Efficient Memory Arbitration in High-Level Synthesis from Multi-threaded Code

Jianyi Cheng, *Student Member, IEEE,* Shane T. Fleming, Yu Ting Chen, Jason Anderson, *Senior Member, IEEE,* John Wickerson, *Senior Member, IEEE,* George A. Constantinides, *Senior Member, IEEE*

Abstract—High-level synthesis (HLS) is an increasingly popular method for generating hardware from a description written in a software language like C/C++. Traditionally, HLS tools have operated on sequential code, however in recent years there has been a drive to synthesise multi-threaded code. In this context, a major challenge facing HLS tools is how to automatically partition memory among parallel threads to fully exploit the bandwidth available on an FPGA device and minimise memory contention. Existing partitioning approaches require inefficient arbitration circuitry to serialise accesses to each bank because they make conservative assumptions about which threads might access which memory banks. In this article, we design a static analysis that can prove certain memory banks are only accessed by certain threads, and use this analysis to simplify or even remove the arbiters while preserving correctness. We show how this analysis can be implemented using the Microsoft Boogie verifier on top of satisfiability modulo theories (SMT) solver, and propose a tool named EASY using automatic formal verification. Our work supports arbitrary input code with any irregular memory access patterns and indirect array addressing forms. We implement our approach in LLVM and integrate it into the LegUp HLS tool. For a set of typical application benchmarks our results have shown that EASY can achieve 0.13× (avg. 0.43×) of area and 1.64× (avg. 1.28×) of performance compared to the baseline, with little additional compilation time relative to the long time in hardware synthesis.

Index Terms—High-Level Synthesis, HLS, Formal Methods, Multi-threaded Code, FPGA.

1 Introduction

FPGAs are starting to become mainstream devices for custom computing, particularly deployed in datacentres, through, such as the Microsoft Catapult project [1] and Amazon EC2 F1 instances [2]. Using these FPGA devices, however, requires familiarity with digital design at a low abstraction level. This hinders their use by those who do not have a hardware background. Aiming to bring the benefits of custom hardware to software engineers, high-level synthesis (HLS) allows the use of a familiar language, such as C, and automatically translates a program into a hardware description. This process can significantly reduce the design effort and time compared to manual register transfer level (RTL) implementations. Various HLS tools have been developed by both academia and industry, such as LegUp from the University of Toronto [3], Bambu from the Politecnico di Milano [4], Xilinx Vivado HLS [5] and Intel's HLS Compiler [6].

The input to HLS tools for parallel hardware synthesis can be either single-threaded (sequential) code (e.g. Xilinx Vivado HLS) or multi-threaded (concurrent) code (e.g. LegUp). Our work targets multi-threaded input, aiming to solve three key challenges in the general HLS world: 1) While FPGA devices provide large amounts of compute, the memory bandwidth often limits their throughput. 2) To increase the memory bandwidth, partitioning schemes can be used to split memory into smaller distributed memories or banks, allowing for parallel accesses to data items. However, to ensure each thread can still access any data in the partitioned memory, arbitration logic must be used to serialise accesses to each individual partition. 3) As the number of memory banks or compute threads increases, the overheads of arbitration grow quadratically, resulting in a challenge in scalability.

When optimising the memory architecture of on-chip memory, HLS tools like LegUp address the memory bandwidth and correctness challenges by performing automated memory partitioning and using a crossbar arbiter to ensure global accessibility. However, this approach does not solve the scalability issue, as the fully connected arbitration logic imposes excessive routing overheads and lengthy critical paths. One solution to this problem is for users to manually edit the software program to specify disjoint regions of memory, which enables the optimisation of arbitration logic. If a user specifies that a region of memory is only accessed by one thread, then no arbitration logic is required. However, for complex code it can often be challenging and error prone for the user to manually determine memory bank exclusivity. Such an approach is counter to the fully automated design philosophy of HLS tools.

In this article, we propose EASY to solve the scalability challenge by employing ideas from traditional software verification. Overcoming the challenge requires solving the problem of determining which threads can access which memory partitions. We reduce this problem to the problem of verifying assertions in a Boogie program. EASY automati-

- *J. Cheng, J. Wickerson and G. Constantinides are with the Department of Electrical and Electronic Engineering, Imperial College London. E-mail: {jianyi.cheng17, j.wickerson, g.constantinides}@imperial.ac.uk*
- *S. Fleming is with the Department of Computer Science at Swansea University. E-mail: s.t.fleming@swansea.ac.uk*
- *Y. Chen and J. Anderson are with the Department of Electrical and Computer Engineering, University of Toronto, Canada. E-mail: {joyuting.chen@mail.utoronto.ca, janders@ece.utoronto.ca}*

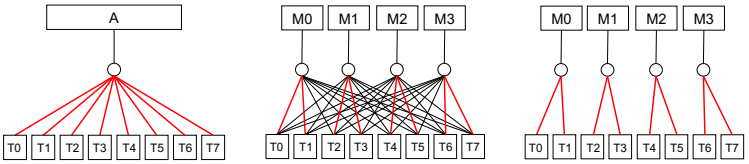
```
1 T_seq(){
2   K = 1024;
3   for(i=0;i<8*K;
      i++)
4     s += A[f(i)
      ];
5 }
```

Sequential function.

```
1 T(n){ // n = {0..7}
2   K = 1024;
3   for(i=n*K;i<(n+1)*K;
      i++)
4     accum += A[f(i)];
5   s += accum;
6 }
```

Thread function

(a) A minimal example of a sequential function and its parallel form in a 8-thread code to perform array summation.



