Using Runahead Execution to Hide Memory Latency in High Level Synthesis

Shane T. Fleming and David B. Thomas,
Dept. of Electrical and Electronic Engineering, Imperial College London, London, UK.
{sf306, dt10}@ic.ac.uk

***Abstract*—Reads and writes to global data in off-chip RAM can limit the performance achieved with HLS tools, as each access takes multiple cycles and usually blocks progress in the application state machine. This can be combated by using data prefetchers, which hide access time by predicting the next memory access and loading it into a cache before it's required. Unfortunately, current prefetchers are only useful for memory accesses with known regular patterns, such as walking arrays, and are ineffective for those that use irregular patterns over application-specific data structures. In this work we demonstrate prefetchers that are tailor made for applications, even if they have irregular memory accesses. This is achieved through program slicing, a static analysis technique that extracts the memory structure of the input code and automatically constructs an application-specific prefetcher. Both our analysis and tool are fully automated and implemented as a new compiler flag in LegUp, an open source HLS tool. In this work we create a theoretical model showing that speedup must be between 1x and 2x, we also evaluate five benchmarks, achieving an average speedup of 1.38x with an average resource overhead of 1.15x.**

I. Introduction

High-Level Synthesis (HLS) tools are used to map high-level application descriptions into FPGA circuits. Traditionally HLS tools were reserved for applications with regular memory access patterns, however, in recent years, they have been used to develop applications with irregular access patterns. Irregular memory accesses are problematic as they are difficult to predict statically at synthesis time because they are often dependent on runtime data.

Predicting memory accesses is important as it can be used to prefetch data before it is needed hiding memory latency. Consider the C code in Figure 1(a) which accumulates all elements within a linked list; without prefetching, the performance of this application will be limited by the memory access latency. To hide memory latency a runtime prefetching system is required, but current generic FPGA prefetchers are limited to regular access patterns and so perform poorly for irregular applications such as the code in Figure 1The reason these generic approaches are limited to regular patterns is because they do not consider semantic information for memory accesses, but what if this was taken into consideration and used to construct the prefetcher?

This work aims to answer this question by creating a static code analysis which extract an applications memory access structure to build a runtime application specific prefetcher. The static code analysis technique used is called *program slicing* [1], and has been previously applied to HLS for FPGA reliability [2] and ASIC performance [3]. Program slicing creates an executable fragment of the original program, known as a slice, that only evaluates specific instructions referred to

```
   float AccumulateList() {
I0     float data, Acc = 0.0;
I1     struct *node tmp = HEAD;
I2     while(tmp != NULL) {
I3         data = tmp->data;
I4         Acc = Acc + data;
I5         tmp = tmp->nxt;
       }
I6 return Acc; }                    (a)
```

```
   Slice_{I3, I5} {
S0   float data;
S1   struct *node tmp = HEAD;
S2   while(tmp != NULL) {
S3       data = tmp->data;
S4       tmp = tmp->nxt;
     }
   }                                (b)
```

Fig. 1: (a) Accumulate Linked List example application, (b) precomputation slice for AccumulateList with slicing criterion I_5 and I_3

as *slicing criterion*. Runahead execution is when a slice can resolve their slicing criterion ahead of the original program, we refer to a slice used in this context as a *Precomputation Slice* (pslice). This paper presents an algorithm and tool for automatically constructing pslices for HLS generated FPGA hardware, where the slicing criterion is every long latency global load in the hardware accelerated function. The generated pslice prefetches these loads, regardless of how irregular the access pattern is.

For example, Figure 1(b) shows a pslice for the `AccumulateList()` function from Figure 1(a), where the slicing criterion are the load instructions I_5 and I_3. Operations superfluous to the slicing criterion are excluded from the pslice, so in this case I_4 is excluded from the slice as it does not influence the result of I_5 or I_3. This dropped instruction (I_4) is an expensive multicycle floating-point operation; its removal enables the pslice to execute an iteration of its loop in fewer cycles than the original circuit, fetching the memory into a local buffer. Our tool automatically detects and builds pslices to exploit run ahead opportunities like this, only requiring the user to set a compiler flag. The main contributions of this paper are:

- **RELISH** (**R**unahead **E**xecution of **L**oad **I**nstructions via **S**liced **H**ardware) – an open source HLS optimisation pass built for LegUp an open source HLS tool, which automatically constructs a pslice for a hardware accelerated function. (Sections II – III)
- **A theoretical performance model** showing that using a pslice results in a 1x – 2x speedup, along with how memory latency and compute time affect performance. (Section IV)
- **An evaluation of five benchmarks** with both regular and irregular memory access, demonstrating performance gains between 1.02x and 1.69x. (Section V).

II. Our Solution

In RELISH we want to evaluate loads ahead of the original circuit, so every load in the original function is set as a slicing criterion. To effectively prefetch data the pslice needs to run

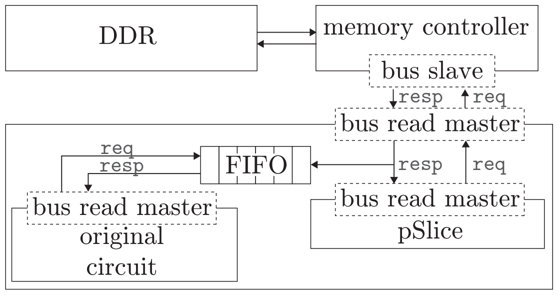


Fig. 2: Overview of the architecture when a pslice is used with the original circuit.

ahead, which depends on whether excluded instructions enable the pslice to remove false dependencies on the load instructions.

