A New Approach to Automatic Memory Banking using Trace-Based Address Mining

Yuan Zhou * Khalid Al-Hawaj * Zhiru Zhang

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY

{yz882, ka429, zhiruz}@cornell.edu

ABSTRACT

Recent years have seen an increased deployment of FPGAs as programmable accelerators for improving the performance and energy efficiency of compute-intensive applications. A well-known "secret sauce" of achieving highly efficient FPGA acceleration is to create application-specific memory architecture that fully exploits the vast amounts of on-chip memory bandwidth provided by the reconfigurable fabric. In particular, memory banking is widely employed when multiple parallel memory accesses are needed to meet a demanding throughput constraint.

In this paper we propose TraceBanking, a novel and flexible trace-driven address mining algorithm that can automatically generate efficient memory banking schemes by analyzing a stream of memory address bits. Unlike mainstream memory partitioning techniques that are based on static compile-time analysis, TraceBanking only relies on simple source-level instrumentation to provide the memory trace of interest without enforcing any coding restrictions. More importantly, our technique can effectively handle memory traces that exhibit either affine or non-affine access patterns, and produce efficient banking solutions with a reasonable runtime. Furthermore, TraceBanking can be used to process a reduced memory trace with the aid of an SMT prover to verify if the resulting banking scheme is indeed conflict free. Our experiments on Xilinx FPGAs show that TraceBanking achieves competitive performance and resource usage compared to the state-of-the-art across a set of real-life benchmarks with affine memory accesses. We also perform a case study on a face detection algorithm to show that TraceBanking is capable of generating a highly area-efficient memory partitioning based on a sequence of addresses without any obvious access patterns.

1. INTRODUCTION

With the general-purpose CPU performance scaling significantly slowing in the past decade, heterogeneous computer architectures that integrate specialized accelerators are gaining popularity to achieve improved performance and energy efficiency. Along the line, field-programmable gate arrays (FPGAs) have evolved into an attractive option for fulfilling the role of application-specific hardware acceleration, owing to the many recent technological advances on FPGA hardware capabilities as well as the software tooling support for high-level design entries.

An FPGA-based hardware accelerator is typically highly parallelized and/or deeply pipelined in order to achieve a desirable throughput. As a result, multiple parallel accesses to a single on-chip memory are often required to provide the necessary data bandwidth to sustain the high throughput of the accelerator. However, the embedded memory blocks available on modern FPGA devices (e.g., BRAMs) only provide a very limited number of ports for concurrent reads/writes.[1] Simply replicating the memory blocks would not be feasible due to the steep area overhead and potential memory coherence overhead resulting from write operations.

A more viable solution is memory banking, which partitions a memory block into several smaller banks; thus, concurrent memory accesses are distributed to different banks, avoiding or minimizing banking conflicts. Since each memory bank only holds a subset of the original memory contents, memory banking usually yields a significantly lower storage overhead compared to memory duplication. Nevertheless, additional banking logic is still required to orchestrate the data movement between banked memories and compute units in the accelerator. For non-expert FPGA designers, devising a minimum-conflict banking scheme with low hardware overheads is certainly a challenging task. While commercial high-level synthesis (HLS) tools provide some basic support for array partitioning [8], the users remain responsible for manually specifying the banking scheme via vendor-specific pragmas or directives. For this reason, there is an active body of HLS research tackling the problem of automatic array partitioning (i.e., memory banking) given a throughput constraint that is usually specified in terms of pipeline initiation interval (II) [6, 12, 14, 16, 17].

In this paper, we also focus our study on automatic memory banking, and propose *TraceBanking*, a trace-based banking algorithm that is very different from the existing methods. Specifically, TraceBanking mines a stream of memory address bits to determine a banking scheme that minimizes the number of access conflicts and simplifies the banking logic. Unlike mainstream techniques that are mostly

* The first and second authors contributed equally to this work.

FPGA '17, February 22-24, 2017, Monterey, CA, USA

 ISBN 978-1-4503-4354-1/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3020078.3021734

[1] Even for ASICs, it is not feasible to have a large number of memory ports due to the excessive area and power overhead [15].

based on static compiler analysis, TraceBanking only relies on simple source-level instrumentation to provide the memory trace of interest without enforcing any coding restrictions (such as static control parts often required by polyhedral analysis [3]). The major technical contributions of our work are threefold:

(1) We offer a fresh look at memory banking, by waiving the requirements of using static compile-time analysis. We show that from a trace of memory addresses, we can identify a set of "interesting" address bits that form the basis of the hardware-efficient memory banking function. In addition, our technique is able to form banking functions that do not belong to the solution space of the existing linear-transformation-based techniques.

(2) We propose a two-step heuristic to solve the trace-based memory banking problem. This heuristic is not only able to exploit regular memory access patterns, but can also generate efficient solutions for applications with irregular memory accesses.

(3) We propose an SMT-based checker that can formally verify if a memory banking solution is free of access conflicts under all possible execution traces. This allows the usage of a reduced (or incomplete) memory trace to significantly speed up TraceBanking, but without the risk of accepting an inferior banking solution. We believe that this formal verification technique is also useful for validating the soundness of existing memory banking algorithms, even though they are designed to be correct by construction.

The rest of this paper is organized as follows: Section 2 presents related work in memory banking; Section 3 formulates the trace-based memory banking problem; Section 4 provides motivational examples to illustrate the intuition behind our work; Section 5 describes our address mining algorithm in detail; Section 6 introduces the SMT-based banking solution checker; Section 7 reports the experimental results on commonly used benchmarks with affine memory accesses, which is followed by a case study on a face detection application with irregular memory accesses in Section 8; and Section 9 concludes this paper with discussion on future work.

2. RELATED WORK

There is a recent line of research that investigates the problem of automatic array partitioning in the context of HLS [20]. Initial efforts focus on one-dimensional arrays and attempt to find a proper cyclic partitioning with optimal scheduling to ensure conflict-free parallel data accesses [7, 11]. More recent proposals such as [16, 17] generalize these results to handle nested loops and multi-dimensional arrays.

Notably, linear transformation is extensively used among the existing array partitioning techniques. For example, the LTB approach [17] searches for a coefficient vector $\vec{\alpha}$ to construct a cyclic banking function $bank(\vec{x}) = (\vec{\alpha} \cdot \vec{x})\%N$, given the number of banks N and the affine memory access pattern. Meng et al. proposed a fast algorithm to generate the LTB coefficient vector $\vec{\alpha}$ according to the topology of the memory access pattern in multi-dimensional memory space [12]. The GMP approach is a generalization of LTB, which can generate block-cyclic banking function in the form of $bank(\vec{x}) = \lfloor(\vec{\alpha} \cdot \vec{x})/B\rfloor\%N$ [16]. More recently, Cilardo et al. proposed a lattice-based banking algorithm using polyhedron analysis [6].