(b) A example of memory partitioning of a 8-thread hardware and arbitration simplification. (1) all the threads (T) accessing the shared memory results in performance bottleneck in memory bandwidth; (2) the bandwidth is increased by partitioning the memory into 4 banks (M) but cause huge arbitration logic; (3) the arbitration logic is simplified by proving some of them cannot be used.

Fig. 1. Motivating Examples. We show our work can *simplify* the memory arbitration logic among multiple threads and memory partitions.

cally transforms a multi-threaded program into a related sequential program in a verification-oriented language called Boogie [7], together with assertions. We apply the Boogie tool flow to automatically generate satisfiability modulo theory (SMT) queries, and then EASY interprets the corresponding results as directives to simplify the arbitration logic in the circuit. Our work can address the scalability challenge by extending the current LegUp tool flow with a fully automated static analysis that supports arbitrary input code.

Contributions

The main contributions of this work are:

1) A general technique that uses formal methods to prove memory bank exclusivity for arbitrary multi-threaded input code (Section 2);
2) A technique for transforming generic LLVM intermediate representation (IR) code for multi-threaded programs into the Microsoft Boogie (single-threaded) verification language (Section 4) and a fully automated HLS pass that calls the Boogie verifier to formally prove that arbitration is not required between certain combinations of memory banks and program threads, enabling the removal or radical simplification of arbitration logic in an automated fashion (Section 5); and
3) Analysis and results showing that, over a number of benchmarks, the proposed approach can achieve an area saving of up to 87% (avg. 57%) and a wall-clock time improvement of up to 39% (avg. 22%) compared to the design with fully-connected arbiters (Section 6).

Relationship to Prior Publications

This article expands on a conference paper by Cheng *et al.* [8]. The previous work only supports partitioned arrays that are directly addressed, *i.e.* the array index being accessed does not depend on the data in the array. However, indirectly addressed arrays are required in various contemporary HLS applications, such as sparse matrix computation in deep learning and graph analytics and sorting algorithms in databases. Applications with sparse memory architecture cannot be optimised by our prior work. Our additional contribution that this article makes is the support for indirectly addressed partitioned arrays and the results of evaluating this over a new series of benchmarks. We further demonstrate that our analysis can correctly perform code transformation and verification with such irregular and complex memory access patterns. For applications containing indirectly addressed partitioned arrays that cannot be simplified by the original work, we show up to a 75% (avg. 66%) area saving and up to 36% (avg. 25%) wall-clock time reduction.

Auxiliary Material

All of the results from our experiments are available online [9]. EASY and benchmarks are also open-sourced [10].

2 Motivating Example

Fig. 1(a)[1] gives two examples of C code for the same operation: sequential code and multi-threaded code. The sequential code on the left side of the figure accumulates the data stored in array `A` sequentially in a loop. In multi-threaded code, the process is parallelized using eight threads. Each thread function `T_n`, where `n` ranges from 0 to 7, can be represented as the function on the right. In both the sequential code and the multi-threaded code, the element of `A` that is selected for each assignment uses the loop iterator, `i`, and the pure function `int f(int i)`. This example can be synthesised into three hardware architectures when using different constraints, as shown in Fig. 1(b). Array `A` is implemented as a shared memory block by default. Since array `A` is accessed by all the threads, an arbiter is placed to serialise the memory accesses, as shown in Fig. 1(b1). There may be memory contention when two or more threads access the same memory port concurrently. To minimise this contention, array `A` can be partitioned into four memory banks to increase the memory bandwidth. Then each thread will use the function `f` to decide which memory bank they are going to access. If `f` is sufficiently complicated, it may be non-trivial for a developer to know how the memory should be partitioned for parallel access. Therefore, to ensure correctness, HLS tools like LegUp implement an arbiter for each partitioned memory bank connecting to *all* threads, as shown in Fig. 1(b2).

1. The optimal number of memory partitions for this example should equal to the number of threads for complete removal of arbiters. Here we show that even if this is not the case, arbiter logic can be simplified by our technique.

