time and performance projection accuracy **[11]**. Though their proposed approach mitigates long simulation times, it still does not provide insights on the bottlenecks limiting performance, and on how performance can be improved. These limitations of simulation motivated the creation of the roof-line model by William et al. The roof-line model that attempts to provide valuable insights on the primary factors affecting the performance of a system. William et al. further observe that as processors speed continue to outperform available memory systems, memory bandwidth will often be the constraining resource in system performance. Like Williams et al. we propose a simpler approach to predicting which applications are applicable to FPGAs in an insightful manner. However, unlike their work, we target FPGAs and assume that any level of computation demanded by an application is likely to be supported. Using instrumentation driven simulation for memory characterization of workloads was first introduced by Jaleel **[12]** as he characterized the memory profiles of SPEC benchmarks on a Xeon Processor using Pin, a DBI framework introduced by Intel. Similarly, Ostadzadeh et al. use Pin to approximate application behavior in function of the memory bandwidth used by the kernels to predict their performance on FPGAs **[13]**. However, their work relies on x86 memory traces to guide the their optimizations. Our approach predicts the expected memory profiles on the Spatial Architectures. To develop our algorithm for predicting spatial traces, we borrow from compiler memory optimization techniques for enhancing memory locality. More specifically, reuse distance was first introduced by Wolf and Lam as they observed that the performance of parallel architectures are limited by memory bottleneck, and software optimizations that exploit locality are necessary to improve the performance of algorithms **[14]**. Baradaran et al. similarly observe that FPGAs are nothing but parallel systems, and that their configurable embedded memory blocks can be exploited for performance improvements. They rely on reuse analysis to guide the mapping of data onto FPGA Block RAMs **[15]**. However, they assume unlimited storage and most later decide on how to map the data. We take a different approach and limit the reuse distance to be representative of the storage capacity supported by the FPGA. Though our work borrows from familiar pre-existing computer architecture ideas, it combines them in a novel way to provide a useful tool for a broader software community.

6 Summary and Conclusion

In this paper we presented a novel approach for predicting spatial memory traces from abundantly available x86 benchmarks to help the broader software community in projecting the performance of existing and emerging workloads on FPGAs. By implementing our algo-rithm in a Pintool, we demonstrate that we can bound expected memory bandwidth of various benchmarks on FPGAs within 10% of the reference value in a fast, transparent, and highly portable manner. Such prediction accuracy is sufficient for projecting the performance of applications given certain memory bandwidth constraints on FPGAs using bound and bottleneck approaches such as the roof-line model. We further demonstrated that examining the obtained traces can provide FPGA programmers with insights on how to further optimize their implementations for benchmarks with irregular access patterns in **Section 4**. Since the obtained traces are very similar to those obtained on an FPGA, they can be used directly as inputs to memory subsystem simulators to determine the best memory sub-system configuration for each benchmark and workload.
Though our entire work was within the context of a single thread, we believe that our tool will be even more useful if it is extended to multiple threads. By bounding the required memory bandwidth of an application, we believe that parameters such as the number of workers used in par-allel algorithms can be tuned to provide optimal performance based on the memory bandwidth consumption predicted by our tool under fixed memory re-source constraints. We do not present this work here, but hope to extend our tool to support multiple threads and present it in our future work.

References

[1] J. S. da Silva and S. Bampi, "Area-oriented Iterative Method for Design Space Exploration with High-Level Synthesis," 2015 IEEE 6th Latin American Symposium on Circuits Systems (LASCAS), Montevideo, 2015, pp. 1-4.

[2] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM, 52:65–76, 2009.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of Programming Language Design and Implementation (PLDI), 2005, pages 191–200, Chicago, Illinois, USA, June 2005.

[4] "CORAL Benchmark Codes," 2014. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/.

[5] Princeton University, "PARSEC: The Princeton Application Repository for Shared Memory Computers," 2007. [Online]. Available: http://parsec.cs.princeton.edu/.

[6] J. Teubner, R. Mueller, and G. Alonso, "FPGA Acceleration for the Ffrequent Item Problem," in Proc. of the 26th Int'l Conference on Data Engineering (ICDE), Long Beach, CA, USA, Mar. 2010.

[7] F. Winterstein, K. Fleming, H. J. Yang, J. Wickerson and G. Constantinides, "Custom-sized caches in application-specific memory hierarchies," 2015 International Conference on Field Programmable Technology (FPT), Queenstown, 2015, pp. 144-151.

[8] D. Burger et al. "The SimpleScalar Tool Set, Version 2.0." Technical Report 1342, Computer Sciences-Department, University of Wisconsin-Madison, June 1997.

[9] J. Engblom and D. Ekblom. "Simics: A commercially proven full-system simulation framework." In Workshop on Simulation in European Space Programmes, Nov. 2006.

[10] Mendel Rosenblum and Mani Varadarajan. "SimOS: A Fast Operating System Simulation Environment." Technical Report CSL-TR-94-631, Stanford University, July 1994.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout. "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulations. In International Conference for High Performance Computing, Networking, Storage and Analysis, " Nov. 2011.

[12] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation – A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites," Intel Corporation, VSSAD, 2007.

[13] S. A. Ostadzadeh, M. Corina, C. Galuzzi, and K. Bertels, "tQUAD - Memory Bandwidth Usage Analysis," in International Conference on Parallel Processing (ICPP), 2010, September 2010, pp. 217–226.

[14] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm." In Proceedings of Programming Language Design and Implementation (PLDI), 1991.

[15] N. Baradaran, Joonseok Park and P. C. Diniz, "Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks," In Proceedings of the International Conference on Field Programmable Technology (FPT), 2004, pp. 145-152.