We define false dependencies as situations where an instruction, `x`, must wait for the execution of another instruction, `y`, to complete, even though the result of `y` does not directly influence the result of the `x`. Figure 1(a) shows an example of this where the instructions I_3 and I_5 must wait for the instruction I_4 to complete, despite it having no influence over their value but because of scheduling restrictions as they are within in the same basic block. Removing enough of these false dependencies enables the pslice to run ahead of the original, since the latency of the loop body has been reduced. If it is not possible to remove false dependencies, then the pslice is unable to run ahead of the original, and there is no reason to instantiate it. Fortunately, it is possible to statically detect cases where there is insufficient run ahead, allowing the compiler to build a pslice only when it is beneficial. Through detecting such cases we can ensure that RELISH will never make an application slower.

Using every global memory read as a slicing criterion provides the important property that all loads in the pslice appear in the original program order. This property enables the pslice to be coupled with the original function via a FIFO, detailed in Figure 2. Such tight coupling provides an advantage over traditional prefetchers which generally rely on a weaker coupling via a cache.

The current implementation of RELISH targets reconfigurable SoC systems. Figure 2 shows how hardware functions connect to the processing system with accesses to global memory serviced through a global memory controller bus slave port. The read bus of the memory controller can be split into two channels; a request channel that is given the address of the global memory location along with handshaking signals to start the transaction; and a response channel containing the returned data and handshaking signals to complete the transaction.

RELISH modifies the interconnect between each hardware accelerated function and the memory hierarchy so that the function's pslice takes complete responsibility for the read request and response channels. All global read requests are issued from the pslice, then responses are fed both directly into a FIFO connected to the original circuit and to the pslice. Whenever the original accelerated function issues a global read it pops the data off the FIFO, unless the FIFO is empty where it blocks waiting for the pslice to push the next read response.

A - Case study, linked list accumulation: Constructing the pslice in Figure 1(b) from the code in Figure 1(a) requires two main steps. Firstly all global loads in the program are used as slicing criterion; in this example, only two loads are used; I_3 and I_5. The second step then performs a dependency analysis grabbing any instruction which may influence the outcome of any slicing criterion. In this case there are dependencies: $\{ I_3 \leftarrow I_2, I_3 \leftarrow I_1, I_3 \leftarrow I_0, I_5 \leftarrow I_2, I_5 \leftarrow I_1 \}$, so the slice is constructed from the instructions, $\{ I_3, I_5, I_2, I_1, I_0 \}$.

Modern HLS tools, such as LegUp, build hardware from code that has been transformed into a Control and Data Flow Graph (CDFG) [10]. In this representation, the code is arranged into straight line blocks - known as basic blocks - which must have a single entry point at the top and one or more exit branches at the bottom. HLS tools exploit the fine grain parallelism of FPGA devices by scheduling operations within a basic block to execute in parallel while obeying both structural and data dependencies.

The HLS schedule for the example in Figure 1 is given in Figure 4 provided we assume: that there is a single memory port; that load operations, such as I_3, complete in four cycles; that floating-point addition operations, such as I_4, complete in 8 cycles; and that every other instruction completes in a single cycle. In this case, the run ahead opportunities for the pslice is in the inner loop that iterates over the linked list (`BB3`), where the pslice can complete four cycles faster than the original since it does not require the instruction I_4.

The performance benefits of this can be seen in Figure 3 which shows timing diagrams for: the original program (P), the pslice (S), and the original with slice ($P - S$), where the input is a 4 element linked list. For the first iteration, there is no speedup experienced in $P - S$ since S has not yet had a chance to outrun P. However, in subsequent iterations, the benefit of S running ahead can be seen, with smaller memory latency in $P - S$. By the end of the function call S traversed the linked list 17 cycles ahead of the original circuit enabling $P - S$ to finish 12 cycles ahead.

III. Technical Details

Invoking RELISH occurs between stages of the LegUp HLS toolchain and is completely automated, building the circuit, its associated pslice, and logic to interface them. A user wishing to apply RELISH to a compilation is required to do nothing more than set a compiler flag.

Specifically, it works within the LegUp **hybridparallel** flow where the user provides C or C++ source code along with a list of functions to accelerate and the tool generates software-only and hardware-only portions. Multiple hardware accelerators may also be executed concurrently using pthreads or OpenMP pragmas. Natively this flow targets an Altera/Intel CycloneV SoC device. However, we wished to use existing Xilinx based compiler passes for future research and so ported it to a Xilinx Zynq device.

RELISH is a type of LLVM transformation pass, i.e. something that accepts LLVM-IR (intermediate representation), transforms it, and then emits it. The RELISH transformation modifies the input in two ways. Firstly, for each function listed to be accelerated, it builds a pslice and implements it as a hardware accelerated pthreads function in the source

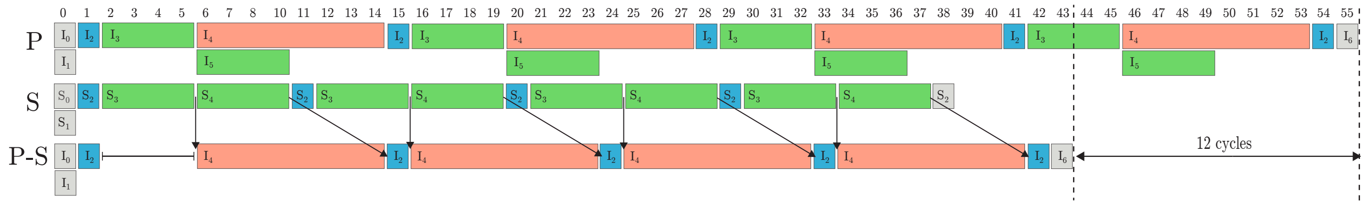


Fig. 3: Execution trace for the HLS program and pslice in Figure 1. Where P is the original program, S is the pslice, and P_S the execution of the original program with support from the pslice.