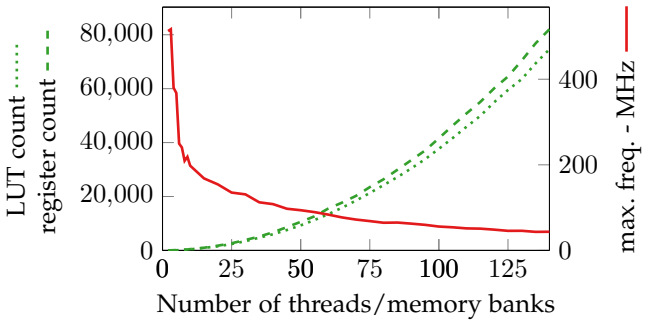


Fig. 2. Evaluation on performance and chip area of original arbiters.

However, there may be scenarios wherein a thread cannot touch all the memory banks. For example, a simple case is `f(i) = i`. In this case, the array indices accessed in `T0` never overlap with those accessed by the other threads. That is, for the array `A` of size $8K$, thread i only touches elements with indices $i \times K$ to $(i+1) \times K - 1$ of array `A`. As these threads are accessing mutually disjoint regions of `A`, a block partitioning strategy can be applied, where the array `A` of size $8K$ can be divided into four sub-arrays, each of size $2K$. Threads and banks are now in two-to-one correspondence, so we can simplify the arbiters. This results in the simplified architecture shown in Fig. 1(b3).

Since the size and delay of the arbitration logic grow quadratically, this simplification process can be highly effective when there are many threads and banks. Fig. 2 shows how the arbitration logic grows in area and maximum clock frequency as the number of memory banks and the same number of threads. We take a single memory arbiter module used by LegUp [3] and synthesise it in Quartus [11] with different parameters, *i.e.* the number of connected threads. Then the area of total memory arbitration can be estimated by multiplying the area of a single arbiter by the number of memory partitions, and the maximum clock frequency of the arbitration logic can be estimated as the same as that of a single arbiter. As expected, increasing the number of threads decreases the maximum clock frequency and quadratically increases the hardware utilisation. None of these arbiters is simplified because the HLS tool conservatively assumes that any thread can access any bank.

To address the scalability issue, we propose our tool named EASY that proves the absence of access relations between each thread and each partitioned bank. A access relation refers to which threads accesses which banks. The memory accesses observed through simulation provide an under-approximation of this relation, where Boogie program provide an over-approximation of this relation. The safety of removing arbiters is verified using assertions (details in Section 4), for instance, that the black wires in the arbitration of Fig. 1(b2) cannot be used. The optimised hardware of the given example produced by EASY is shown in Fig. 1(b3).

3 Background

In this section, the tool flow of the LegUp HLS tool used in our work is introduced, followed by the explanation of the prior work on memory partitioning. Related works on memory partitioning using static program analysis are then reviewed and compared to our approach. We also compare against an alternative, dynamic approach to arbitration optimisation. Finally, the Boogie verifier we use in our work is introduced.

3.1 The LegUp HLS tool

HLS tools, like Vivado HLS [5], take sequential code as inputs. There are also HLS tools supporting multi-threaded code as inputs, like LegUp [3], Catapult [12], Stratus [13], and OpenCL-based tools [14].[2] The problem is generic: how to synthesise efficient memory arbitration to handle data parallelism. For instance, in OpenCL, the shared data is considered as global memory shared by multiple kernels, while in LegUp, similarly, threads can share global memory.

For this work, we use the LegUp HLS tool [3], in the form of C-style `pthreads` [16]. It is also a well-known and open-source HLS tool under active development. Support for multi-threading is essential since our technique uses formal methods to optimise the generated memory interface for concurrently executing hardware threads, which is realised through the synthesis of multi-threaded code. LegUp HLS allows spatial parallelism in hardware to be exploited by software engineers, through the synthesis of concurrent threads into parallel hardware modules.

LegUp is built upon the LLVM [17] compiler framework. LLVM consists of a frontend and a backend. The frontend converts the input program into LLVM-IR, which can then be optimised using a collection of pre-existing optimisation passes. The backend receives the final optimised LLVM-IR and generates architecture-specific machine code. In the case of LegUp, the backend performs HLS and produces a Verilog RTL circuit implementation. We summarise the key stages in LegUp's multi-threaded synthesis flow as follows.

1) The input C is transformed into LLVM-IR via Clang [18].
2) Each thread function destined for hardware is extracted from the rest of the source, creating a hardware LLVM-IR source for each function, and a software (host-code) LLVM-IR source.
3) The split LLVM-IR sources are transformed multiple times by a series of optimisation passes: some LegUp specific, such as word-length minimisation, and others generic, such as dead-code elimination. For the LLVM-IR host source, an additional transformation is made to convert all the `pthread_create` calls into the appropriate hardware function calls.
4) Each transformed LLVM-IR source is then turned into a Verilog description of a circuit using the traditional scheduling, allocation, and binding steps [19].
5) Interconnect logic and memory interfaces are generated to connect each of the circuits to the host system and instantiate them the appropriate number of times.
6) The software host code is compiled, and an FPGA hardware bitstream is generated by synthesising the Verilog using FPGA vendor tools.