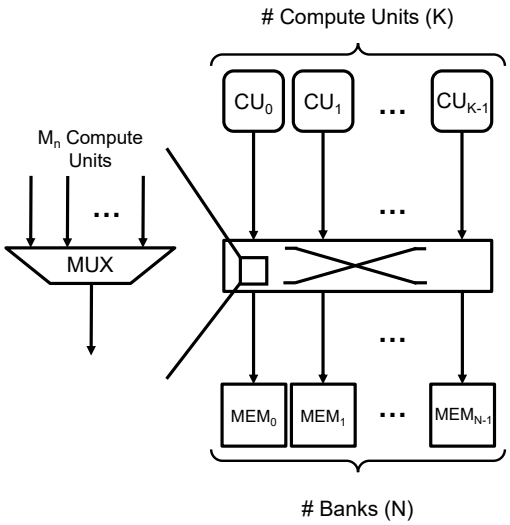


Figure 1: Hardware template for memory banking.

The aforementioned techniques all employ static compile-time analysis and are only effective with affine data access patterns. To our best knowledge, we are the first to introduce a comprehensive trace-based banking algorithm that is not limited to affine memory accesses. Along the lines of trace-based memory optimization, one relevant proposal is [4], which attempts to partition an array of data structures into distinct arrays by leveraging hints from software memory traces. However, this technique does not directly tackle memory banking for multi-dimensional arrays.

Besides memory partitioning, parallel data accesses can be further facilitated by creating data reuse buffer that exploits locality in memory access patterns. For many image processing and signal processing applications, data reuse is a more hardware-efficient solution due to the regular memory access patterns in stencil-like operations. Along these lines, a recent work by Su et al. introduced an efficient method of combining linear reuse analysis and cyclic memory partitioning to generate application-specific reuse-chains and memory-banking [14]. In this work, however, we focus on memory banking without data reuse; nevertheless, we believe that our trace-based approach can also be extended to generate data reuse buffers and will explore this topic in future work.

3. PROBLEM FORMULATION

In this section we provide the definitions and formulate the trace-based memory banking problem. An example of the hardware architecture under discussion, shown in Figure 1, contains a set of compute units and memory banks connected by a crossbar. The number of compute units is denoted as K, and the number of memory banks is denoted as N. For simplicity, we assume that each compute unit only has one memory load port, and each memory bank only has one read port. Our problem formulation as well as the proposed technique can be generalized to handle multi-bank and multi-port memories. In the following, we first define several important concepts before formulating the actual optimization problem.

Definition 1. **Memory Trace**: A memory trace T is a sequence of addresses that are grouped into L lists, where all addresses in the same list need to be accessed in parallel. Each of these lists is called a **step**. Each step contains K addresses which are issued by the compute units. In the following discussions, we refer to the l-th step in memory trace T as $T[l]$, and the memory operation requested by the k-th compute unit in step l as $T[l][k]$. If compute unit k does not issue any memory request in step l, $T[l][k]$ is marked as invalid.

Definition 2. **Memory Banking**: A memory banking solution of a trace T consists of a banking function $bank(A)$ and an offset function $offset(A)$. $bank(A)$ maps address A to a memory bank ID, while $offset(A)$ determines the intra-bank position. A memory banking solution can be fully represented by a set of binary variables $\{b_{A,n} \mid A \in T, 0 \leq n < N, n \in \mathbb{N}\}$, where $b_{A,n}$ evaluates to one if and only if address A is mapped to bank n, otherwise evaluates to zero.

Definition 3. **Banking Conflict**: A banking conflict occurs when two different addresses in the same step are mapped to the same memory bank.

Definition 4. **Mux Size**: The mux size of a memory bank n, M_n, refers to the number of compute units which access bank n in the memory trace. M_n can be represented by binary variables $\{b_{A,n}\}$ using the following equation:

$$M_n = \sum_{k=0}^{K-1} \bigvee_{l=0}^{L-1} b_{T[l][k],n}$$

With the above definitions, we can formulate the memory banking problem as an integer linear programming (ILP) problem:

Problem: Given a memory trace T, find a mapping function $bank(A)$ to optimize the following objective function, which minimizes memory access conflicts as the primary goal and reduces muxing overhead as the secondary goal.

$$\begin{aligned} Objective: \ & \alpha \cdot Conflicts + \beta \cdot Muxing \\ & = \alpha \cdot \sum_{l=0}^{L-1} \sum_{\forall A_i, A_j \in T[l]} s_{A_i,A_j} + \beta \cdot \sum_{n=0}^{N-1} M_n \end{aligned}$$

subject to

$$i, j \in [0, K-1], i \neq j, \ s_{A_i,A_j} = \sum_{n=0}^{N-1} (b_{A_i,n} \cdot b_{A_j,n}).$$

Here the addresses A_i and A_j refer to the i-th and j-th address in step $T[l]$, respectively. The binary variable s_{A_i,A_j} equals to one if and only if addresses A_i and A_j are mapped to the same bank, otherwise equals to zero (we omit the linearization of non-linear terms due to page limit).

4. MOTIVATIONAL EXAMPLES

We use two examples to illustrate the intuition behind TraceBanking. We start with the very simple example in Figure 2, where Figure 2(a) shows a simple loop kernel containing two memory accesses per iteration, and Figure 2(b) shows the associated (truncated) memory trace. From Figure 2(b), it is not difficult to tell that the two addresses in each iteration always differ in the least significant bit (LSB). We refer to such bit as a **mask bit**. Informally, we define

```
int A[SIZE+1];
for (int i=0; i<SIZE; i++)
  #pragma HLS pipeline II=1
  foo(A[i], A[i+1]);
```

(a) Loop Kernel

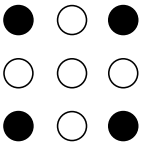
Step	addr0	addr1
0	00000**0**	00000**1**
1	00000**1**	00001**0**
2	00001**0**	00001**1**
3	00001**1**	00010**0**
...	...	...

(b) Memory Trace

Figure 2: Simple loop example — **(a)** Pipelined loop kernel with two memory accesses in each cycle. **(b)** Memory trace of the loop kernel.

```
int A[Rows][Cols];
for (int i=1; i<Rows-1; i++)
  for (int j=1; j<Cols-1; j++)
    #pragma HLS pipeline II=1
    foo(A[i-1][j-1], A[i-1][j+1],
        A[i+1][j-1], A[i+1][j+1]);
```

(a) Loop Kernel



(b) Mem Pattern

Step	addr0	addr1	addr2	addr3
0	0**0**0|0**0**0	0**0**0|0**1**0	0**1**0|0**0**0	0**1**0|0**1**0
1	0**0**0|0**0**1	0**0**0|0**1**1	0**1**0|0**0**1	0**1**0|0**1**1
...	...	...	...	...
5	0**0**0|1**0**1	0**0**0|1**1**1	0**1**0|1**0**1	0**1**0|1**1**1
...	...	...	...	...
Cols-1	0**0**1|0**0**0	0**0**1|0**1**0	0**1**1|0**0**0	0**1**1|0**1**0
...	...	...	...	...

