Custom-Sized Caches in Application-Specific Memory Hierarchies

Felix Winterstein*, Kermin Fleming†, Hsin-Jung Yang‡, John Wickerson*, George Constantinides*
*Department of Electrical and Electronic Engineering, Imperial College London
†SSG, Intel Corporation
‡CSAIL, Massachusetts Institute of Technology
Email: *{f.winterstein12, j.wickerson, g.constantinides}@imperial.ac.uk, †kermin.fleming@intel.com, ‡hjyang@csail.mit.edu

***Abstract*—Developing FPGA implementations with an input specification in a high-level programming language such as C/C++ or OpenCL allows for a substantially shortened design cycle compared to a design entry at register transfer level. This work targets high-level synthesis (HLS) implementations that process large amounts of data and therefore require access to an off-chip memory. We leverage the customizability of the FPGA on-chip memory to automatically construct a multi-cache architecture in order to enhance the performance of the interface between parallel functional units of the HLS core and an external memory. Our focus is on automatic cache sizing. Firstly, our technique determines and uses up unused left-over block RAM resources for the construction of on-chip caches. Secondly, we devise a high-level cache performance estimation based on the memory access trace of the program. We use this memory trace to find a heterogeneous configuration of cache sizes, tailored to the application's memory access characteristic, that maximizes the performance of the multi-cache system subject to an on-chip memory resource constraint. We evaluate our technique with three benchmark implementations on an FPGA board and obtain a reduction in execution latency of up to $2\times$ ($1.5\times$ on average) when compared to a one-size-fits-all cache sizing. We also quantify the impact of our automatically generated cache system on the overall energy consumption of the implementation.**

I. Introduction

Recent efforts on high-level synthesis (HLS) flows greatly alleviate the difficulty of designing an FPGA implementation at register transfer level (RTL). Generating RTL designs from programs written in high-level languages such as C/C++ or OpenCL can substantially reduce the design time of FPGA implementations. HLS users are provided with a rich set of directives which guide the RTL generation and trigger various compiler optimizations, which allow them to make use of the customizability of FPGA designs. Modern HLS compilers can deliver a quality of results, in terms of resource usage and execution speed of the synthesized circuits, that is becoming acceptable for more and more users.

This paper focuses on the integration of HLS kernels into an FPGA platform. In particular, we focus on data- and communication-intensive HLS designs that require access to an off-chip memory system because the application data do not fit into the limited on-chip storage. In this case, access to board- or host-level memory with substantially lower bandwidth can easily become the performance bottleneck. A common solution to this problem is to insert on-chip caches or specialized buffers into the interface to an off-chip memory hierarchy to enhance performance. The customizability of the on-chip memory, a key benefit of FPGAs, gives enormous freedom to a compiler to construct a tailor-made cache architecture underneath a uniform interface layer provided to the user core. Yet, the design space is large and finding optimal choices for the memory hierarchy parameters is time-consuming and requires significant domain expertise on the part of the programmer. Recent efforts aim to automate the optimization of the memory hierarchy [1]–[6]. A common approach is to compile multiple parallel on-chip caches in order to leverage the memory-level parallelism of FPGAs. Much of the related work focuses on the parallelism of such a multi-cache architecture [1]–[3], but the problem of optimal sizing of these caches is unaddressed. In this work, we focus on optimizing the capacity of the caches produced by these automated flows.

In applications with large memory footprints, the bulk of the data necessarily resides off-chip. In these cases, the HLS core often keeps only small data structures on-chip. Consequently, the amount of on-chip block random access memory (RAM) used by the core is often smaller than the amount of the block RAM available. We implement a cache compilation flow that automatically uses up the left-over block RAM and enlarges the on-chip caches. Secondly, the size of each cache is set individually in order to obtain a size distribution across the parallel caches that is optimal for the memory access pattern of a particular application. For example, the accesses to some data structures of an application have good locality and increasing the cache size improves performance. On the other hand, some memory access traces have very little locality or access small data structures, and scaling up the cache beyond a certain size is of no use. In such a case, an application-specific cache sizing will very likely have superior performance compared to a one-size-fits-all solution.

It is important to note that our technique does not rely on successive synthesis and place-and-route cycles, but instead estimates the cache performance for different sizes with a pre-RTL, dynamic program analysis of the input code to an HLS tool. Our approach relies on a prediction of the performance of each cache from the applications reference stream, and finds a size configuration that maximizes the aggregate performance subject to a resource constraint. Although determining statically the cache size requirements

978-1-4673-9091-0/15/$31.00 © 2015 IEEE

and hence the size of the data structures is possible in some corner cases, we adopt here a run-time profiling approach for capturing the memory reference trace in order to ensure wide applicability. Especially in heap-manipulating programs, the absolute data structure size is often unknown at compile time. Our dynamic analysis can handle this type of program at the expense of relying on a representative input data set provided by the user. In summary, our contributions are:

- We implement an automated size scaling of private, direct-mapped on-chip caches with a fixed line width that uses spare block RAM resources (Section III-A).
- We develop a cache hit rate estimator based on the memory reference trace of the program under test. The original code is instrumented with profiling instructions for gathering the memory reference trace. The code instrumentation is made in the LLVM intermediate representation of the program, which allows us to support HLS flows based on LLVM. We use Vivado HLS as an exemplary tool in this paper (Section III-B).
- We generate individual sizing information for the multi-cache system. We cast the cache size assignment into a Multiple-Choice Knapsack Problem in order to find the best distribution for a user-provided memory access pattern of a particular application (Section III-C).
- We build physical implementations of the HLS cores and the memory hierarchy on a Xilinx VC707 FPGA board to perform end-to-end evaluations (Section IV).
- We demonstrate our technique using three benchmarks running on the FPGA board. We evaluate the accuracy of the hit rate estimation by comparison with measured hit rates. We show up to 2× improvements on the overall execution time compared to a one-size-fits-all cache sizing. We also characterize the impact of our cache insertion in terms of power and energy consumption (Section V).