2. Both forms of expressing parallelism are in current usage in a variety of programming languages, with multi-threading typically used for larger kernels of code [15]. We are not advocating for one form or another, but to ensure that actually-used programming idioms are well supported in HLS.

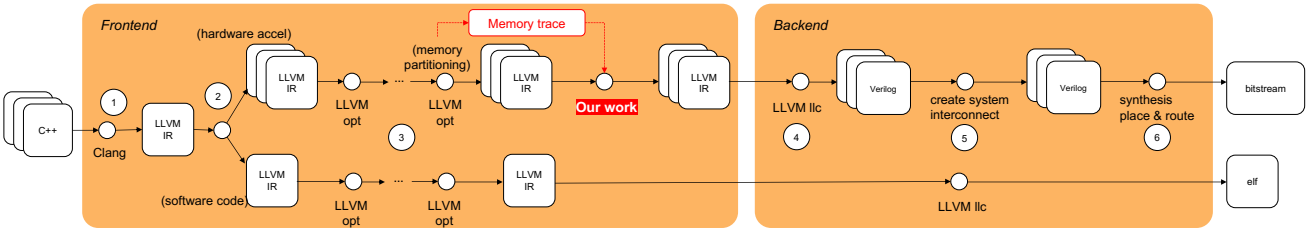


Fig. 3. A coarse overview of the LegUp multi-threaded toolflow.

Fig. 3 shows a labelled tool flow of the stages outlined above, where stages 1–3 are often referred to as the frontend and stages 4–6 are referred to as the backend. EASY is integrated at the end of the frontend.

In LegUp, each hardware thread is synthesised into a hardware circuit with a Finite State Machine (FSM) and datapath. The hardware circuits corresponding to threads operate independently. That is, there is no global schedule requiring (or enforcing) data synchronisation on memories shared among threads (which can instead be achieved by using locks and synchronisation barriers common in other HLS tools [20]).

One advantage of FPGA devices is the high internal memory bandwidth, as the numerous small distributed memories (BRAMs) can be accessed concurrently. For any data shared between multiple hardware threads, LegUp constructs a shared memory out of BRAMs to provide fast local access to the data.

As mentioned above, LegUp is unable to determine which thread will access which of the shared memories, forcing it to take a conservative approach by assuming any thread can access any shared memory. This assumption requires the construction of expensive crossbar-like interconnect and arbitration logic between all threads and every shared memory, as illustrated in Fig. 1(b1). In the current generated hardware, whenever multiple threads compete for a shared memory, only one can be granted access. The rest of the threads are stalled, unable to make progress.

3.2 Memory Partitioning Schemes

To exploit the high internal memory bandwidth available on FPGA devices, LegUp includes a partitioning optimisation that can split a shared memory into multiple smaller memories [21]. Memory partitioning allows multiple simultaneous accesses to a shared memory between concurrently executing threads. With an appropriately chosen partitioning scheme, simultaneous memory port accesses, which would have previously resulted in contention, can now occur without stall cycles. By splitting memories into smaller blocks, each of the constituent smaller memories can service disjoint regions of the overall address space independently. Provided multiple threads are requesting access to portions of the address space serviced by different memories, they can access them simultaneously. The capability to partition memories thereby increases access parallelism, improving application performance.

If partitioning is well balanced, with threads accessing disjoint regions of the address space concurrently, then performance is improved. However, if some of the banks are very "hot" with frequent accesses and others are very "cold" with infrequent accesses, then the overheads of the additional circuitry may outweigh the benefit seen by the increased throughput. To select appropriate partitioning strategies, LegUp adopts an automated simulation-trace-based approach using a light-weight memory simulator. Different partitioning schemes, `complete`, `block`, `cyclic` and `block-cyclic`, are applied and simulated on the initial memory trace to experimentally identify the partitioning strategy with the lowest memory contention.

While this approach greatly improves the throughput for concurrent shared-memory applications, it cannot guarantee that multiple threads never access the same partition. This exacerbates the scalability challenge discussed in Section 3.1, as now the complexity of the arbitration logic scales not just with the number of shared memories and threads, but also with the number of banks per shared memory. Our analysis alleviates this scalability challenge, as it is able to prove which threads cannot access which banks of a shared memory through a static analysis of the code. EASY enables safely optimising the arbitration logic between memory banks andthreads, reducing its complexity and raising performance.

TABLE 1
Program analysis approaches for memory partitioning in HLS.