(c) Sample memory trace

i / j	0	1	2	3	4	5	6	7
0	0	1	2	3	0	1	2	3
1	0	1	2	3	0	1	2	3
2	1	2	3	0	1	2	3	0
3	1	2	3	0	1	2	3	0
4	2	3	0	1	2	3	0	1
5	2	3	0	1	2	3	0	1
6	3	0	1	2	3	0	1	2
7	3	0	1	2	3	0	1	2

(d) GMP solution

i / j	0	1	2	3	4	5	6	7
0	0	0	1	1	0	0	1	1
1	0	0	1	1	0	0	1	1
2	2	2	3	3	2	2	3	3
3	2	2	3	3	2	2	3	3
4	0	0	1	1	0	0	1	1
5	0	0	1	1	0	0	1	1
6	2	2	3	3	2	2	3	3
7	2	2	3	3	2	2	3	3

(e) An alternative solution

Figure 3: Bicubic interpolation — **(a)** Pipelined loop kernel with four memory accesses in each cycle. **(b)** Memory access pattern of the loop kernel in two-dimensional memory space. **(c)** Memory trace generated by concatenating array indexes: Addresses are formed by concatenating the two-dimensional array indexes i and j (for simplicity, i and j are both truncated to three bits). **(d)** GMP banking solution [16]. **(e)** An alternative solution generated by selecting mask bits.

the mask bits as a subset of address bits that can differentiate all memory addresses included in the same step. We argue that the mask bits provide important hints for finding a memory banking solution. In this particular case, if we partition array `A` based on the value of the LSB, we end up with a cyclic banking scheme that enables the loop to be fully pipelined.

Figure 3 shows another (and perhaps more interesting) example from bicubic interpolation [1]. In this case, the innermost loop has four memory accesses per iteration to a two-dimensional array. The memory access pattern is illustrated in Figure 3(b) and the corresponding address stream is shown in Figure 3(c). Existing techniques, such

as GMP [16], analyze the symbolic expression of memory accesses and search for appropriate coefficients to construct a banking function in the form of $bank(i,j) = \lfloor(\alpha_0 i + \alpha_1 j)/B\rfloor \% N$. Figure 3(d) shows the resulting 4-bank partitioning scheme, where $\alpha_0 = 1$, $\alpha_1 = 2$, and $B = 2$.

Figure 3(e) shows an alternative banking scheme, which is not in the solution space of the GMP approach.[2] By examining the memory trace in Figure 3(c), we can identify two mask bits: the second-to-last bit of i, plus the second-to-last bit of j. These two bits combined can differentiate the four memory accesses belonging to the same iteration. As a result, we can divide the original array into four memory banks according to the values of the two mask bits and arrive at the alternative scheme in Figure 3(e).

These two examples demonstrate the possibility of performing memory banking based on a stream of memory addresses. Although these examples both have affine memory access patterns, TraceBanking is also capable of generating memory partitioning for applications with irregular memory accesses. Regardless of the memory access pattern, it is important to identify the mask bits that form the basis of banking. The value of mask bits is referred to as **mask ID**. In the following sections, we will discuss how TraceBanking identifies mask bits and derives efficient banking accordingly.

5. TRACEBANKING ALGORITHM

A straightforward method to optimize the objective function formulated in Section 3 is to use an ILP solver. However, ILP solvers are not scalable in general. Therefore, there is a need for heuristics which can find an optimal mapping between addresses and banks with a reasonable execution time.

In this section, we introduce TraceBanking algorithm, a flow of heuristics to solve the problem formulated in Section 3. Specifically, TraceBanking takes the number of available banks as a constraint and finds an optimized mapping by solving two sub-problems: (1) Finding a set of promising address bits to form mask bits, and (2) Finding a mapping between the generated mask IDs and available banks.

The flow of our algorithm is shown in Figure 4. The raw memory trace is first compressed, and a search is performed to find a feasible set of mask bits. Then, a graph will be generated based on the discovered mask bits. The generated graph will be colored such that each color represents a distinct memory bank. Since only the mask bits are used in the banking function, TraceBanking can potentially generate more area-efficient hardware to calculate bank IDs.

5.1 Finding Masks Bits

TraceBanking finds a set of bits from the address to form a mask in the first step. At the beginning, raw trace is preprocessed to remove redundant information, i.e. memory accesses with the same address in the same step; because having multiple accesses with the same address can be satisfied in the same cycle with no overhead. Then, the raw memory trace is compressed by initial-compression, which combines steps with identical accesses into a single step. A weight property is added to each step indicating its frequency in the raw trace. The resultant compressed trace is referred to as T_c. Figure 6(a) shows the compression process for our bicubic example; no compression can be performed since no identical steps exist in the raw trace.

[2]We omit the formal proof due to space limit.

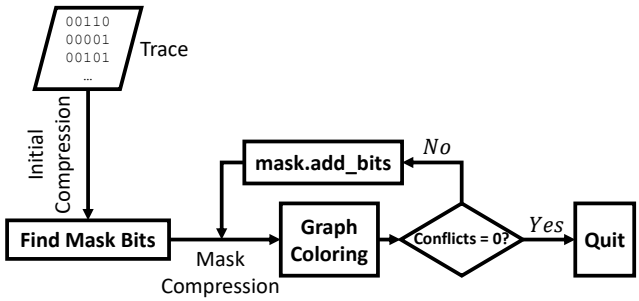


Figure 4: General structure of the proposed flow.