II. Motivating Example

Although our technique can be applied to arbitrary programs, we primarily target programs that involve pointer chasing because of their sensitivity to the memory access latency and because their irregular access pattern results in long response times of an external synchronous dynamic RAM (SDRAM). Listing 1 shows an excerpt of a heap-manipulating program. The function `reflectTree` traverses a pointer-linked tree data structure (nodes of type `treeNode`). The depth-first traversal is managed with a second data structure implemented as a linked list. The nodes of the list (type `forestNode`) contain a pointer to a sub-tree and a pointer to the next node in the list. The head of the list is manipulated by the auxiliary functions `push` and `pop`. The tree has been built up by another function of the same program.

Our static program analysis in [3] can determine that the tree and the linked list each can be split into sub-partitions which reside in disjoint address spaces in the heap memory. Furthermore, the source code is transformed such that 1) the modified code passes through a standard HLS tool

```
1  //main traversal function
2  void reflectTree(treeNode *root) {
3    forestNode *s = push(root, NULL);
4    while (s != NULL) {
5      treeNode *u;
6      s = pop(&u, s);
7      treeNode *left = u->left;
8      treeNode *right = u->right;
9      u->left = right;
10     u->right = left;
11     if  (u->left!=NULL && u->right!=NULL)  {
12       s = push(u->left, s);
13       s = push(u->right, s);
14     }
15   }
16 }
17 //auxiliary function push (add new list head)
18 forestNode* push(treeNode *u, forestNode *s){
19   forestNode *t = new forestNode;
20   t->u = u; t->n = s;
21   return t;
22 }
23 //auxiliary function pop (delete list head)
24 forestNode* pop(treeNode **u, forestNode *s){
25   *u = s->u;
26   forestNode *t = s->n;
27   delete s; return t;
28 }
```

Listing 1: C-like pseudo code for a tree reflection.

(substituting heap accesses with array accesses), and 2) the `while`-loop in line 4 is split into parallel sub-loops that process their own, private heap partition independently to parallelize the application. Finally, since the heap resides in a large off-chip memory by default, the notion of disjoint, non-overlapping address regions allows the tool to instantiate parallel private on-chip caches for each sub-partition (of each data structure type). All caches have a fixed size of 32 KB.

Assuming we have only run the transformation of pointer references and cache insertion without asking for additional parallelization, the hardware implementation has a private cache for `forestNode` and `treeNode`. The RTL design for the modified source code is generated with an HLS tool, for example Xilinx Vivado HLS, which also provides information of the block RAM resources consumed by the HLS core itself. In this case, the core uses 112 36kbits RAM blocks which leaves 918 left-over blocks in a Virtex 7 device (xc7vx485tffg1761-2) to be used by the platform surrounding the HLS core. With a conservative 40%-margin, 550 RAM blocks (2200 KB[1]) can be repurposed as cache memories.

Our technique then estimates the performance of the caches from the memory reference trace, which is obtained from running the HLS input program with a representative input data set provided by the user. The reference stream, together with the knowledge of the cache type (direct-mapped, set-associative, fully-associative) allows us to model the aggregate hit rate of the multi-cache system. For $K = 2$ private caches as in this example, there is no interaction between the caches and the aggregate hit rate is given by:

$$\eta = \frac{\sum_{i=0}^{K-1} h_i(B_i)}{\sum_{i=0}^{K-1} t_i}, \tag{1}$$

[1]32 kbits in a Xilinx 36K-RAM block can be used to store user data

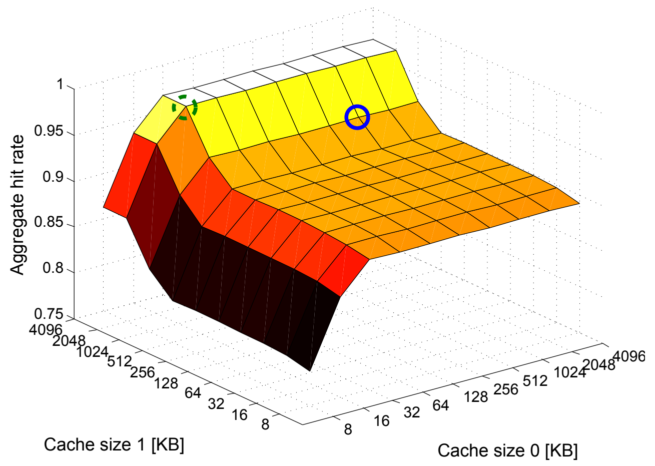

Fig. 1: Aggregate hit rate estimate.

where h_i is the number of hits in cache i of size B_i, t_i is the total number of accesses to cache i. Fig. 1 shows the aggregate hit rate for the two direct-mapped caches over different size configurations. The design space spans hit rates from 79% to 97%. The hit rate of cache 0 (for data of type `forestNode`) reaches its maximum at a size of 32 KB and then plateaus. The reason for the steep improvement with low sizes and early saturation is the high locality of the memory accesses made to the stack-like linked list and the fact that just 32 KB of cache memory is sufficient to keep the entire data structure on-chip. For tree nodes (cache 1), a 2 MB cache is needed to fit all tree data. Clearly, spending the same amount of memory resources on both caches is sub-optimal.