		Approaches for memory partitioning		
		Polyhedral analysis	Simulations	Formal methods
Input code	Single-threaded	[22], [23], [24]	[25]	[25], [26]
	Multi-threaded		[21]	Our work

3.3 Automatic Analysis of Memory Partitioning

Program analysis has been an active research area for optimising circuit generation with HLS tools. Table 1 shows a comparison between our work and the following related works. Polyhedral techniques are popular for memory analysis [27], and have been extended to HLS by the work of Liu *et al.* [28]. Recent work such as Wang *et al.* [22] on polyhedral analysis for memory partitioning has shown promising results through performing cyclic-like partitioning, exploring the transformation between accesses to an original multi-dimensional array and to the newly partitioned arrays. However, this approach is incompatible with bank switching, where the data in one bank is shared by multiple hardware units, prompting Gallo *et al.*'s lattice-based banking algorithm [23]. Winterstein *et al.* [26] propose a tool named MATCHUP to identify the private and shared

memory region for a certain range of heap-manipulating programs using separation logic, targeting off-chip memory, while we perform analysis of arbitrary code but currently only work with on-chip memory. The most recent work by Escobedo and Lin [24] proposes a graph-based approach using formal methods to compute a minimal memory banking for stencil-based computing kernels to avoid memory contention. In summary, these works are not intended for multi-threaded input, but are rather directed towards partitioning of arrays with multiple accesses in a loop body. Moreover, these techniques often require code to have a particular structure (*e.g.* polyhedral loop nests). The detailed comparison between the prior work [21] targeting multi-threaded code and our work is explained in Section 3.2.

Satisfiability Modulo Theory (SMT) solver-based approaches are relatively new to the HLS community but have shown potential strengths in hardware optimisation. The closest piece of work to this paper is by Zhou *et al.* [25], which proposes an SMT-based checker for the verification of memory access conflicts based on parallel execution with banked memory, resulting in a highly area-efficient memory architecture. Both works use simulation traces as a starting point for formal analysis, an approach referred to as 'mining' by Zhou *et al.* [25].

However, Zhou *et al.* [25] only analyse a loop kernel, without taking the control structures of the whole program into account, and formulate an SMT query on the banking functions themselves. Hence, their tool does not support input-dependent memory traces, which can be analysed by EASY. Moreover, their tool optimises memory access conflicts and reduces MUX overhead for a single loop schedule, while EASY optimises the memory arbitrations among multiple threads. Finally, Zhou *et al.* [25] prove the existence of bank conflicts, allowing each parallel hardware unit to access different memory banks concurrently. Conversely, we identify the memory banks *never* accessed by certain threads over the whole execution, because we target arbiter removal. In this article, we directly use the method by Chen and Anderson [21].

3.4 NoC Approaches for Memory Partitioning

In LegUp, each memory bank has an arbiter connected to all threads. The trace-based analysis in LegUp [21] allows the arbiters to predict the memory request from threads. However, for complex cases like indirectly-addressed arrays, the static analysis is limited and the memory contention can still be frequent. Solutions can be found in two aspects: dynamic approach and static analysis. Islam and Kapre [29] propose a dynamic solution to the inefficiency of memory arbiters from LegUp HLS tool when synthesising hardware with indirect array addressing. They replace the original round-robin arbiters by Networks-on-Chip (NoCs), which are router-based packet-switching networks, for memory arbitration enabled by routing over a packet-switched fabric. One main advantage of such a dynamic approach is that they can handle unpredictable cases efficiently. However, they still need the arbiters between each memory bank and all threads, while EASY aims to achieve minimal area, optimally zero area, in arbitration logic.

3.5 Microsoft Boogie

Boogie is an automatic program verifier from Microsoft Research, built on top of SMT solvers [7]. Boogie uses its own intermediate verification language (IVL) to represent the behaviour of the program being verified. Instead of executing the program, an SMT solver is applied to reason about the program's behaviour, including the values that its variables may take. Encoding of verification as SMT queries is automatically performed by Boogie, hidden from the user. Other works have proposed the automated translation of an original program to Boogie IVL, such as SMACK [30], an automated translator of LLVM-IR code into the equivalent Boogie program. We built our own code generation because our Boogie program is specifically designed to solve the memory banking problem. EASY approximates the memory access patterns with the given parameters, while SMACK proves the exact program behaviour.

The effectiveness of verification in EASY depends on the Boogie verifier itself. Using Boogie is beneficial for analysis of non-linear memory accesses like `A[(i%10) ? i*i:i]`, which is not supported in common techniques like polyhedral analysis. Also, the worst case of EASY does not cause larger area but the same hardware design as the baseline.

4 METHODOLOGY

This section explains how EASY generates a Boogie program for verification. First, we show how the problem of memory arbitration is formalised. Then we introduce the Boogie constructs that are used by EASY. Next, we illustrate how a Boogie program describes the memory behaviour and proves the absence of access relations. We also show how Boogie efficiently handles loops. Finally, we explain how supports for indirectly addressed arrays are implemented.