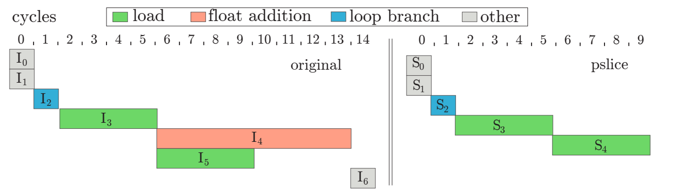
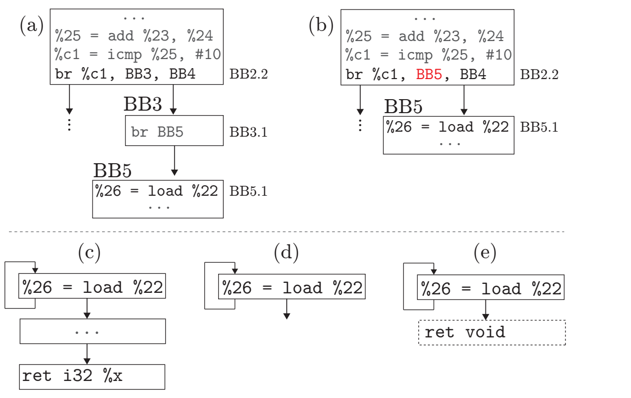


Fig. 4: HLS schedule for the original function and pslice in Figure 1



(a)

```
...
%25 = add %23, %24
%c1 = icmp %25, #10
br %c1, BB3, BB4
```

BB2.2

BB3

```
br BB5
```

BB3.1

BB5

```
%26 = load %22
...
```

BB5.1

(b)

```
...
%25 = add %23, %24
%c1 = icmp %25, #10
br %c1, BB5, BB4
```

BB2.2

BB5

```
%26 = load %22
...
```

BB5.1

(c)

```
%26 = load %22
...
ret i32 %x
```

(d)

```
%26 = load %22
```

(e)

```
%26 = load %22
ret void
```

Fig. 5: (a) Example fragment of a pslice where because `BB5.1` is reachable from `BB2.2` all branch instructions are included. (b) Optimised pslice through pruning unused basic block `BB3`. (c) Original program with non-void return. (d) pslice with broken hanging branch. (e) Repaired pslice

(Section III-A). Secondly, for each newly created pslice, a call to it is added above each corresponding function call in the SW portion of the hybrid parallel code.

After the pslice construction, the LLVM-IR is handed to the LegUp hybridparallel backend to generate Verilog modules for the accelerated functions and their corresponding pslices. The final stage of the tool takes the generated output from the LegUp hybridparallel flow and for each pslice/accelerator pair, joins them via an AXI-based FIFO interface (Section III-B).

A - pslice Construction: The input Program, P, can be viewed as a set of single static assignment instructions, while it's pslice S ($S \subseteq P$) contains only the set of instructions required to evaluate a set of slicing criterion C ($C \subseteq P$). For this work we are interested in evaluating loads in S ahead of the original loads in P, so C is the set of all global load instructions in P. In order to construct S we defined the following functions:

- $\texttt{isOP}(x \in P, y \in P)$ – returns `true` if a variable assigned by x is used as an operand of y.
- $\texttt{isCD}(x \in P, y \in P)$ – returns `true` if x has a control dependency on the result of y.
- $\texttt{isBR}(x \in P)$ – returns `true` if x is a branch instruction.

We can then say that S is the smallest set that satisfies all of the following properties:

1) $C \subseteq S$ (S must contain all of the slicing criterion)
2) $\forall x, y : y \in S \wedge \texttt{isOP}(x, y) \rightarrow x \in S$
3) $\forall x, y : y \in S \wedge \texttt{inBR}(x) \wedge \texttt{isCD}(x, y) \rightarrow x \in S$

While the rules above can be used to construct a valid pslice the result can be further optimised through pruning unused basic blocks. For example, since Figure 5(a) has a load at `BB5.1` the branch operations `BB3.1` and `BB2.2` must be included in the pslice; however, since the only instruction in `BB3` is an unconditional branch the entire basic block can be pruned giving Figure 5(b).

One final further step is required to complete the pslice. The last point in every dependency chain is a slicing criterion, this usually means that there is no `return` instruction within the pslice, so one must be added to make it a legal LLVM function [4]. A simple fix for this is to detect all hanging branches in the pslice and link them to a basic block containing a single `return void` instruction. This is demonstrated in Figure 5 where, Figure 5(c) shows an unsliced program, Figure 5(d) shows the pslice for that function, and Figure 5(e) shows the repaired dangling branch.

B - pslice Integration: The LegUp hybridparallel flow produces a hardware circuit for every `pthread` function call specified for acceleration. RELISH transforms the LLVM-IR before LegUp has split it into hardware and software portions. For each accelerated function call a pslice function is created using the analysis in Section III-A and added to the IR. Calls to this new pslice function are added to the IR preceding every call to the appropriate `pthread_create`. Metadata is added indicating that the pslice call is also a `pthread` call so that downstream when LegUp generates `pthread` stubs it will also do so for the pslice, enabling it to execute in parallel with the original code.

Every pslice is connected to its corresponding circuit via the Avalon master interface, as seen in Figure 2, where the pslice handles all read requests placing all responses in a FIFO for the original circuit to retrieve them later. However the FIFO is not just used to pass prefetched data from the pslice but for synchronisation, preventing the pslice from running ahead too far by stalling it when the FIFO is full and ensuring that FIFO

Original Code	pslice
`...`	`...`
`%1 = LD_addr();`	`%1 = LD_addr();`
`%2 = ST_addr();`	`%3 = ld, %1;`
`%x = compute();`	
`st %2, %x;`	
`%3 = ld, %1;`	

Fig. 6: LLVM-IR code to demonstrate possible write-after-read (WAR) hazard with current pslice implementation.

pop()s are blocking when the FIFO is empty. Stalling the pslice is achieved by spoofing a bus master transaction by asserting the `waitrequest` signal when the FIFO `full` signal is high, tricking the pslice into thinking it has an outstanding transaction. Implementing a blocking FIFO pop() is achieved in a similar fashion by asserting `waitrequest` of the original function when a read is attempted and the FIFO `empty` signal is high.