The advantage of our technique over a one-size-fits-all cache scaling becomes obvious when we take the memory resource constraint of 2200 KB into account. With a fixed size for all caches, on this grid, we could implement caches with a maximal capacity of 1024 KB each, which corresponds to the point marked with the solid-line blue circle in Fig. 1. A cache sizing tailored to the access pattern of the application allows us to decide that a size of 32 KB for cache 0 and 2048 KB for cache 1 maximizes the hit rate while still satisfying the resource constraint. This design point is marked with the dashed green circle in Fig. 1. In general, programs will use more than two parallel caches, and the disparity between fixed-size and application specific cache sizing will be larger.

Replacing a fixed-size scaling with a specific size distribution relies on the ability to 1) predict the performance of each cache from the application's reference stream, and 2) to find a cache size configuration that maximizes the aggregate performance subject to a memory resource constraint. The next section describes how both problems are addressed.

III. Cache Sizing

Our cache sizing flow has three components: 1) It first determines unused block RAM resources, which requires an estimation of the memory resources used by the HLS core itself. 2) We predict the hit/miss counts of each cache for different sizes. 3) The amount of spare block RAM and the cache performance estimates are combined into an optimization problem which finds a variable size configuration in the multi-cache system that maximizes the aggregate hit rate.

A. On-chip Memory Utilization Estimation

We obtain high level estimates of the block RAM consumption from the HLS tool to determine the left-over RAM resources. Here, we use Vivado HLS, which provides estimates of the number of LUTs, flip-flops, DSP slices and RAM blocks consumed by the HLS core. Compared to LUTs, flip-flops and DSP slices, the predicted amount of memory is relatively accurate. The only cases where the high-level prediction deviates from the implementation post placement and routing were observed when the down-stream RTL synthesis tool performed bit truncations that affected operands stored in memory. However, in these cases, the high-level estimate is always higher than the actual usage, which results in a slightly over-conservative but safe estimate. A potential clock rate degradation due to large on-chip RAMs is alleviated with memory banking in combination with pipeline buffers [7].

B. Cache Performance Estimation

We build our sizing technique on top of the multi-cache generator in [3], which primarily targets heap-manipulating programs. For each heap partition and heap-allocated data type, this method instantiates a bus interface in the HLS input code. All buses are later routed through an arbiter to an external memory as we describe in Section IV. We instrument the transformed program with profiling instructions that fill trace buffers, which maintain the memory reference trace for each bus interface. We expect the user to provide a representative input data set for the profiling run. Hence, we may miss corner cases with this dynamic program analysis. However, since cache size is only a performance-related parameter, the functional correctness of the optimization is not compromised. The trace buffers are empty at program start-up. On each access to external memory in the program, the instrumentation code adds the memory address. In this way, we build up reference streams of length M_i:

$$\{\, a_{0,i},\ \ldots,\ a_{M_i-1,i} \,\}, \tag{2}$$

where i is the index of the memory interface. The memory is divided up into *blocks*, some of which will have copies in the cache. The block width L is equal to the cache line size. For a data width smaller than L the *block reference streams*

$$\{\, \lfloor \tfrac{a_{0,i}}{L} \rfloor,\ \ldots,\ \lfloor \tfrac{a_{M_i-1,i}}{L} \rfloor \,\} \tag{3}$$

give us the dynamic trace of memory accesses at the granularity of the cache line size. The cache line size is a fixed parameter in our analysis. If the user data width is larger, a cache access is split into multiple sequential chunks in our implementation. We model this by expanding the block reference stream (3) accordingly in a post-processing step.

The cache size remains the only variable parameter in the hit rate estimation. Other parameters such as associativity and support for disjoint/shared memory accesses are fixed but must be taken into account. We focus on private caches, where no interaction between caches occurs. The hit rate of fully-associative caches can be precisely determined using the *stack distance* metric [8]–[11], which counts the number of unique references 'between' accesses to the same address. A cache with B lines then filters out references with stack distance larger than B. The stack distance distribution of

Algorithm 1 Hit rate of a private, direct-mapped cache.

```
1:  Input:
2:  Block reference stream $\mathcal{S}$
3:  Number of cache lines $B$
4:  Output:
5:  Miss count $n_{\text{miss}}$
6:  Hit count $n_{\text{hit}}$
7:  function ESTIMATE($\mathcal{S}$)
8:      $\mathcal{S}_{\text{u}} \leftarrow \text{unique}(\mathcal{S})$    ▷ keep unique block references
9:      $n_{\text{miss}}, n_{\text{hit}} \leftarrow 0$
10:     for all $r \in \mathcal{S}_u$ do
11:         $\mathcal{I} \leftarrow \text{findAll}(\mathcal{S} = r)$    ▷ get indices of entries equal to $r$
12:         $c \leftarrow r \bmod B$    ▷ cache line accessed by $r$
13:         $n_{\text{miss}} \leftarrow n_{\text{miss}} + 1$    ▷ first access is always a cold miss
14:         for $j = 1 \ldots \text{length}(\mathcal{I}) - 1$ do    ▷ loop over remaining accesses
15:             $\mathcal{R}' \leftarrow \mathcal{S}(\mathcal{I}(j-1)+1 : \mathcal{I}(j)-1)$    ▷ intervening refs
16:             $\mathcal{C}' \leftarrow \mathcal{R}' \bmod B$    ▷ intervening cache line refs
17:             if $\text{find}(\mathcal{C}' = c) = \emptyset$ then
18:                 $n_{\text{hit}} \leftarrow n_{\text{hit}} + 1$    ▷ hit
19:             else
20:                 $n_{\text{miss}} \leftarrow n_{\text{miss}} + 1$    ▷ conflict miss
21:             end if
22:         end for
23:     end for
24:     return $n_{\text{miss}}, n_{\text{hit}}$
25: end function
```

a reference stream allows us to count cold misses (cache misses due to empty cache at program start-up) and capacity misses (misses due to line eviction because the cache is full) in fully-associative caches. In lower-associativity caches, additional conflict misses occur (eviction due to intervening references although the cache is not full) which the stack distance approach can only approximate [10], [11]. The prediction accuracy worsens with decreasing associativity.