4.1 Problem Formalisation

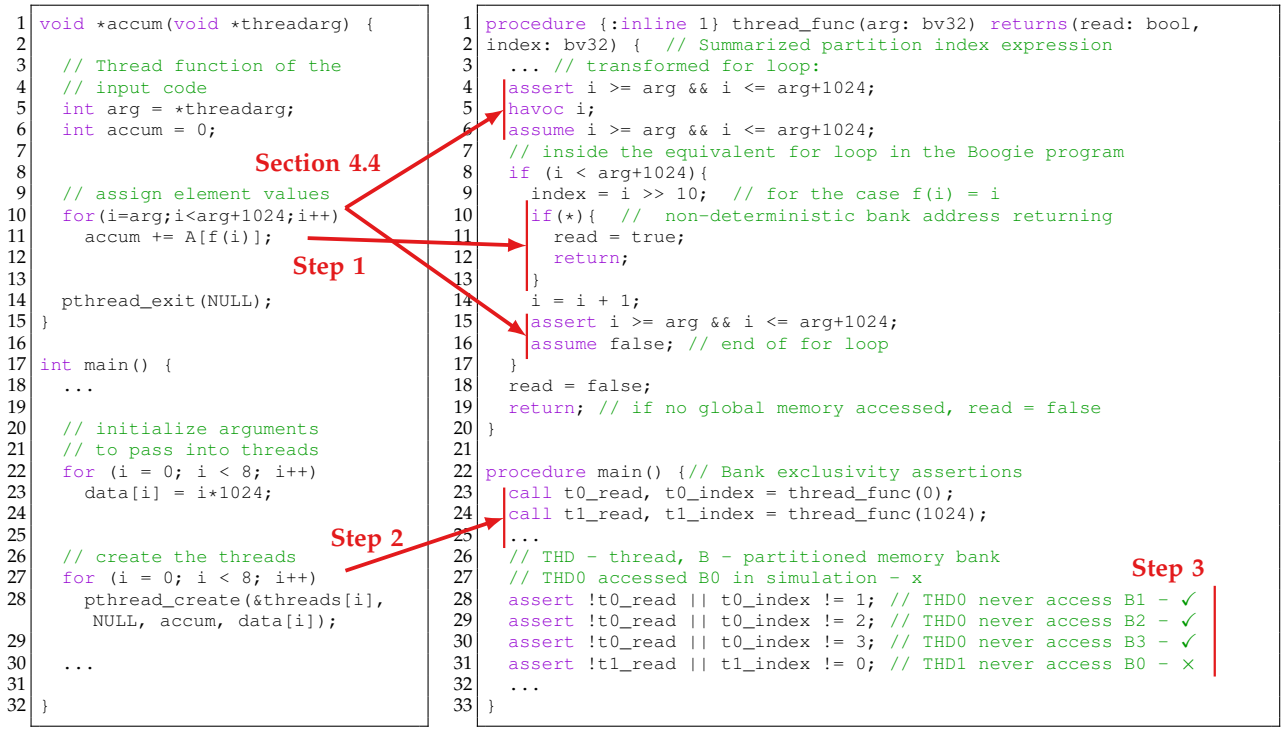
To formalise the problem, we define the following symbols:

- T – the set of threads;
- S_b – the set of threads that do not access block b during simulation;
- G_b – the set of threads that cannot access block b, also known as the "ground truth"; and
- F_b – the set of threads that have been formally proven by EASY to never access block b.

The original arbitration method for partitioned memory [21] is to build an arbiter with $|T|$ connections, while our work builds an arbiter with $|T - F_b|$ connections. The 'best case' is $|T - F_b| = 1$, *i.e.* the bank is exclusive to a certain thread and no arbiter is needed. The aim of our work is to find F_b for each bank.

Since $G_b \subseteq S_b \subseteq T$, the objectives of our work in finding F_b are mainly two aspects: 1) soundness, which means that $F_b \subseteq G_b$. This is essential for ensuring the results are correct; 2) precision, which means that amongst all those F_b that are sound, we want F_b as large as possible, that is, we want $F_b = G_b$ in the best case.

To ensure the soundness of our approach, we choose to use Boogie and its assertions. This formal verification intermediate language description allows us to over-approximate the memory behaviour for F_b, *i.e.* $F_b \subseteq G_b$. In



(a) Input multi-threaded C code, M

(b) Output Boogie program, $transform(M)$.

Fig. 4. The Boogie program $transform(M)$ is generated from the C program M. Each assertion in $transform(M)$ encodes that one of the threads in M never accesses one of the memory partitions. Whenever the Boogie Verifier is able to prove one of these assertions, one of the arbitration wires can be safely removed when synthesising M.

the following sections, we show how to use constructs like non-deterministic choices, loop interpretations and an indirect addressing formulation to cover the possible memory accesses. In addition, our approach focuses on the accuracy in the description of the memory behaviour to get minimum $G_b - F_b$. Over the benchmarks we evaluated, we have $F_b = S_b$, and since $F_b \subseteq G_b \subseteq S_b$, it follows that $F_b = G_b$.