In RELISH we wanted to use existing Xilinx-based infrastructure available to us, so we developed an AXI interface with an Avalon/AXI bridge built into it to connect to the exposed Avalon ports generated by LegUp. Unfortunately, this introduces some small latency (2 cycles) either side of the FIFO to push and pop read responses. With some extra engineering effort this could be optimised but is outside the scope of this work.

C - Current Limitations: To minimise initial design complexity, we have restricted RELISH to functions that do not both read and write to the same global memory locations. This restriction avoids the complication of handling write-after-read (WAR) hazards, while still allowing for adequate testing of the benefits of the approach. Figure 6 shows an example of this – in the current implementation, this example is a potentially a problem since no analysis is performed to see if `ST_addr()` or `LD_addr()` can touch the same memory locations which would cause a WAR hazard if the pslice read data before it had been written.

Alleviating this restriction can be done at two levels, either at program analysis level or the interface between the original circuit and pslice. Expanding the program analysis requires analysing the memory space to determine how the global loads and stores are connected. Alternatively, the interface could be modified so the FIFO link resembles something closer to a cache where stores from the original circuit can update elements of the FIFO. Expanding the analysis has problems since it requires the inclusion of more instructions into the pslice, potentially increasing the overheads and hindering its runahead capabilities. While modifying the FIFO link requires more logic especially if the FIFO is large, however results in Section V indicate that a FIFO depth of only 16 elements is required. We believe that adapting the interconnect is the best approach for future work.

IV. Performance Model

Not all applications benefit from using a pslice to fetch loads; for some, loads may have too many true dependencies preventing their pslice from running ahead, while others may have so few memory operations that optimising them has little impact on performance. The amount of runahead, the memory latency, and the execution time are parameters which interact to influence the potential speedup obtained when using a pslice. In this section, we explore the balance between these parameters and mathematically derive equations to model speedup.

Speedup, S_p is calculated by dividing the original execution time, T_o, with the new execution time of the optimised version, T_{pref}, giving $S_p = T_o/T_{pref}$. The original execution time can be separated into two components, α for the time spent performing compute, and β the amount of time spent blocked on memory operations, giving: $T_o = \alpha + \beta$. For the execution time of the pslice, T_r, we define α_r for the time it spends performing compute, which allow us to build the equation, $T_r = \alpha_r + \beta$ for it's execution time.

To derive an equation for T_{perf} we need to consider two situations: first the case where $\alpha_r + \beta < \alpha$, which means the pslice has fetched the data and placed it in a FIFO before the original compute has had a chance to request it; in this situation memory latency has been fully hidden.

$$T_o = \vert\!\longleftarrow \alpha \longrightarrow\!\!\longleftarrow \beta \longrightarrow\!\vert$$
$$T_r = \vert\!\longleftarrow \alpha_r \longrightarrow\!\!\longleftarrow \beta \longrightarrow\!\vert$$
$$T_{pref} = \vert\!\longleftarrow \alpha \longrightarrow\!\vert$$

For the alternative case where $\alpha_r + \beta > \alpha$ the original compute needs to block for some time before the pslice has finished fetching from memory, this can be seen below where the original circuit is blocked for β_{rem}.

$$T_o = \vert\!\longleftarrow \alpha \longrightarrow\!\!\longleftarrow \beta \longrightarrow\!\vert$$
$$T_r = \vert\!\longleftarrow \alpha_r \longrightarrow\!\!\longleftarrow \beta \longrightarrow\!\vert$$
$$T_{pref} = \vert\!\longleftarrow \alpha \longrightarrow\!\!\longleftarrow \beta_{rem} \longrightarrow\!\vert$$

This means that the execution time of T_{pref} can be expressed as the critical path between α and $\alpha_r + \beta$ giving us, $T_{pref} = \max(\alpha, \alpha_r + \beta)$. Using this we get the following expression for speedup:

$$S_p = \frac{\alpha + \beta}{\max(\alpha, \alpha_r + \beta)} \tag{1}$$

Using Equation 1 and since $\alpha_r \geq 0$, we can establish an upper bound for S_p. In the case where $\alpha \leq \beta$ we get:

$$S_p \leq \frac{\alpha + \beta}{\max(\alpha, \beta)} \leq \frac{\alpha + \beta}{\beta} \leq \frac{\beta + \beta}{\beta} = 2$$

Similar reasoning for the case where $\alpha \geq \beta$ holds, so we find that for either cases $S_p \leq 2$.

The maximum speedup occurs when the denominator of Equation 1 is minimised, which occurs at $\alpha = \alpha_r + \beta$. Using this we define a term, γ, which we call the Runahead-Compute Balance (RCB), where:

$$\gamma = \frac{\alpha}{\alpha_r + \beta}$$

Maximum speedup occurs when RCB $\gamma = 1$, so we can substitute this into Equation 1 to get the maximum speedup $S_p^{\uparrow}$.

$$S_p^{\uparrow} = \frac{(\alpha_r + \beta) + \beta}{\max(\alpha_r + \beta, \beta)} = \frac{(\alpha_r + \beta) + \beta}{(\alpha_r + \beta)} = 1 + \frac{\beta}{\alpha_r + \beta} \tag{2}$$

Equation 1 also shows that the minimum speedup is x1, implying that a pslice never results in a performance penalty.

$$\lim_{\beta\to\infty} \frac{\frac{\alpha}{\beta}+1}{\max(\frac{\alpha_r}{\beta}+1,\frac{\alpha}{\beta})} = \frac{1}{1} \qquad \lim_{\alpha\to\infty} \frac{1+\frac{\beta}{\alpha}}{\max(\frac{\alpha_r+\beta}{\alpha},1)} = \frac{1}{1}$$

From this analysis, we can conclude that the maximum achievable speedup due to a pslice is 2x, while in the worse case the speedup is 1x, so under the theoretical model the pslice never slows the application down. Where an application lies in this range depends on the balance between three variables, the amount of compute for the original circuit (α), the amount of compute for the runahead pslice (α_r), and the memory latency of the system (β). Section V-B aims to verify this model using a synthetic benchmark.