Because we target direct-mapped caches and because our goal is an accurate prediction, we devise a precise hit rate determination for direct-mapped caches. For each reference r and the previous reference r' to the same block address, we examine the intervening references made between r' and r. A conflict miss occurs if at least one intervening reference accesses the same cache line, which is determined with a modulo operation using the cache size B as divisor. Algorithm 1 shows Matlab-like pseudo code of the hit rate estimator for direct-mapped caches of size B. It predicts the the number of hits (n_{hit}) and misses (n_{miss}) of the cache dependent on its size, which allows us to compare the performance of cached memory interfaces with different block reference streams) relative to the other caches and select a configuration of cache sizes that maximizes the aggregate hit rate.

C. Optimization Strategy

Our compiler generates K caches as described above. With Algorithm 1, we can estimate the performance of each independent cache $h_i(B)$, $i = 0 \ldots K-1$ once we have obtained the corresponding reference streams. We assign different sizes to the caches in such a way that the aggregate hit rate is maximized. To this end, we assign to each cache a set of N cache sizes $\mathcal{B}_i = \{B_0, B_1, \ldots, B_{N-1}\}$ and compute the hit rate relative to the total number of accesses for each size. We cast the search for the best size assignment for each cache into an optimization problem and define the following variables:

$p_{ij} = h_i(B_j)$	the profit (hit rate of cache i)
$w_{ij} = bram_i(B_j)$	the cost (block RAM consumption of cache i)
C	the global constraint on the available block RAM resources
$x_{ij} \in \{0, 1\}$	a binary variable,

where $i = 0 \ldots K-1$ iterates over caches and $j = 0 \ldots N-1$ iterates of cache sizes. We phrase the maximization problem as a *Multiple-Choice Knapsack Problem* (MCKP) [12] as follows:

$$\begin{aligned} &\text{maximize} && \textstyle\sum_{i=0}^{K-1} \sum_{j=0}^{N-1} p_{ij} x_{ij} \\ &\text{subject to} && \textstyle\sum_{i=0}^{K-1} \sum_{j=0}^{N-1} w_{ij} x_{ij} \le C \\ &\text{and} && \textstyle\sum_{j=0}^{N-1} x_{ij} = 1, \quad i = 0 \ldots K-1 \end{aligned} \tag{4}$$

The objective in (4) maximizes the aggregate hit rate of K caches. The first constraint enforces memory resource limits and the second constraint ensures that, for each cache, exactly one size from the set $\mathcal{B}_i$ is selected by the algorithm. We solve the Knapsack problem with an algorithm by Pisinger *et al.* [12] based on dynamic programming. The next section describes the code generation and transformations before and after cache sizing.

IV. Code Generation

Most state-of-the-art HLS tools, such as Vivado HLS, LegUp [13] and ROCCC [14], compile the input code into the LLVM intermediate representation (IR) prior to RTL generation and perform code analysis and transformations at IR-level. Our code instrumentations and the profiling run are performed at IR-level. The instrumentation code is stripped out before the source code is passed to the RTL generation.

We primarily target heap-manipulating programs. After heap partitioning by the up-stream static program analysis [3], the HLS core has a bus interface for each partition and heap-allocated data structure type because the heap resides in off-chip memory by default. Dereferencing of heap-directed pointers is substituted using an auxiliary static pointer variable (`_aux35`) as shown in Listings 2 and 3 for the instruction `t->u = u`. All heap-directed pointers in the original code are turned into integer variables (`i32`), which ensures synthesizability by standard HLS tools and easy address trace profiling. Additional instrumentation code is added after the memory access in line 7 of Listing 3 to record a write access to the base address at `%tmp3` plus the field offset.

The HLS core is embedded in the LEAP platform [15]. LEAP provides an abstraction layer which provides unified interfaces to the user application. Underneath this layer, LEAP implements drivers and communicates with the FPGA-specific components and I/O systems. In particular, LEAP Scratchpads [16], [17], LEAP's memory service, build a memory hierarchy underneath a simple request/response protocol to which the memory bus interfaces of our HLS core connect. LEAP

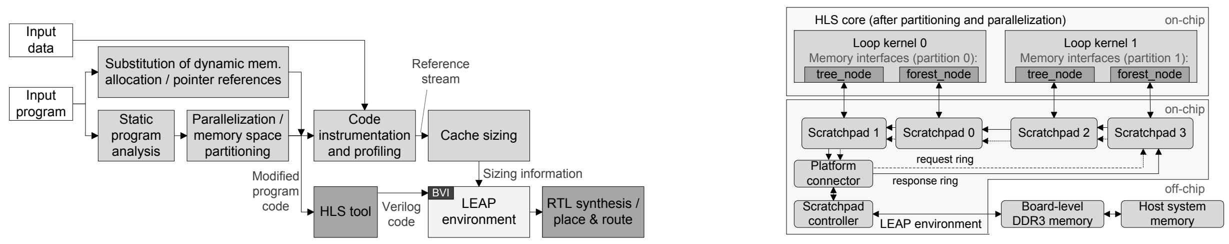

Fig. 2: Synthesis flow (left) and the synthesized architecture for the parallelized **Reflect tree** example in Section II (right).

```
// t->u = u; Original LLVM code
%tmp2 = load %struct.tree_node_t** %u
%tmp3 = load %struct.forestNode** %t
%tmp4 = getelementptr inbounds %struct.forestNode*
    %tmp3, i32 0, i32 0
store %struct.treeNode* %tmp2, %struct.treeNode**
    %tmp4
```