4.2 Boogie Constructs

We use a Boogie program to verify the memory arbitration problem. In addition to all the commands one would expect in a standard programming language, Boogie contains a number of verification-specific language constructs. Here we list those used in our work and therefore explain below:

1) `havoc x;` The `havoc` command assigns an arbitrary value to the variable `x`. This can be used to prove an assertion that is true for *any* value of the variable, unlike simulation-based testing which will only check assertions for particular test vectors.
2) `assume c;` The `assume` command tells the verifier that the condition `c` can be assumed to be true when trying to prove subsequent assertions. For example `{havoc x; assume (x>0);}` together encode that the variable `x` can be any positive value.
3) `if (*) {A} else {B};` The special `(*)` condition tells the verifier that either branch might be taken. This construct is called non-deterministic choice.
4) `assert c;` This instructs the verifier to try to prove the condition `c`. For example `{havoc x; assume (x>1); assert (x>0);}` should pass, because every variable greater than one is also greater than zero.
5) `forall x :: c1(x) ==> c2(x);` This condition holds if condition `c2` holds for all values of `x` that satisfy condition `c1`. These conditions can be used with `assume` for assumptions or `assert` for verification.

4.3 Generating a Boogie program

Using the Boogie primitive operations outlined in Section 4.2, we can generate a Boogie program that can be used to verify bank exclusivity in our multi-threaded input code. Our approach is fully automated – an input LegUp multi-threaded source is automatically analysed, resulting in a Boogie program as part of the compiler pass. This code generation consists of four main steps as follows:

Step 1: We use non-deterministic choice to exhaustively explore the state-space of all possible memory accesses.
Step 2: For each hardware-accelerated thread call in the `main` function we call a separate instance of the single-threaded Boogie program.
Step 3: Within each Boogie thread instance, we generate Boogie `assert` statements that are used to test for memory bank exclusivity.
Step 4: We describe the data of the address array being overwritten in a loop using additional loop invariants with `forall` constructs.

In this section, we explain the first three steps with a code example shown in Fig. 4. Step 4 is explained in Sec-

tion 4.5. Based on prior work on memory partitioning [21], we introduce several terms that are used in our model:

Partition index — The address of the memory partition.
Local address — The address of the requested data in the partition. The array index in the input program is a function of the partition index and the local address.
Address array — The array that controls the partitioned array accesses.

For instance, in Fig. 4, we have `(i >> 10)` as the partition index, the 10 LSBs of `i` as the local address, and no address array. In Fig. 6, the address array is array `B`.

Step 1: A procedure named `thread_func` represents the memory behaviour of the original thread function (named `accum` in the example code). From the input code, it is intuitive that a thread accesses the array index range from `A[i]` to `A[i+1023]` in the `for` loops when `f(i) = i`. In the equivalent loop in `thread_func`, the partition index is required to identify which partition is accessed. In this example, the partitioning index is bit 10, 11 and 12 of the 13-bit array index, but the expression can be arbitrary in general. The goal of the Boogie program is to prove whether the partition index value in any iteration cannot equal to the partition index of a partitioned bank, *e.g.* $\forall i \in [arg, arg + 1024), partition\_index(i) \neq B$. This means that the current access at address `i` cannot touch the partitioned bank with a partition index of B across all the loop iterations.

A novelty of this work is that we use the non-deterministic choice concept `if(*)` to model the fact that we need to capture the partition index accessed by *any* loop iteration. If the verification succeeds, we can deduce that the assertions hold whichever way this non-deterministic choice is resolved. Taking the `if` branch causes the verifier to flag that a read has happened with partition index `index` – if the branch is not taken, this corresponds to skipping this particular memory access in the original code. Such arbitrary behaviour allows Boogie to jump to any loop iteration and capture the memory behaviour in any iteration. For this example, any loop iteration in thread 0 can only return `index = 0` because the set of all possible partition indices is $\{0\}$ for array indices ranging from 0 to 1023. Similarly thread 1 only returns `index = 1`. The use of `assume false` instructs the verifier to ignore the rest part of the function since the partition index is invalid when reaching this point, that is, all the `if(*)` are skipped.

The generated Boogie program only describes the program behaviour of the partitioned memory for the memory arbitration problem. EASY uses the pre-existing slicing tool by Fleming and Thomas [31] to automatically extract the memory behaviour from the input code. The sliced code is a list of instructions that affects the partitioned memory access, disregarding all other irrelevant instructions in the thread function. When the data in an array is accessed such as `A[f(i)]` in Fig. 1(a), the corresponding instruction is represented as:

```
1 index = partition_index;
2 if (*) { // arbitrarily true
3     read = true;
4     return;
5 }
```

in Boogie to capture its behaviour across all the loop iterations.