V. Evaluation

This section evaluates RELISH in a live Zynq based system, based on measurements from hardware performance counters. There are two sets of experiments: a synthetic benchmark used to validate the analysis from Section IV; and five application benchmarks used to explore the overall achievable speedup versus resource.

A - Experimental Setup: We built the system using the LegUp-4.0 Xilinx tool flow as discussed in Section III, with each experiment performed on a Digilent Zedboard (Xilinx Zynq device, XC7Z020CLG484-1). For each application, SW was executed on the Zynq ARM cores and hardware circuits were connected to the memory system through the ACP port, allowing coherent access to the processing system's hardened L2 cache. The L2 cache was enabled for all tests.

Two different modes were tested; a baremetal mode where the experiment SW was running alone on the ARM cores, and a mode where Linux was also executing on the ARM. The Linux OS was Ubuntu 12.10 (Linux 3.12) and was used to obtain performance results where there was increased memory contention with higher system load.

Currently LegUp does not natively support floating-point operations in the Xilinx flow, so for applications where floating-point operations were used the softfloat library [11] was used to implement floating-point operations using integer arithmetic. Table I shows a performance comparison compared to VivadoHLS, showing the main impact is that the native floating-point would be around 1/3 the size, but the soft floating-point is 1.5 times faster. From the point of view of RELISH this is simply moving around the baseline α and α_r, so it is not expected to provide better or worse results.

The following performance counters were added to the FIFO interface logic.

- Cycle count for the original circuit & pslice
- Memory latency for the original circuit & pslice
- Number of cycles the FIFO full signal was asserted
- Number of cycles the FIFO empty signal was asserted

In all cases the circuits were targeting a 100MHz clock, and all experiments met timing.

B - Synthetic Benchmark: A circuit for the synthetic benchmark in Listing 1 was generated along with its corresponding pslice. This benchmark allowed us to vary: the problem size (N), the length of time spent blocked on memory (M); and the amount of compute (C). Through varying these parameters we were able to relate them back to the equations proposed in Section IV, such that $\text{N}\,T_{pref} = \text{C}\;\alpha_o + \text{M}\;\beta$, where α_r remains constant.

operation	implementation	cycles	LUTs	REGs	DSPs
fp_add	VivadoHLS	6	1157	1202	2
	Softfloat + LegUp	4	4021	4671	0
fp_mul	VivadoHLS	6	1003	1138	3
	Softfloat + LegUp	4	2624	3222	8

TABLE I: Softfloat implemented with LegUp performance and overheads vs VivadoHLS

Listing 1: Synthetic benchmark used to validate the model in Section IV

```
int synthetic(float32 *maddr, int M, int C, int N){
   volatile float32 t[BUFFSIZE], acc=0.0;
      for(int i=0; i<N; i++) {
        for(int m=0; m<M; m++)//Memory section
          t[m] = maddr[m];
        for(int c=0; c<C; c++) //Compute section
          acc = float32_add(acc, t[c]);
      }
    return acc;
}
```

Figure 7 shows the speedup of the synthetic benchmark in relation to the upperbound of the model as the parameters, N, M, and C, are varied to generate different RCB (γ) values. There is a strong resemblance to the shape of the model since there is a linear increase in speedup from the origin to a point around RCB (γ)=0.9 where it then switches to an inversely proportional relationship.

While the shape of the synthetic speedup matches the model, there are some differences as the synthetic benchmark never reaches the speedup of the model and the peak occurs a little before the model. We believe this can be explained by the constant extra two cycles incurred when accessing the FIFO connecting the pslice and the original circuit. The model assumes all memory latency can be eliminated by the pslice. However, in reality, these cycles prevent that stopping the speedup from reaching the theoretical maximum. These extra cycles also influence the RCB (γ) moving the position of the peak speedup. Another factor influencing the maximum speedup is α_r as Equation 2 implies that maximum speedup is only possible when α_r is negligible.

The synthetic benchmark crosses the x-axis at an RCB of $\gamma = 0.23$, this means that for a sufficiently memory bound application adding a pslice, in reality, could result in a slowdown, not a speedup. Similarly, this is also due to the additional latency introduced by the FIFO. With additional engineering effort, we believe that this FIFO interconnect could be further optimised dropping the overhead from two cycles to zero further improving the performance of using RELISH .

C - Benchmarks: Figure 7 also contains the speedup for five benchmarks (baremetal) plotted against their application's RCB (γ). For every case speedup is less than the theoretical upper bound, generally following the curve of the synthetic benchmark, and is always greater than 1.0x. Five benchmarks were used to test RELISH :