Listing 2: Original LLVM IR of the statement `t->u = u`.

```
// t->u = u; Transformed LLVM code
%tmp2 = load %struct.tree_node_t** %u
%tmp3 = load i32* %t
%_aux35 = call %struct.forestNode*
    @auxMakePointer_0(%struct.forestNode*
    getelementptr inbounds ([4294967296 x %struct.
    forestNode]* @heap_partition0, i32 0, i32 0),
    i32 %tmp3)
%tmp4 = getelementptr inbounds %struct.forestNode*
    %_aux35, i32 0, i32 0
store i32 %tmp2, i32* %tmp4
// additional instrumentation code to record a
    write access to address %tmp3 + field offset
```

Listing 3: Transformed LLVM IR.

Scratchpads provide easy access to the board-level SDRAM and host-level main memory if the memory accesses made by the FPGA application go beyond the capacity of board-level RAM. Optionally, each scratchpad can contain an on-chip, direct-mapped L1 cache, which is implemented with block RAM resources. LEAP provides private scratchpads [16] which access private, disjoint address spaces. In the case of overlapping, *i.e.* shared memory regions, scratchpads with coherent caches [17] can be instantiated, which are connected through an on-chip coherency network. The generated RTL code of the HLS core connects to the Bluespec-based LEAP framework through a thin Verilog/Bluespec wrapper. The latter uses Bluespec's `import BVI` functionality. Fig. 2 shows the complete synthesis flow and the synthesized architecture for the parallelized version of the **Reflect tree** example from Section II (two parallel loop kernels 0 and 1).

V. Evaluation

We evaluate our technique using three HLS implementations with irregular memory access patterns. Although our cache sizing can be applied to any program requiring access to off-chip memory, we focus on heap-manipulating programs that involve pointer chasing.

Merger: The application builds up four linked lists in parallel by performing a sorted insertion of the input values. After the build phase, the heads of the lists are repeatedly compared. The head with the smallest key is deleted and its data entry appears at the output of the core, resulting in a sorted sequence of all input values. The program traverses the lists for each insertion and is therefore very sensitive to memory access time. The implementation uses eight parallel caches.

TABLE I: High-level block RAM estimation accuracy (results in 36K-RAM blocks).

Design component	Estimate	Post-PAR
Merger (8 scratchpads)		
HLS core	512	512
Interface wrapper	12	12
Scratchpad internal FIFOs	12	12
LEAP platform (without scratchpads, fixed)	50.5	50.5
Total consumption without caches	586.5	586.5
Unused left-over blocks	340.5	340.5
Reflect tree (4 scratchpads)		
HLS core	208	158
Interface wrapper	21	21
Scratchpad internal FIFOs	19	19
LEAP platform (without scratchpads, fixed)	50.5	50.5
Total consumption without caches	298.5	248.5
Unused left-over blocks	628.5	678.5
Filter (6 scratchpads)		
HLS core	275	241
Interface wrapper	32	32
Scratchpad internal FIFOs	24	24
LEAP platform (without scratchpads, fixed)	50.5	50.5
Total consumption without caches	381.5	347.5
Unused left-over blocks	545.5	579.5

Reflect tree: This is a more complex version of the motivating example in Section II. The application traverses a binary tree and recursively swaps the left and right child pointer of some nodes to produce a partially mirrored tree. It additionally performs some computation at each node and updates the data fields of the tree nodes. The implementation consists of a tree building and a tree traversal phase. The HLS core consists of two parallel units and uses two caches for tree nodes and two caches for the linked lists (`forest`).

Filter: This is an efficient k-means clustering algorithm, which partitions a 3D-data set into k clusters. The application first builds a kd-tree by recursively sub-dividing the data set. In a second step, the tree is traversed in post-order fashion to store the sum of sub-tree data in each node. The actual clustering step repeatedly traverses the tree and iteratively improves the cluster assignment until it converges to a locally optimal solution. The HLS core consists of two parallel units and uses six caches, two for tree nodes, two for forest nodes, and two for intermediate cluster centers.

We use Vivado HLS 2014.1 for high-level synthesis of our benchmarks. The LEAP framework around the HLS core is built with the Bluespec 2014-07-A compiler. We implement the FPGA designs in a hybrid flow using Synopsys Synplify Premier 2014.03.1 for RTL synthesis and

TABLE II: Cache hit/miss count estimation for **Reflect tree**.

Cache size	h_{meas}	h_{est}	error	$h_{\text{est}}^{\text{SD}}$	error$^{\text{SD}}$
Cache 0					
1024	86.46%	86.46%	0.00%	84.71%	−2.06%
8192	87.44%	87.44%	0.00%	87.12%	−0.36%
32768	87.49%	87.49%	0.00%	87.42%	−0.08%
65536	87.50%	87.50%	0.00%	87.87%	0.42%
262144	95.83%	95.83%	0.00%	91.43%	−4.82%
Cache 1					
1024	86.35%	86.35%	0.00%	84.44%	−2.26%
8192	87.03%	87.03%	0.00%	87.12%	0.11%
32768	95.68%	95.68%	0.00%	91.04%	−5.09%
65536	95.68%	95.68%	0.00%	92.94%	−2.94%
262144	95.68%	95.68%	0.00%	94.90%	−0.82%

Vivado 2014.4 for placement and routing. All implementations run on a Xilinx VC707 board containing a Virtex 7 FPGA (xc7vx485tffg1761-2) with 1030 36K-RAM blocks.