Algorithm 1 findMasksBits

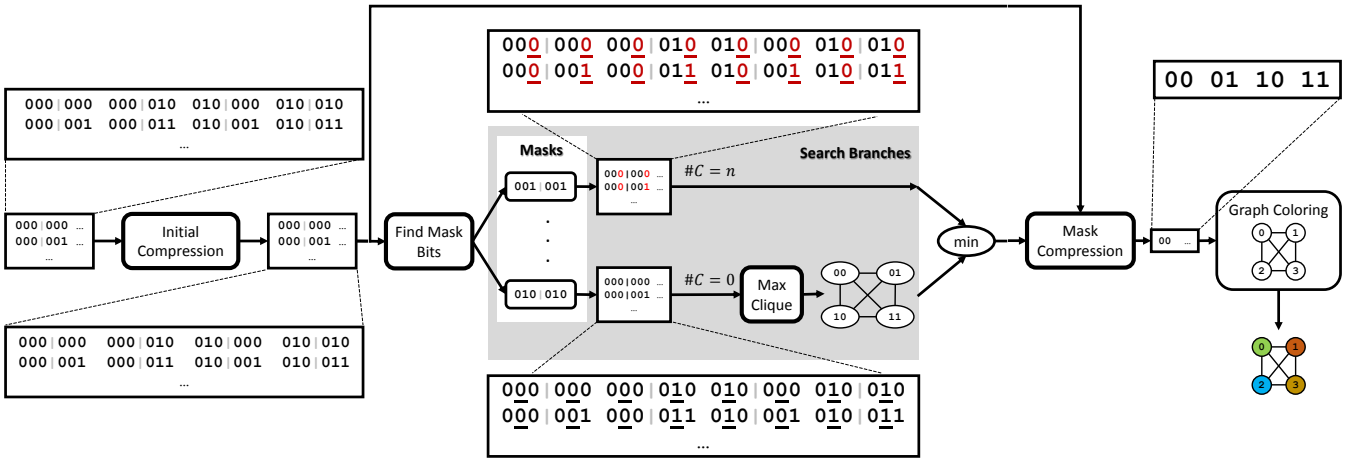
```
Input : N – number of available banks;
        T_c – Compressed memory trace;
for nbits ← ⌈log2(N_A)⌉ to address_size do
    mask ← all possible mask combinations with size nbits
    while mask ≠ null do
        num_conflicts ← calculateConflicts(T_c, mask)
        if num_conflicts ⩽ min_conflicts then
            /* Possible Solution                    */
            min_conflicts ← num_conflicts
            /* Test graph colorability              */
            G ← constructGraph(T_c, mask)
            if χ(G) ⩽ min_χ then
                min_χ ← χ(G)
            end
            if min_conflicts = 0 and min_χ ⩽ N then
                break
            end
        end
        mask ← next(mask)
    end
end
```

Figure 5: The first heuristic in our flow to find mask bits.

After cleaning up and compressing the trace, TraceBanking performs a multi-objective exhaustive search using the `findMasksBits` algorithm in Figure 5. This algorithm takes the available number of banks, N, as well as the compressed memory trace, T_c, as inputs. It evaluates any candidate mask using two objectives: *mask IDs' conflicts* and *conflict graph colorability*.

The search starts with masks that includes $\lceil log_2(N_A)\rceil$ bits, where N_A is the maximum number of memory accesses in all steps in the compressed memory trace. It tries all possible combinations of $\lceil log_2(N_A)\rceil$ bits; each combination constructs a unique mask which maps addresses to mask IDs. Going through the compressed memory trace, the algorithm evaluates mask IDs conflicts by counting the number of times when two addresses in the same step have the same mask ID.

After finding a mask that has the lowest number of mask IDs conflicts, the algorithm constructs a conflict graph — every node in the graph represents a mask ID; edges between nodes indicate mask IDs that appeared together in at least one step, and the edges' weights represent the frequency. Thus, the problem is transformed to a graph coloring problem. The algorithm calculates a lower bound for



(a) **(b)** **(c)**

Figure 6: Bicubic example being processed throughout our algorithm — **(a)** Initial part of the flow where the raw memory trace is compressed and preprocessed; in this example, no compression nor preprocessing is possible. **(b)** The first heuristic of the flow conducting a search for mask bits; underlining shows the bits considered for masking and conflicts are highlighted with red colored mask bits. For masks with no conflicts, the heuristic checks colorability by max-clique. Then, the heuristic takes the mask with minimum conflicts and minimum max-clique (#C symbolizes number of conflicts for each search branch). For bicubic, the mask with no-conflicts and a max-clique of at most N, 4, is `010 010`. **(c)** Finally, the second heuristic mask-compresses the trace and colors the generated graph. It is obvious that for bicubic the graph is rather simple to color as it is only a clique of size 4.

Algorithm 2 mapMaskIDsToBanks

Input : N – number of available banks;
T_c – Compressed memory trace;
$mask$ – Initial Mask;

do
  `/* Perform mask-compression */`
  $T_{mc} \leftarrow$ maskCompression(T_c, $mask$)
  `/* Construct a graph */`
  $G \leftarrow$ constructGraph(T_{mc})
  `/* Create a seed using greedy coloring */`
  $S \leftarrow$ greedyGraphColoring(G, N)
  `/* Color G using evolutionary algorithm */`
  $num_{conflicts}, mapping \leftarrow$ eaGraphColoring(S, N)
  `/* Ending conditions */`
  if $bits\_remaining(mask) = 0$ **then**
    break
  else if $num_{conflicts} \neq 0$ **then**
    $mask \leftarrow$ performBestFirstSearch(T_c, $mask$)
  end
while $num_{conflicts} \neq 0$;

Figure 7: The second heuristic in our flow to map mask IDs to banks.

the colorability of the conflict graph by finding the maximum clique; where graphs with maximum clique size greater than the number of banks, N, *cannot* be colored with N colors. Using bicubic as an example, Figure 6(b) shows the resulting graph; it is obvious that the graph has a maximum clique of 4; therefore, the first step concludes with the mask 010 010.

5.2 Mapping Mask IDs to Banks

The second step, with its algorithm shown in Figure 7, takes the mask found by `findMasksBits` and finds bank assignments for mask IDs such that the number of potential conflicts is minimized. To reduce complexity and redundant work, the algorithm further compresses the trace by applying mask-compression, which is similar to initial-compression explained earlier except that the addresses are replaced with their corresponding mask IDs. After that, the algorithm will construct a conflict graph from the mask-compressed trace.

To find a coloring for the generated graph, the algorithm reduces the problem to maximum coloring.[3] The number of banks represents the number of colors available for coloring. The algorithm then generates a colored seed, S, using multiple order-based greedy heuristics [2, 10]. If the seed is not conflict-free, TraceBanking attempts to minimize the number of conflicts using an evolutionary algorithm [10]. In each evolutionary step, TraceBanking performs a set of heuristics that showed efficiency in coloring memory-accesses graphs [10]. Once a coloring for a conflict graph is found, the evolutionary algorithm concludes with banking function $bank(A)$ constructed from the coloring.

If the algorithm cannot find a conflict-free coloring in a bounded number of evolutionary steps, it is assumed that the graph is uncolorable. Then, TraceBanking proceeds to perform a best-first search. The search will modify the mask by adding one extra bit to it. It is reasonable to assume that address bits that are part of the final mask have an additive effect in reducing conflicts when considered; as a result, the best-first search tests the colorability of remaining bits by adding them to the mask in isolation. Then, the search includes the bit that yields a graph with the minimum number of conflicts permanently to the mask. Since this is a rough assumption, TraceBanking might use more bits than theoretically needed to find a feasible banking.