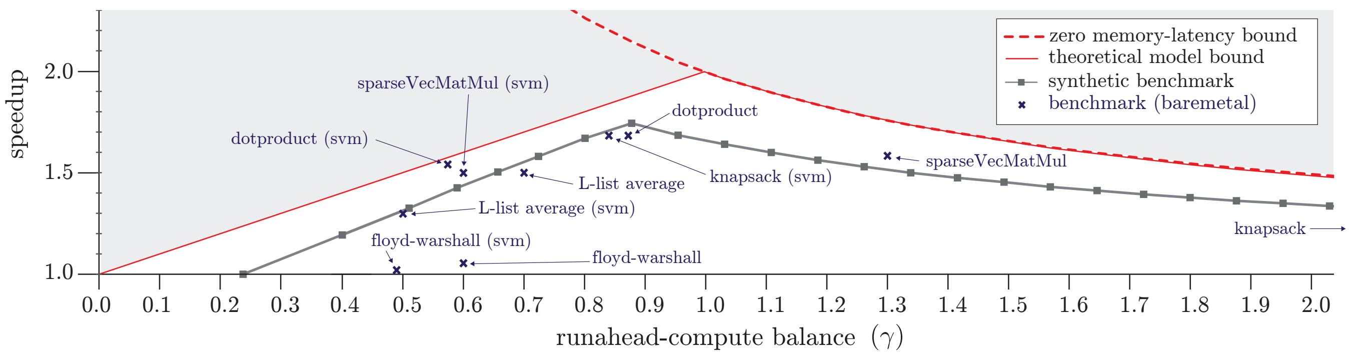


Fig. 7: Graph showing the speedup for the synthetic benchmark as the parameters, α_o, β, and α_r are artificially varied. Also shows the approximate locations for a selection of benchmarks explored in this study.

			baremetal		*Linux*					**Δfmax**
benchmark	**memory**	**version**	**cycles**	**RCB (γ)**	**cycles**	**RCB (γ)**	**LUTs**	**REGs**	**DSPs**	**(MHz)**
sparseVecMatMul	physical	original	5658K	-	4611K	-	6258	6821	8	
	physical	pslice	3573K (**1.59x**)	1.14	2798K (**1.65x**)	0.75	7144 (**14%**)	7843 (**15%**)	8 (**0%**)	-9
	SVM	original	120491K	-	10555K	-	6398	7180	8	
	SVM	pslice	8015K (**1.51x**)	0.6	87962K (**1.20x**)	0.39	7628 (**14%**)	7969 (**11%**)	8 (**0%**)	-8
knapsack	physical	original	2395K	-	2589K	-	5195	6625	0	
	physical	pslice	2106K (**1.13x**)	3.93	2132K (**1.21x**)	1.02	5640 (**9%**)	7736 (**17%**)	0 (**0%**)	0
	SVM	original	4181K	-	6554K	-	5382	6722	0	
	SVM	pslice	2474K (**1.69x**)	0.84	6273K (**1.04x**)	0.36	6110 (**14%**)	7907 (**18%**)	0 (**0%**)	0
floyd-warshall	physical	original	9995K	-	12270K	-	5005	5983	6	
	physical	pslice	9457K (**1.06x**)	0.61	11711K (**1.05x**)	0.48	8427 (**68%**)	9722 (**62%**)	12 (**100%**)	+6
	SVM	original	26916K	-	35349K	-	5156	6586	6	
	SVM	pslice	26932K (**1.02x**)	0.50	30303K (**1.16x**)	0.46	8771 (**70%**)	10810 (**64%**)	12 (**100%**)	-13
L-list average	physical	original	1980K	-	2542K	-	9055	10393	25	
	physical	pslice	1313K (**1.51x**)	0.70	17764K (**1.43x**)	0.52	9404 (**4%**)	11005 (**6%**)	25 (**0%**)	+2
	SVM	original	6076K	-	8097K	-	9236	11109	25	
	SVM	pslice	6218K (**1.30x**)	0.51	6210K (**1.30x**)	0.44	10017(**8%**)	11212 (**1%**)	25 (**0%**)	-11
dotproduct	physical	original	1892K	-	1556K	-	5917	6343	8	
	physical	pslice	1128K (**1.69x**)	0.88	971K (**1.60x**)	1.23	6603 (**12%**)	7091 (**12%**)	8 (**0%**)	-3
	SVM	original	3041129K	-	2362K	-	6084	6428	8	
	SVM	pslice	1928697K (**1.57x**)	0.58	1946K (**1.21x**)	0.51	6828 (**12%**)	7230 (**12%**)	8 (**0%**)	-4
geomean	-	-	**1.38x**	0.81	**1.27x**	0.57	**15%**	**13%**	-	-4

TABLE II: Results for baremetal and Linux versions of each benchmark. **1.*x** = speedup, ***%** = resource overhead

sparseVecMatMul - a `float` sparse vector matrix multiplication where the matrix is in compressed row storage format and stored in global shared memory.

knapsack - a combinatorial optimisation problem, where every item and associated weight (`float`) are stored in an array in global shared memory.

floyd-warshall - a `float` weighted graph shortest path algorithm, where the graph is stored in global shared memory.

L-list average - a linked list traversal where a `float` l-list node element is averaged across the entire list that is stored in global shared memory.

dotproduct - a dotproduct of two `float` vectors stored in global shared memory.

For each benchmark a *shared virtual memory* (SVM) version was also implemented, where instead of using physical memory address space a virtual memory space shared with the host SW on the ARM was used. The SVM implementation was achieved by developing HLS libraries that walk the two level page table structure present in the on-chip ARM system, where the first level has 4096 entries, and the second level has 256 entries. This allows the hardware to map virtual addresses to physical addresses in pages mapped in physical DDR memory shared by SW and HW.

Adding SVM to a benchmark increases its overall memory latency resulting in its RCB (γ) changing. This change is because each access may introduce a page table walk requiring additional memory transactions per load operation. For example consider the dotproduct baremetal benchmark, without SVM its RCB (γ) = 0.88 giving it a speedup `1.69x` close to it's theoretical maximum `1.80x` (derived from Equation 2), however for the SVM implementation the extra memory operations shifts it's RCB (γ) further to the left reducing it to 0.58 and reducing its speedup to `1.57x`.