A. *Left-Over Block RAM Estimation*

Our automatic cache scaling relies on the ability to estimate the amount of block RAM used by the HLS implementation for core-internal storage. Once the tool decided which variables in the code go into block RAM, a conservative estimate can be easily made. Vivado HLS, for example, provides such an estimate after RTL generation. Table I compares the high-level block RAM estimation of 36K-RAM blocks with post placement-and-routing (PAR) results (18K-blocks count as 0.5 36K-blocks). Additional block RAM is used in FIFOs of the wrappers connecting HLS bus interfaces to LEAP scratchpad ports. Similarly, our scratchpad interfaces (scratchpads without caches) contain some FIFOs as well. The RAM usage of these FIFOs can be precisely determined from the Verilog/BSV code. The LEAP-based framework uses a fixed amount of RAM. The only uncertainty are the estimates made by the HLS tool, but these are always higher than the post-PAR consumption. because of bit truncations made by the RTL synthesis tool. We also include a 10% security margin in the left-over portion that will be used for the cache implementations.

B. *Validating the Cache Performance Estimation*

We validate or cache model with measurements of the actual hit/miss rates. LEAP Scratchpads collect the number of hits and misses for each cache during execution of the application. Table II compares the measured individual hit rate h_{meas} for different cache sizes with the estimated values h_{est} from Algorithm 1. The hit rates are calculated with $h = n_{\text{hit}}/(n_{\text{hit}} + n_{\text{miss}})$ and the cache sizes are given in terms of 64bit lines. We also include the relative error. Additionally, we compare the stack distance-based approximation in [10] ($h_{\text{est}}^{\text{SD}}$, error$^{\text{SD}}$) with our estimator. Due to space limitations, we show results only for two caches of the **Reflect tree** benchmark. Our estimation matches exactly the measured hit/miss counts, *i.e.* Algorithm 1 models our direct-mapped caches perfectly. The approximation by Brehob and Enbody [10] tends to underestimate the hit rate of direct-mapped caches, an observation also made in [10].

C. *Latency and Resource Utilization*

Our technique improves the aggregate hit rate of the multi-cache architecture. The following results show the impact of

TABLE III: Latency and resource utilization.

Case	LUTs	FFs	DSPs	BRAMs	Hit rate	Lat. /ms	S
Merger (8 scratchpads, 200000 random input values)							
1	67072	66502	19	586.5	0	7.64	1
2	90840	88200	38	634.5	0.05	7.91	0.97
3	92735	88550	38	858.5	0.79	3.20	2.39
4	91945	88377	39	874.5	0.99	1.54	4.95
Reflect tree (4 scratchpads, 36863 tree nodes)							
1	73915	76594	57	248.5	0	345.6	1
2	84298	82700	72	278.5	0.91	145.5	2.38
3	84824	82828	71	862.5	0.95	123.3	2.80
4	84860	82826	70	944.5	0.99	100.3	3.44
Filter (6 scratchpads, 24575 kd-tree nodes, k = 128 clusters)							
1	90570	91272	97	347.5	0	598.7	1
2	106765	102856	116	383.5	0.93	231.9	2.58
3	110363	110030	116	971.5	0.97	235.5	2.54
4	106850	102965	112	829.5	0.99	221.8	2.70

the cache scaling on the overall execution latency and on the FPGA resource usage. All results are obtained from a physical implementation on the VC707 board. We compare four cases:

1) An implementation without any caches,
2) An implementation with a small fixed cache size of 1024 lines,
3) An implementation with a fixed size for all caches but scaled up to the maximally possible size,
4) A variably-sized multi-cache system as delivered by our technique in Section III.

The clock frequency target is set to 100 MHz in all cases and all designs meet this clock constraint. All caches have a line width of 64 bits. Table III shows the timing as well as resource utilization. It shows the utilization of look-up tables (LUTs), flip-flops (FFs), DSP slices (DSPs) and 36K-RAM blocks (BRAMs). We also show the aggregate hit rate (measured) and the execution latency. The latency is normalized depending on the application: the latency per input value for **Merger**, the overall latency of a completed tree traversal for **Reflect tree**, and the latency per clustering iteration for **Filter**. We compare the speed-up S with respect to the base case (case 1).

In addition to more BRAM, we observe a sudden increase in LUT, FF and DSP utilization once caches are included in the scratchpads. LUTs and FFs increase only marginally when scaling the caches up, leaving the BRAM usage as the limiting factor. The hit rate and latency improvements for **Merger** are substantial and grow steadily with larger cache sizes. There is a significant asymmetry between the linked lists in the application and the large improvement of the variable sizing over a fixed sizing (cases 3 and 4) is due to the fact that larger caches support longer lists.

For the tree-based benchmarks, we see a different characteristic of the latency improvement. Even small caches lift the aggregate hit rate above 90%. This reflects the behavior in Fig. 1: the `forestNode` data structures, with a stack-like access pattern, are very small (but heavily accessed) in the average case and a small cache is sufficient to keep all data on-chip. Consequently, the optimization algorithm in Section III-C opts to use more memory resources for the large tree structure. For **Reflect tree**, this improves the aggregate

TABLE IV: Power and energy measurements.