Taking the bicubic example from before, the algorithm will take the mask found by `findMasksBits` and its corresponding conflict graph. Then, it will attempt to color the four-node conflict graph shown in Figure 6(c). Because the

[3] In this paper we strictly target conflict-free solutions. However, TraceBanking is easily extended to adapt conflict-less solutions.

graph is actually a clique of four, it will be colored with four different colors, as shown in Figure 6(c). The resulting colored graph is conflict-free. Therefore, the algorithm finishes and produces the solution in Figure 3(e).

5.3 Offset Generation

After finding the mask bits and generating the banking function $bank(A)$, we need to find an offset function $offset(A)$ to transform an address A to a corresponding intra-bank offset. An intuitive method to generate the offset function is to simply scan every data element in the data domain and assign consecutive integers to data elements in each bank. Without any constraints on the offset function, this integer counting method is effective for both regular and irregular banking solutions. In addition, this method is optimal in terms of storage overhead since the data elements in each bank are guaranteed to have consecutive intra-bank offsets.

5.4 Uncovering Closed-Form Representations

The banking and offset functions obtained from Sections 5.2 and 5.3 are represented in the form of look-up tables by default. For applications with regular memory access patterns, it is possible to convert the look-up tables generated by TraceBanking into equivalent closed-form equations, which essentially uncovers and exploits the regularity in the original application.

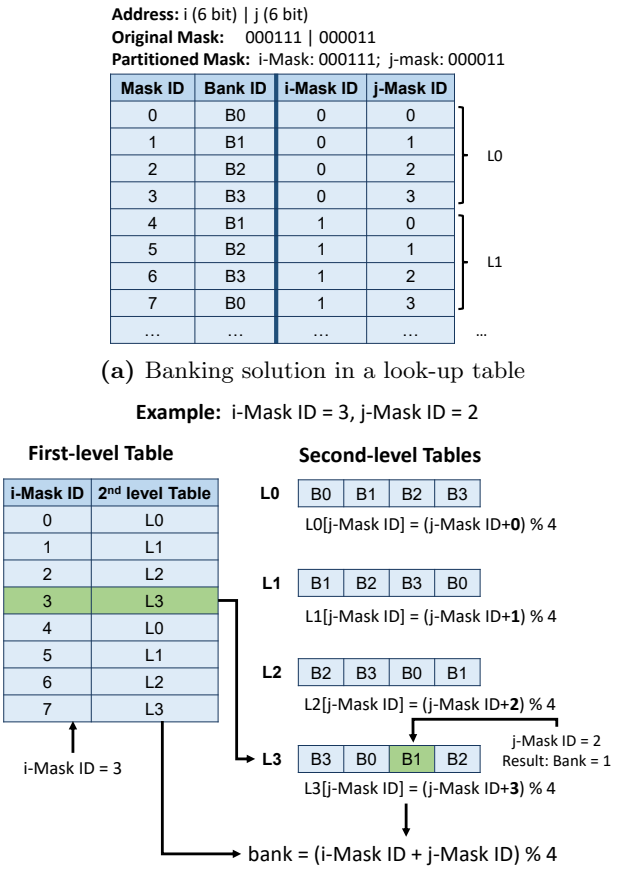
Our key idea is to decompose the look-up table into multiple stages of smaller look-up tables, and use a simple search to map the sub-tables into equations. The composition of memory addresses is retrieved from source-level instrumentation. An example is shown in Figure 8. The original 5-bit mask shown in Figure 8(a) is divided into two disjoint submasks: i-Mask and j-Mask — according to the corresponding array indices. By grouping the entries with the same i-Mask ID, the original banking solution shown in Figure 8(a) is decomposed into two levels, where the first level is used to determine the look-up table for the second level. Figure 8(b) shows how to index the decomposed look-up tables, where the i-Mask ID is used to index the first-level table and j-Mask ID is needed to index the second-level table and retrieve the actual bank ID. As illustrated in Figure 8(b), each second-level look-up table can be represented by a modulus operation. By searching for coefficients to represent the relationship between i-Mask ID and the constants in the equations (highlighted in bold in Figure 8(b)), we can represent the original banking solution with one closed-form equation shown in Figure 8(b). Clearly, this approach can easily be generalized to arrays with higher dimensions. We also use a similar method to uncover the closed-form equation for an offset function, if such representation exists.

According to our experiments on a set of benchmarks with affine memory accesses, all of the results generated by TraceBanking can be represented by our equation template which is generalized from block-cyclic partitioning. Some of our solutions fall into the category of the cyclic partitioning scheme mentioned in the LTB approach [17]. Other solutions are not in the solution space of block-cyclic partitioning. Nonetheless, they can be efficiently represented with a few number of mask bits (e.g., bicubic solution in Figure 3(e)).

6. SMT-BASED VERIFICATION

The previous discussion in Section 5 assume that the input memory trace to TraceBanking is complete. In other words, the input trace captures all memory accesses from the entire software execution. In this case, our solution is supposed to be sound in terms of guaranteeing no banking conflicts. When the given memory trace is incomplete or input-dependent, it is necessary to have a formal mechanism to verify if the resulting solution remains conflict-free under all possible scenarios.

To this end, we propose an SMT-based checker to validate the soundness of the solution with the aid of a simple compiler analysis. The checker takes the memory banking solution from TraceBanking, and the address expressions of the loop kernel from compiler analysis. With this information, we can formulate the SMT problem as shown in Figure 9(a). The integer variables for the SMT problem correspond to loop induction variables in the original application. We represent the banking solution as a function of array indices, and expressions of array indices as functions of loop induction variables. Then, we specify the iteration



(a) Banking solution in a look-up table

(b) Multi-level look-up table and closed-form solution

Figure 8: Example of mapping banking solution into closed-form equations — **(a)** Mask bits and the banking solution: An address bit noted as '1' is a mask bit, while an address bit noted as '0' is not. The mask bits are divided into two parts, i-Mask and j-Mask, according to the concatenation of array indices. **(b)** i-Mask is used to index the first-level table, and j-Mask is used to index the corresponding second-level table. Each second-level table can be represented with a closed-form equation. Constants in bold indicate the relationship between bank ID and i-Mask ID.