Applying SVM to a benchmark does not always mean that there is a slowdown, as it depends on how the RCB (γ) of that benchmark moves. For example, knapsack is heavily compute bound with an RCB (γ) of 3.93, giving it a modest speedup of `1.13x`. When SVM is applied, this moves the RCB (γ) to 0.84, much closer to the peak speed up, resulting in a `1.69x` speedup.

The floyd-warshall benchmark appears as an anomaly in the results, as given its RCB ($\gamma = 0.61$) it would expect a higher speedup than its achieved speedup of 1.06x. We believe that is due to the amount of replication required for the pslice: this benchmark has a much higher α_r value than all the other benchmarks, limiting its theoretical maximum speedup (according to Equation 2) to `1.5x`, far lower than the $S_p^{\uparrow}$ for all other benchmarks.

Table II shows detailed results for all benchmarks where there is no operating system (baremetal) and where a Linux operating system is running. On average the presence of Linux increases contention in the memory system, which in turn increases the memory latency for the FPGA circuit generally reducing the RCB (γ) value. Since the majority of baremetal RCB (γ) < 1 adding linux usually results in a reduced speedup compared to baremetal due to increased memory latency reducing the RCB (γ) further. However knapsack has a very high baremetal RCB (γ) of 3.95 and when this benchmark is used with Linux its RCB (γ) moves to 0.95, causing an increase in speedup from 1.13x to 1.21x which is close to its $S_p^{\uparrow}$ (1.28x).

D - Resource Overheads: On average the cost of adding a pslice to a circuit is low (1.15x) relative to the performance gains (1.38x). This could be because what we are replicating in the pslice is address calculation logic, which usually only requires cheap integer arithmetic and control/branching logic. Most benchmarks require less than 20% the resources of the original circuit with the exception of floyd-warshall, where high overheads were due to a floating-point comparison and addition used in a branch decision for which the load had a control dependency.

E - FIFO depth: Figure 8 shows how varying the FIFO depth effects the speedup for both the dotproduct and sparseVecMatMul benchmark (Linux). For sparseVecMatMul we observed that for a FIFO depth of 2 the pslice is prevented from running ahead since its FIFO is full for 11% of the execution time, resulting in the speedup around 1x. However once the FIFO depth was increased to 4 the speedup jumps since the percentage of execution time the FIFO is full drops to a negligible amount (0.01%).

Further increases in the FIFO depth result with little to no increases in performance, giving the same speed up as if the FIFO was infinitely deep (i.e. FIFO is never full). This is because the pslice and the original circuit quickly reach an equilibrium where reads are being produced by the pslice at the same rate as the original circuit is consuming them. The RCB (γ) value of the sparseVecMatMul (Linux) benchmark is 0.75 meaning that $\alpha_r + \beta > \alpha_o$ implying that the original circuit needs to block for some time waiting for the pslice to fetch data. This RCB means that a large FIFO is useless as it will often be empty (this applies to all benchmarks in the region RCB $\gamma < 1.0$).

The RCB (γ) value for dotproduct (Linux) is 1.23 which means that $\alpha_o > \alpha_r + \beta$, i.e. the rate at which reads are consumed is less that the rate at which they are produced, so FIFO depth has a greater impact on performance. However, the benefits of extra FIFO depth are marginal and disappear quickly once the FIFO becomes full. Figure 8 shows this effect, where the line for infinite depth is above the depth 2 (always full), however as the size of the problem grows the difference shrinks from 3.5% to 1%. For depth 16 it can be seen that it starts closely matching the performance of the infinite depth FIFO until the point where it starts to become full around problem size 17, where its performance drops down to the same speed

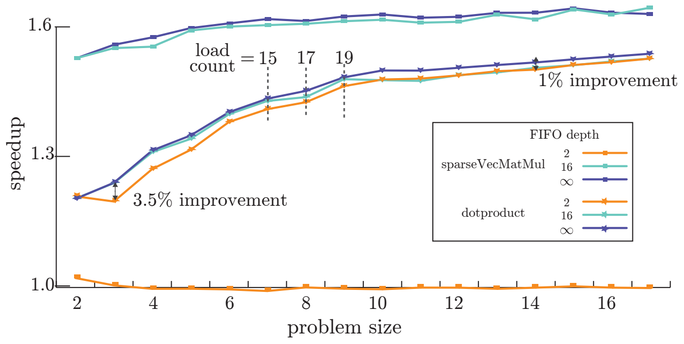


Fig. 8: Graph showing the speedup for the `dotproduct` and `sparseVecMatMul` benchmarks for implementations using different FIFO depths

as the depth 2 case. We conclude that there is little benefit in using a large FIFO depth regardless of RCB (γ) value, for this reason, we have selected a depth of 16 for all benchmarks.

VI. Related Work

	runtime generic	*compile time generic*	*application specific*
hidden from user	LEAP, 2013[8]	**our approach** Cheng & Wawrzynek, 2014[5] Chen & Shu [3]	-
transformation required	VivadoHLS	Zhao et al., 2016[9]	CoRAM++, 2015[7] APMC, 2014[6]

TABLE III: Categorised table of related work

Figure III shows a categorised table of related work aiming at hiding memory latency in HLS, with columns indicating how generic the approach is and rows indicating the amount of user involvement. The default way to attempt to hide memory latency is to try to transform the application into a streaming function. To aid in achieving this Xilinx's VivadoHLS tool provides streaming interface types and DMA engines to convert memory mapped locations into streams. While these tools facilitate hiding memory latency they often require considerable restructuring of the code making them time-consuming to use, and while this technique can be applied to any streaming interface, it is often not possible to turn an irregular application into a streaming one.

Alternatively, LEAP [12] provides a general memory abstraction for FPGA hardware kernels agnostic to the underlying code, where latency insensitive channels are used to communicate with scratchpads and caches. To improve LEAPs performance, stride prefetchers, [8] were added to its memory infrastructure. These prefetchers examine the access pattern of any hardware kernel attached to them and attempt to predict its access pattern, prefetching its guesses. While this provides a generic transparent interface improving performance behind the scenes, it again only works for regular access patterns where items are accessed at fixed offsets. Such an approach would have little effect for irregular applications such as a linked list where pointer chasing is required.