Case	P_{FPGA} /W	P_{SDRAM} /W	E_{FPGA} /mJ	E_{SDRAM} /mJ	E_{total} /mJ	R
Merger (8 scratchpads, 200000 random input values)						
1	1.8	1.1	13.6	8.4	22.0	1
2	2.1	1.1	16.9	8.5	25.4	0.87
3	2.6	1.0	8.2	3.3	11.5	1.92
4	2.6	1.0	4.1	1.6	5.6	3.93
Reflect tree (4 scratchpads, 36863 tree nodes)						
1	1.9	1.2	655.2	401.9	1060.5	1
2	2.0	1.2	289.5	179.2	471.1	2.3
3	3.4	1.2	420.3	144.1	564.4	1.88
4	3.8	1.0	376.5	101.1	477.6	2.22
Filter (6 scratchpads, 24575 kd-tree nodes, $k = 128$ clusters)						
1	2.0	1.3	1208.2	768.7	1976.9	1
2	2.1	1.0	494.5	234.4	728.8	2.71
3	3.6	1.0	855.4	242.9	1098.2	1.80
4	3.4	1.0	745.7	227.8	973.5	2.03

hit rate by 5% compared to a homogeneous maximum sizing. Although the hit rates for **Filter** and **Reflect tree** are similar, the latency improvement from cache scaling for **Filter** is small. This is mainly due to high core-internal computation between memory accesses, which makes the effect of a shorter access time to the tree data less significant.

D. Energy Consumption

We quantify the impact of our cache insertion and scaling on the overall energy consumption. To this end, we measure the instantaneous power consumption of the FPGA and the board-level SDRAM while the applications are running. We collect power figures for three out of the 12 power rails on the VC707: `VCCINTFPGA` is the main supply of the FPGA and `VCCBRAM` is an additional block RAM supply. We combine both to obtain the main supply of the FPGA. The third rail is `VCC1V5` is a supply of the SDRAM. All other rails do not change notably their power levels during execution of our applications. We integrate power over the three latencies defined in the previous section: We show the energy per input value for **Merger**, the energy per completed tree traversal for **Reflect tree** and the energy per clustering iteration for **Filter**. Table IV shows the main energy consumption of the FPGA (E_{FPGA}), the energy attributed to the SDRAM (E_{SDRAM}) and the total energy for the four cases above. We also show the energy improvement R compared to the base case (case 1). The instantaneous power consumption is steady during the execution, so Table IV also shows the mean power consumptions P_{FPGA} and P_{SDRAM}.

Including caches always comes along with an increased power consumption of the FPGA. For large caches, the extra power consumption is significant (up to 100%). The latency reduction must be large enough to counter this effect and improve E_{FPGA} and E_{total}. Large caches always improve the energy consumption with respect to a cacheless memory interface in our implementations. In all benchmarks, the application-specific cache sizing outperforms fixed sizing in terms of energy reduction. Interestingly, small caches (case 2) in the **Reflect tree** and **Filter** benchmark have the best performance in terms of energy. The trade-offs when optimizing for energy instead of hit rate are different.

VI. Related Work

Much of the related work on FPGA-targeted caches has focused on efficient implementations of the cache micro-architecture in FPGAs [7], [16]–[19]. FCache [18] as well as an extension of LEAP Scratchpads in [17] target the micro-architecture of coherency mechanisms for shared memory systems in FPGAs. In the current work, we use LEAP Scratchpads as building blocks for constructing the multi-cache system. As we scale up the caches to large sizes, the overall clock frequency can eventually be degraded due to the large monolithic cache memories. We rely on a recent mitigation of this problem [7] which implements banked block RAMs for the cache and allows us to maintain the clock rate.

Recent work has also explored the design space of the cache micro-architecture [7], [19], [20]. Matthews *et al.* [20] explore the efficiency in terms of speed-up versus area increase of parallel coherent L1 caches with respect to size, associativity and replacement rule in an FPGA-based soft multi-core processor. Similarly, Choi *et al.* [19] compare different configurations of cache size, line size and associativity of shared on-chip caches, in addition to two approaches for increasing the number of access ports of the shared cache. The goal in this work is different. We infer cost/performance estimates prior to implementation and devise an automated cache system construction for a given application instead of exploring the cache micro-architecture.

There is a substantial body of work on on-chip buffer generation using the *polyhedral model*, for example [4]–[6], for nested loop kernels whose loop indices are bounded by constants or affine functions of the surrounding loop indices, and where memory references are affine functions of the loop indices. For this type of loop kernels, it is possible to determine statically the memory reference trace which can provide exact information about the required on-chip buffer size and pre-fetching volume alongside a precise determination of the impact of transformations to enhance locality or parallelism at compile time. Our approach is less powerful in that we cannot rely on a static analysis. This, in turn, allows us to handle arbitrary problems with irregular memory access patterns and control flow. In particular, we focus on pointer-based programs here, a type of codes which cannot be analyzed in the polyhedral model.

Automated multi-cache generation from high-level specifications has been addressed in [1]–[3], [21]. Wingbermuehle *et al.* [21] implement a method similar to ours in that left-over memory resources are used to enhance the memory sub-system of stream-based kernels. Their work explores more parameters than our current technique (size, associativity, replacement rule and write policy), but the search in the parameter space is based on a simulated annealing-like technique. Another major difference to our work is that we target HLS applications without any assumption on the compute paradigm. Cheng *et al.* [2] target arbitrary input code and use memory reference profiling to partition the program's address space into independent groups. They assign private on-chip caches for each group, but they do not implement application-specific cache sizing. Since the partitioning is based on a dynamic program analysis, a fall-back mechanism must be included because an incorrect partitioning can compromise functional correctness. This work leaves the memory space partitioning to a static analysis in [3] and use the dynamic analysis only for performance but not correctness-critical optimizations. CHiMPS' many-cache system [1] is notable in that it also constructs parallel caches based on left-over block RAM, clock rate degradation and predicted miss rate, although the prediction is not described in detail in the paper. The key difference of our work is the non-uniform sizing, which is realized by solving an optimization problem to find the best assignment of cache sizes subject to a resource constraint.