Define loop induction variables as SMT variables:
int $\vec{i}$
Define banking function:
int B($\overrightarrow{idx}$)
/*definition of the banking function*/
Define expressions of array indices:
int[ ] $idx_0(\vec{i})$
/*represent array indices in the 1^{st} load instruction*/
int[ ] $idx_1(\vec{i})$
/*represent array indices in the 2^{nd} load instruction*/
...
/*define the total number of load instructions*/
const int instr_cnt = K
Construct iteration domain $\mathbb{D}$:
assert (($i[0] > 0$) and ($i[1] > 0$) and ...)
Constraint for having at least one conflict:
assert

$$\bigvee_{\forall \vec{i} \in \mathbb{D}} \bigvee_{\forall a,b \in [0,K-1], a \neq b} B(idx_a(\vec{i})) = B(idx_b(\vec{i}))$$

(a) General SMT formulation

Define loop induction variables as SMT variables:
int i, j
Define banking function:
int B(i, j)
/*select the mask bits from indices*/
return (i & 0x2) || ((j & 0x2) >> 1)
Construct iteration domain $\mathbb{D}$:
assert (($i > 1$) and ($j > 1$) and
($i <$ Rows-1) and ($j <$ Cols-1))
Constraint for having at least one conflict:
assert ((B(i-1, j-1) = B(i-1, j+1)) or
(B(i-1, j-1) = B(i+1, j-1)) or
(B(i-1, j-1) = B(i+1, j+1)) or
(B(i-1, j+1) = B(i+1, j-1)) or
(B(i-1, j+1) = B(i+1, j+1)) or
(B(i+1, j-1) = B(i+1, j+1)))

(b) Example of Bicubic interpolation

Figure 9: SMT formulation of the banking solution checker — **(a)** General formulation. **(b)** Example of Bicubic interpolation.

domain as a constraint. Additionally, we add one constraint of having at least one banking conflict in the whole iteration domain. If the SMT problem is proven to be unsatisfiable, there is no memory access conflict in all iterations and the banking solution is valid.

The example shown in Figure 9(b) illustrates how the SMT-based checker validates the banking solution for bicubic interpolation shown in Figure 3(e): the loop induction variables i and j are used as SMT variables; and the banking function is represented symbolically. The constraints specify boundaries for the SMT variables, and compare every pair of addresses from the same iteration to check for conflicts.

As a formal verification technique, the SMT-based checker is also useful for validating the soundness of existing memory banking algorithms to detect any bugs in compiler analysis and code transformation, even though they are designed to be correct by construction.

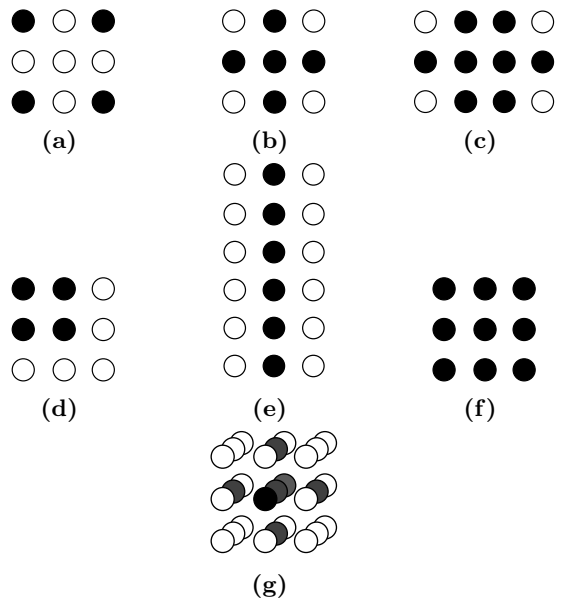


Figure 10: Memory access patterns of benchmarks — **(a)** BICUBIC, **(b)** DECONV, **(c)** DENOISE-UR, **(d)** MOTION_C, **(e)** MOTION_LV, **(f)** SOBEL, **(g)** STENCIL3D.

7. EXPERIMENTAL RESULTS

In our experiments, memory traces are generated by source-level instrumentation of the loop kernels. The addresses in the memory traces are constructed by concatenating multi-dimensional array indices. The core algorithm of Trace-Banking processes the memory trace and generates banking and offset functions. This algorithm is implemented in C. We use Vivado Design Suite 2016.2 from Xilinx [19] for high-level synthesis (HLS), logic synthesis and simulation. The target FPGA device is Xilinx Virtex-7. The memory banking flow takes in the memory trace and generates solutions in the form of look-up tables or close-form equations. We use Z3, an SMT theorem prover, to verify the generated solutions [9]. Each verified banking solution as well as the corresponding application are implemented as a synthesizable HLS code.

We adopt six different loop kernels from the GMP work [16]. In addition, we add Stencil3D benchmark from MachSuite [13], which accesses a three-dimensional array, to stress the robustness and scalability of our approach. The memory access patterns of these loop kernels are shown in Figure 10 — the solid dots represent the data elements being accessed in each iteration of the loop kernels. We substitute the processing phase of these loop kernels with a simple summation to better compare the overhead of different memory banking solutions. We also implemented the GMP method [16] as the baseline. All the designs are pipelined with II of one for maximum throughput. The input size of the designs is 64×48 ($5 \times 64 \times 48$ for Stencil3D), and the data size is 8-bit. We employ efficient algorithms from [18] and implement our own modulus functions to minimize area. These customized modulus functions are used in both the baseline and our approach.

7.1 Area Comparison

Table 1 shows comparison with the baseline, where the minimum number of banks is used. Both GMP and our ap-

Table 1: Timing and resource usage comparison with baseline using GMP [16], where the minimum number of memory banks is used — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period.

Benchmark	# Accesses	Method	# Banks	Mask Width	BRAM	Slice	LUT	FF	DSP	CP(ns)
BICUBIC	4	Baseline	4	-	4	74	217	163	0	3.89
		TraceBanking	4	2	4	74 (+0.0%)	212 (-2.3%)	184 (+13%)	0 (+0.0%)	3.66
DECONV	5	Baseline	5	-	5	185	531	383	10	3.52
		TraceBanking	5	12	5	182 (-1.6%)	541 (+1.9%)	383 (+0.0%)	10 (+0.0%)	3.37
DENOISE-UR	8	Baseline	8	-	8	180	616	391	0	4.15
		TraceBanking	8	4	8	188 (+4.4%)	623 (+1.1%)	427 (+9.2%)	0 (+0.0%)	3.62
MOTION_C	4	Baseline	4	-	4	76	186	153	0	3.58
		TraceBanking	4	2	4	68 (-11%)	193 (+3.8%)	190 (+24%)	0 (+0.0%)	3.65
MOTION_LV	6	Baseline	6	-	6	146	425	392	6	3.31
		TraceBanking	6	6	6	146 (+0.0%)	425 (+0.0%)	392 (+0.0%)	6 (+0.0%)	3.31
SOBEL	9	Baseline	9	-	9	405	1296	692	27	3.93
		TraceBanking	9	12	9	350 (-14%)	1059 (-18%)	719 (+3.9%)	27 (+0.0%)	3.96
STENCIL3D	7	Baseline	7	-	14	322	966	700	7	3.82
		TraceBanking	7	15	14	308 (-4.3%)	932 (-3.5%)	624 (-11%)	7 (+0.0%)	3.74
Average						-3.8%	-2.4%	+5.6%	+0.0%	

Table 2: Timing and resource usage comparison with baseline using GMP [16], where the number of memory banks is restricted to be a power-of-two — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period.