CoRAM++ [7], APMC [6], and Zhao et al [9], all attempt to hide memory latency for complex data structures by providing data structure specific memory-interfaces. In [9] HLS alternatives to the STL data structures are provided where accessing data-structure elements can be performed in parallel, but writing to them is restricted, allowing the designer to convert pre-existing code into a parallel implementation automatically. Similarly, CoRAM++ and APMC provide data-structure specific prefetching. However, they require more manual intervention as control-threads need to be manually configured to fetch from memory in parallel.

All approaches mentioned so far either can't support irregular applications, as it requires in-depth knowledge of the underlying data structures, or require the user to manually use specific mechanisms to indicate what type of data structure they are using. Our approach aims to extract the memory access structure from the underlying source, enabling us to hide memory latency for irregular applications without any hints or extra information from the designer. One approach similar to ours is by Cheng & Wawrzynek [5], where the authors describe a compiler optimisation that attempts to hide memory latency for irregular loop nests by taking memory operations and decoupling them from the compute. However, this approach is limited in the types of applications that it can accelerate since it cannot transform memory operations with dependencies between them, *"a classic example is the pointer chasing in linked list traversal, where the address for the subsequent memory request would not be available until the current load gets its response"*[5]. Such applications are something that our approach **can** handle, and we believe that our approach is compatible with theirs.

The most similar approach to ours is recent work by Chen & Shu [3], which aims to provide prefetching for HLS tools targeting ASIC design. In this work, they combine stride prefetching for regular accesses, with a program slicing based technique for irregular accesses, achieving impressive RTL simulation results.

Program slicing is an established technique in the SW domain and historically is mainly used to aid debugging. More recently program slicing has been utilised in a multithreaded context to precompute parts of threads to aid branch prediction and to warm caches in the memory hierarchy [13]. While these techniques are somewhat effective[14], they often suffer due to the overheads required to spin up a thread, along with utilising entire core.

The work presented in this paper demonstrates that when applying program slicing techniques to HLS tools a more efficient implementation can be achieved as: we only need to generate the hardware required to compute the slice (i.e. not wasting an entire core); we can tightly integrate the pslice and the original thread, passing information directly between them instead of via a cache or branch predictor; also, there is no overhead cost to spin up the pslice in an HLS context, the hardware is pre-configured and waiting to start along with the original circuit.

VII. Conclusion

Prefetching data for irregular applications is challenging as memory addresses are often non-consecutive. Techniques such as using pslices are an attractive solution to this problem, as they are used to calculate the exact read addresses ahead of the original function they are supporting, enabling them to be effectively prefetched. In this work, we bring the idea of pslices to the HLS domain for FPGA designs and present a new HLS compiler pass called RELISH, which can automatically construct a pslice for a given function.

Through a theoretical model, we were able to show situations where a pslice was able to hide memory latency efficiently and where it could not, with theoretical performance between 1x – 2x. This model was then validated through evaluating circuits generated using RELISH, showing that the shape of our model was correct and speedups between 1.05x – 1.69x were achieved. On average the generated pslices require modest resource overheads (1.15x), we believe this is due to pslices only requiring integer arithmetic for address calculation. We also describe the architecture of the tool and detail the architecture of the FIFO based logic coupling to the pslice: experimental results showed us that a small FIFO depth of 16 elements is sufficient to experience the benefits of using a pslice.

References

[1] Weiser, Mark "Program Slicing." in *Proc. 5th international conference on Software engineering*, pp. 439–449, 1981.

[2] Fleming, S. & Thomas, D. "StitchUp: Automatic control flow protection for high level synthesis circuits" in *Proc. DAC*, pp. 1–6, 2016

[3] Chen, T., and Suh, E. "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling"in *Proc. MICRO*, pp. 1–12, 2016.

[4] Lattner, C., and Adve, V "LLVM: A compilation framework for lifelong program analysis & transformation" in *Proc. CGO*, pp. 74, 2004.

[5] Cheng, S., and Wawrzynek, J. "Architectural Synthesis of Computational Pipelines with Decoupled Memory Access." in *Proc. FPT*, pp. 83–90, 2014.

[6] Hussain, T., Palomar, O., Unsal, O., Cristal, A., Ayguade, E., and Valero, M. "Advanced Pattern based Memory Controller for FPGA based HPC applications." in *Proc. HPCS*, pp. 287-294, 2014.

[7] Weisz, G., amd Hoe, J. C. "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing." in *Proc. FPL*, pp. 1-8, 2015.

[8] Yang, H. J., Fleming, K., Adler, M., and Emer, J. "Optimizing under abstraction: Using prefetching to improve FPGA performance." in *Proc. FPL*, pp. 1-8, 2013.

[9] Zhao, R., Liu, G., Srinath, S., Batten, C., and Zhang, Z. "Improving High-Level Synthesis with Decoupled Data Structure Optimization." in *Proc. DAC*,p. 137, 2016

[10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[11] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html

[12] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. S. Emer, "LEAP scratchpads: automatic memory and cache management for reconfigurable logic," in *Proc. FPGA*, pp. 25–28, 2011.

[13] Islam, A., Xin, T., Vijayalakshmi, S., Ioana, B., and Andreas, M."Self-contained, accurate precomputation prefetching" in *Proc. MICRO*, pp. 153–165, 2015.

[14] Zhang, J., Zhimin, G., Yan, H., Ninghan, Z., and Xiaohan, H. "Helper Thread Prefetching Control Framework on Chip Multi-processor" in *International Journal of Parallel Programming*, pp. 180–202, 2015.