VII. Conclusion

We implement a technique for optimizing the memory hierarchy for HLS applications that require access to off-chip memory. In these applications, the bulk of data usually resides in an external memory and the amount of on-chip RAM used by the HLS core is often smaller than the amount of the block RAM available. Furthermore, we target applications which require parallel memory ports and whose performance is enhanced through the insertion of parallel on-chip caches. We implement a cache compilation flow that automatically uses up the left-over block RAM to scale up the size of the on-chip caches. Secondly, the size of each cache is set individually in order to reach a size distribution across the parallel caches that maximizes the aggregate hit rate of the multi-cache architecture. The pre-synthesis cache performance estimation is based on a high-level cache model and on the memory reference trace of the application obtained from automated profiling. We cast the cache size assignment into a Multiple-Choice Knapsack Problem to find the best size distribution for a given reference trace.

We evaluate the left-over block RAM and cache hit rate estimation, and we demonstrate the latency improvements obtained from our technique using three benchmarks with irregular memory access patterns running on a VC707 FPGA board. We observe up to a $5\times$ speed-up compared to a cacheless memory interface when scaling each on-chip cache to the same maximal size. Our variably-sized multi-cache system also delivers up to a $2\times$ latency improvement ($1.5\times$ on average) compared to the one-size-fits-all solution. Although the insertion of large on-chip caches has a significant impact on the power consumption of the FPGA, we show that our variably-sized multi-cache configuration reduces the total energy by $2.7\times$ (on average) compared to a cacheless memory interface.

There are two important extensions planned for future work. The current work targets private, independent caches. Future work will focus on a model of the coherency protocol in a cache architecture consisting of coherent caches, which must take additional invalidation and owner misses due to interfering accesses by other caches into account. Secondly, our energy measurements in Table IV suggest that the optimal cache sizing changes when we optimize for energy instead of aggregate hit rate. Future work will address the development of an energy model that can be used to minimize the energy consumption of our multi-cache system.

References

[1] A. Putnam, S. Eggers, D. Bennett, E. Dellinger, J. Mason, H. Styles, P. Sundararajan, and R. Wittig, "Performance and power of cache-based reconfigurable computing," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, p. 395, Jun. 2009.

[2] S. Cheng, M. Lin, H. J. Liu, S. Scott, and J. Wawrzynek, "Exploiting Memory-Level Parallelism in Reconfigurable Accelerators," in *Proc. Int. Symp. on Field-Programmable Custom Computing Machines*, 2012, pp. 157–160.

[3] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: Memory Abstractions for Heap Manipulating Programs," in *In Proc. ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, 2015, pp. 136–145.

[4] Q. Liu, G. Constantinides, K. Masselos, and P. Cheung, "Combining Data Reuse With Data-Level Parallelization for FPGA-Targeted Hardware Compilation: A Geometric Programming Framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 305–315, Mar. 2009.

[5] S. Bayliss and G. Constantinides, "Optimizing SDRAM bandwidth for custom FPGA loop accelerators," in *Proc. Int. Symposium on Field Programmable Gate Arrays*, 2012, pp. 195–204.

[6] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2013, pp. 29–38.

[7] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein, and J. Emer, "Scavenger: Automating the construction of application-optimized memory hierarchies," in *In Proc. Field Programmable Logic and Applications*, 2015, pp. 1–8.

[8] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs: Prentice Hall, 1973.

[9] M. Hill and A. Smith, "Evaluating associativity in cpu caches," *IEEE Trans. on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec 1989.

[10] M. Brehob and R. Enbody, "An analytical model of locality and caching," Department of Computer Science, Michigan State University, Tech. Rep., 1996.

[11] K. Beyls and E. H. DHollander, "Reuse distance as a metric for cache behavior," in *In Proc. IASTED Conf. on Parallel and Distributed Computing and Systems*, 2001, pp. 617–662.

[12] D. Pisinger, "A minimal algorithm for the 0-1 knapsack problem." *Operations Research*, vol. 45, pp. 758–767, 1994.

[13] "High-Level Synthesis with LegUp." [Online]. Available: http://legup.eecg.utoronto.ca/

[14] "ROCCC 2.0 — Jacquard Computing." [Online]. Available: http://www.jacquardcomputing.com/roccc/

[15] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The LEAP FPGA Operating System," in *Proc. Int. Symp. on Field Programmable Logic and Appl.*, 2014, pp. 1–8.

[16] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. Emer, "Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2011, pp. 25–28.

[17] H.-J. Yang, K. Fleming, M. Adler, and J. Emer, "LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories," in *Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2014, pp. 117–124.

[18] V. Mirian and P. Chow, "Fcache: A system for cache coherent processing on fpgas," in *Proc. ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 2012, pp. 233–236.

[19] J. Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski, "Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems," in *Int. Symp. on Field-Programmable Custom Computing Machines*, 2012, pp. 17–24.

[20] E. Matthews, N. C. Doyle, and L. Shannon, "Design Space Exploration of L1 Data Caches for FPGA-Based Multiprocessor Systems," in *Proc. Int.Symp. on Field-Programmable Gate Arrays*, 2015, pp. 156–159.

[21] J. G. Wingbermuehle, R. K. Cytron, and R. D. Chamberlain, "Superoptimized memory subsystems for streaming applications," in *Proc. Int. Symp. on Field-Programmable Gate Arrays*, 2015, pp. 126–135.