Benchmark	# Accesses	Method	# Banks	Mask Width	BRAM	Slice	LUT	FF	DSP	CP(ns)
DECONV	5	Baseline	8	-	8	129	418	278	0	3.63
		TraceBanking	8	4	8	125 (-3.1%)	411 (-1.7%)	302 (+8.6%)	0 (+0.0%)	3.11
MOTION_LV	6	Baseline	8	-	8	117	369	237	0	3.56
		TraceBanking	8	3	8	119 (+1.7%)	391 (+6.0%)	282 (+19%)	0 (+0.0%)	3.77
SOBEL	9	Baseline	16	-	16	328	1114	525	0	4.34
		TraceBanking	16	4	16	340 (+3.7%)	1129 (+1.3%)	472 (-10%)	0 (+0.0%)	3.89
STENCIL3D	7	Baseline	8	-	8	195	649	443	0	3.87
		TraceBanking	8	6	8	201 (+3.1%)	655 (+0.9%)	450 (+1.6%)	0 (+0.0%)	3.70
Average						+1.4%	+1.6%	+4.8%	+0.0%	

Table 3: Execution time of TraceBanking on Motion_LV with different array sizes.

Array Size	12×12	32×24	64×48	128×96	320×240	640×480
Runtime (s)	0.0096	2.19	4.88	6.94	12.87	33.38

proach can generate valid banking solutions with the minimum number of memory banks. Our approach is able to reduce the number of slices by 3.8% on average. One of the reasons is that our banking function does not always use all the bits in the address or array indices, which in turn reduces the complexity of the banking logic. For example, in Motion_C, we are able to save 11% of slices with a 2-bit mask. Another reason is that our approach is able to discover additional banking solutions that are not in the search space of the GMP method. For example, in Sobel, our design uses all the 12 index bits but still saves 14% of slices compared to the baseline; while the GMP solution has to perform `mod 9` operations due to its block-cyclic nature, our solution alternates among three consecutive bank IDs in each row of the image, thus only requires `mod 3` operations which is more area-efficient.

As pointed out by [16], an important design trade-off between logic complexity and storage overhead in memory partitioning is to enforce the number of memory banks to be a power-of-two instead of the minimum. Therefore, we conduct this experiment for the four benchmarks whose number of banks is not a power-of-two and compare our results with the baseline. Detailed experiment results are shown in Table 2. Comparing with the corresponding entries in Table 1, the designs in Table 2 generally have less area even though they use more memory banks and a more complex crossbar, because banking functions are significantly simplified when the number of banks is a power-of-two. For GMP designs, multiplication and division become simple shifting operations, while modulus operations are just selecting LSBs. For our designs, the resource saving comes from the reduction in mask width. Compared with baseline, our designs use a negligible 1.4% more slices. In general, the hardware generated by our trace-based memory banking approach is comparable with GMP in terms of area and timing.

7.2 Scalability

TraceBanking is able to generate competitive memory banking solutions from memory traces. However, using a complete memory trace may be expensive when the memory trace is large. Table 3 shows how the execution time of TraceBanking scales with an increasing array size. For applications with affine memory accesses, we can apply trace reduction to reduce the runtime. The general idea is to use

Table 4: Execution time of TraceBanking with reduced memory trace — Initial mask refers to the mask found by Section 5.1, while the Final mask refers to the mask found by Section 5.2.

Benchmark	Reduced Array Size	Mask Width		Runtime (s)
		Initial	Final	
BICUBIC	8 × 8	2	2	0.0093
DENOISE	10 × 10	4	8	3.45
DENOISE2	16 × 16	4	4	0.017
MOTION_C	8 × 8	2	2	0.0094
MOTION_LV	12 × 12	4	4	0.0096
SOBEL	18 × 18	6	10	5.94
STENCIL3D	5 × 14 × 14	6	11	4.37

a partial memory trace which covers an adequate number of steps. Because of memory access pattern redundancy in the trace, the generated banking scheme is likely to comply with banking schemes generated from a full trace. Since the banking scheme generated from a partial trace is not guaranteed to be valid, we use the SMT-based checker proposed in Section 6 to validate it. If the validation fails, we revert to using the complete memory trace.

We perform experiments with reduced memory traces for all the benchmarks listed in Table 1. For the size of the reduced trace, we use an empirical value of $2 \times \#Banks$ in each dimension of the array. For example, if the loop kernel conducts Sobel edge detection on an VGA image (640 × 480), rather than iterating through the whole image, we execute the loop kernel on an 18×18 sub-image and use this reduced trace as the input to TraceBanking. For all the benchmarks listed in Table 1, TraceBanking is able to generate solutions which are proven to be valid using the reduced traces as inputs. Moreover, these solutions are identical to the ones generated from complete traces. The execution time of the SMT-based checker is less than a second. As shown in Table 4, the execution time of TraceBanking is reduced significantly by using partial traces without sacrificing the quality of the solutions.

A critical observation from Table 4 is that, in most benchmarks, the final solution is either in the beginning or at the very end of the search space. TraceBanking exploits the aforementioned observation in pruning the search space by performing two simultaneous searches: forward search and backward search. Forward search starts from the mask with minimum number of bits upward to the mask with maximum number of bits, stopping with the first mask that yields no conflicts. On the other hand, backward search starts from the mask with the maximum number of bits downward to the mask with minimum number of bits, stopping when no bit can be removed without causing conflicts.

8. CASE STUDY: HAAR FACE DETECTION

In this section, we use Haar face detection [5] as a case study to show the efficacy of TraceBanking on applications with non-affine memory accesses. The Haar algorithm uses cascaded classifiers to detect human faces rapidly and robustly. Thousands of weak classifiers are integrated into a Haar system, and each of them has a distinct memory access pattern. A code snippet of the loop kernel in Haar algorithm is shown in Figure 11. The array `window` is a 25 × 25 image buffer and is steadily shifted in from the input image. Therefore, it is implemented with discrete registers. In each iteration, the loop kernel reads pixels into the array `coord` and process them in the function `foo()`. There are 2913 classifiers in total. The constant arrays `rectangles_array[]` store the constants needed to compute the array indices in each iteration. There is an `if` statement inside the loop kernel. When the condition is met, the loop kernel accesses 12 pixels from the `window` array in that iteration; otherwise, 8 pixels are accessed.

```
pixel window[25][25];
pixel coord[12];
int filter_no;

CLASSIFIER:
for (filter_no=0; filter_no<2913; filter_no++){
  #pragma HLS pipeline II=1
  // read array indexes from look-up tables
  int x0 = rectangles_array0[filter_no];
  int y0 = rectangles_array1[filter_no];
  int w0 = rectangles_array2[filter_no];
  ...
  // access 8 data elements from array
  coord[0] = window[y0][x0];
  coord[1] = window[y0][x0+w0];
  ...
  // if condition met, access 4 more elements
  if ( (w2!=0) && (h2!=0) ) {
    coord[8] = window[y2][x2];
    ...
  }
  else {
    coord[8] = 0;
    ...
  }
  // process data
  foo(coord);
}
```

Figure 11: Loop kernel of Haar face recognition.

In order to maximize throughput, we need to fully pipeline the `CLASSIFIER` loop in Figure 11, where each classifier requires eight or 12 parallel accesses to the image buffer. Existing techniques cannot generate an efficient banking solution for this problem due to two reasons: (1) The 2913 classifiers have more than 2000 different memory access patterns in total, and (2) The array indices are non-affine without any linear relationship with the iteration variable `filter_no`. With TraceBanking, we are able to generate a conflict-free banking solution to partition the image buffer `window[25][25]` into 28 memory banks using the whole address as mask bits. The execution time is less than a second. Because the `window` array is a shifting window implemented using discrete registers, in this scenario, the memory banks are actually register banks.

Our baseline is a straightforward design that uses 12 instances of 625-to-1 multiplexer. We compare our design with the baseline and the result is shown in Table 5. The TraceBanking design in Table 5 refers to the memory banking design generated by our approach, and the Full Mux design refers to the baseline. For these two designs, we only extract the loop kernel part shown in Figure 11 to better compare the banking hardware overhead.

Table 5: Timing and resource usage comparison of the two designs — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period; `Latency` = latency of the loop kernel.

Implementation	BRAM	Slice	LUT	FF	DSP	CP(ns)	Latency
TraceBanking	34	4915	8266	12559	6	4.52	2923
Full Mux	22	21275	53553	23785	3	9.22	2919

Table 5 compares the Full Mux design with the TraceBanking design. Our TraceBanking design reduces Slice, LUT and Flip-Flop usage by 76.9%, 84.6% and 47.2%, respectively. Meanwhile, clock period is improved by 51.0%. BRAM usage increases because of the overhead in storing look-up tables for banking and offset functions. The reduction in logic resource usage results from the simplified muxing network in the TraceBanking design. In the TraceBanking design, two levels of multiplexers are used to connect the registers with the compute units, and each multiplexer has less than 30 inputs. In contrast, the Full Mux design uses 12 instances of 625-to-1 multiplexers, which consumes a lot more area. Even worse, the Full Mux design is extremely hard to route and unable to meet the 5ns timing target. Therefore, even though the Full Mux design has similar latency with the TraceBanking design, the total execution time of the loop kernel is much worse. Clearly, the banking scheme generated by TraceBanking helps improve both area and performance of the design, which contains very irregular memory accesses.

9. CONCLUSION AND FUTURE WORK

In this work, we propose TraceBanking, a memory banking approach using trace-based address mining. By analyzing the input memory trace of an application, we select the important address bits which can guide our banking decision, and apply a graph coloring algorithm to generate efficient banking solutions. We also propose an SMT-based checker to validate our solution. Experiments show that TraceBanking can generate comparable hardware with the state-of-the-art partitioning algorithm for applications with affine memory access patterns. In addition, a case study on Haar face detection demonstrates that TraceBanking can generate valid and efficient banking solutions for applications with non-affine memory accesses.

Our work opens a new path in automatically generating parallel hardware architecture using memory trace analysis, and we prove that this compute-intensive task can be accomplished within a reasonable amount of time. The execution time of our approach can be further reduced by launching parallel threads to exploit the heavy parallelism in the best-first search.

Furthermore, we believe our approach can be extended to generate other specialized parallel architectures, such as data reuse buffers commonly used for stencil-like applications. Additionally, trace analysis provides an opportunity for automatically generating a specialized on-chip caching system for applications with non-affine memory accesses, or even data-driven applications whose memory access pattern can not be statically determined.

Acknowledgements

This work was supported in part by Intel Corporation under the ISRA Program, NSF Awards #1337240 and #1453378, a DARPA Young Faculty Award, and a research gift from Xilinx, Inc. Khalid Al-Hawaj is supported by King Abdullah Scholarship Program (KASP) and King Fahd University of Petroleum and Minerals (KFUPM).

10. REFERENCES

[1] Bicubic Interpolation. http://www.mpi-hd.mpg.de/astrophysik/HEA/internal/Numerical_Recipes/f3-6.pdf.

[2] H. Al-Omari and K. E. Sabri. New Graph Coloring Algorithms. *American Journal of Mathematics and Statistics*, 2(4):739–741, 2006.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting Polyhedral Loop Transformations to Work. In *Languages and Compilers for Parallel Computing*, pages 209–225. Springer, 2003.

[4] Y. Ben-Asher and N. Rotem. Automatic Memory Partitioning: Increasing Memory Parallelism via Data Structure Partitioning. *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 155–162, 2010.

[5] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner. FPGA-Based Face Detection System using Haar Classifiers. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2009.

[6] A. Cilardo and L. Gallo. Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):45, 2015.

[7] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(2):15, 2011.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[9] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] T. R. Jensen and B. Toft. *Graph Coloring Problems*, volume 39. John Wiley & Sons, 2011.

[11] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory Partitioning and Scheduling Co-Optimization in Behavioral Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2012.

[12] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei. Efficient Memory Partitioning for Parallel Data Access in Multidimensional Arrays. *Design Automation Conf. (DAC)*, 2015.

[13] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014.

[14] J. Su, F. Yang, X. Zeng, and D. Zhou. Efficient Memory Partitioning for Parallel Data Access via Data Reuse. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 138–147, 2016.

[15] Y. Tatsumi and H. Mattausch. Fast Quadratic Increase of Multiport-Storage-Cell Area with Port Number. *Electronics Letters*, 35(25):2185–2187, 1999.

[16] Y. Wang, P. Li, and J. Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.

[17] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2013.

[18] H. S. Warren. *Hacker's Delight*. Pearson Education, 2013.

[19] Xilinx. Vivado Design Suite - HLx Editions, 2016.2. http://www.xilinx.com/products/design-tools/vivado.html.

[20] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A Platform-Based ESL Synthesis System. In *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112. Springer, 2008.