Step 2: Two further parts of the Boogie model appear in the `main` procedure. The generated Boogie program sequentially calls each thread procedure. These call instructions are separate and independent modules for memory access analysis. Each call instruction returns a read state and an arbitrary partition index among accessed banks. Thus, each called thread procedure returns two variables named `tX_read` and `tX_index`. The `tX_read` indicates whether the returned partition index is valid. It is invalid when all the `if(*)` blocks are skipped. The `tX_index` indicates one of reachable partition indices. EASY then uses assertions to prove the properties of these two variables to identify the set of partitioned banks that this thread cannot access.

Step 3: The final part of the verification code is a list of final assertions. EASY automatically generates these final assertions by enumerating the connections between each partitioned bank and each thread. Each single assertion states that a specific thread cannot return a specific partition index value, *i.e.* that thread cannot touch a specific partitioned bank. If the assertion holds, the corresponding arbitration logic can be removed in the hardware, otherwise, it is necessary to maintain this arbitration wire in the hardware. The prior work on memory partitioning [21] has observed some memory accesses in simulations, indicating which thread can access which partitioned banks. EASY reuses these memory accesses to remove the assertions as an optimisation, as they do not hold. For instance, if bank 0 has been accessed by thread 0 in the simulation, the assertion indicating that thread 0 never accesses bank 0 certainly fails and is removed in the Boogie program.

With a Boogie program containing a list of final assertions that are not observed in the simulation, the Boogie verifier is automatically called to filter all failed assertions. Based on the successful assertions left in the Boogie program, the verification results can correctly identify where arbitration is required in the HLS-generated hardware. When the memory accesses are complex, the Boogie verifier may not be able to prove an assertion that is nonetheless true, and some arbiters might be left in unnecessarily. Even so, in the worst case, our approach does not make the original hardware design worse.

4.4 Handling Loops

Memory accesses in loops are a primary source of memory bottlenecks, as they often correspond to the overwhelming majority of accesses. In our analysis, we aim to support loops without having to unroll them in the Boogie program, in order to support general `while` loops and also to keep the size of the verification code small. A loop in a Boogie program requires the programmer to specify a *loop invariant* to formally abstract the program state. An invariant is a property describing the program state that always holds on

```
1 while (c)
2 invariant phi;
3 {
4   B;
5 }
```

(a) Input code.

```
1 assert phi; // base case
2 havoc modset(B);
3 assume phi;
4 // inductive hypothesis
5 if (c) {
6   B;
7   assert phi; // step case
8   assume false;
9 }
```

(b) Generated Boogie code.

Fig. 5. Loop summary in Boogie using loop-cutting transformation [32].

TABLE 2
Reference table for invariants of `for` loops of the form `for( i=start; cond; step ).`

Loop form	Guessed invariant
`for (i=start; i<end; i++)`	`start<=i && i<end`
`for (i=start; i>end; i--)`	`start>=i && i>end`
`for (i=start; i<=end; i++)`	`start<=i && i<=end`
`for (i=start; i>=end; i--)`	`start>=i && i>=end`

entry to the loop and after every iteration of the loop. Automated generation of loop invariants is an active research area in program verification. Here we adopt the approach described by Chong [32].

Fig. 5 shows the general case of our loop transformation process: Fig. 5(a) shows the general structure of a `while` loop, and Fig. 5(b) shows the equivalent transformed loop in Boogie. In Fig. 5(a), a `while` loop contains a loop condition `c` and a loop body `B`. Additionally, `phi` represents the loop invariant. In Fig. 5(b), the invariants for the loop are established inductively. At the entry point of the loop at line 1, also known as the base case, the first assertion asks Boogie to verify that the loop invariant holds as a precondition for the loop, where an example is shown as line 4 in Fig. 4(b). The code in line 2 skips an arbitrary number of loop iterations, jumping to an arbitrary iteration of the loop. We use `havoc` to select arbitrary values for a set of variables that have carried dependences in the loop body `B`, known as the 'modified set', `modset(B)`. Then the induction hypothesis is `assume`'d in order to restrict these arbitrary values still make the loop invariants hold.

The loop invariants still hold after each loop iteration, verified by the assertion on line 7. Once the verifier has proved the assertion, we do not want the effects of executing `c` or `B` to be visible to code that may come after the loop. So, we issue an `assume false` on line 8, which has the effect of 'cancelling' this control flow path. Hence, any code that comes after the loop can assume that the if-statement on line 5 was *not* taken, *i.e.* that we have exited the loop. In summary, our formulation describes *one* arbitrary iteration of the whole loop, instead of the whole loop iteration. A main advantage is that it has short code length and the verifer can still prove the property of loop by capturing *any* iteration.

The selection of an appropriate invariant `phi` is key to verification success. We find that the loops we encounter in practice have a simple structure of increasing or decreasing single-strided for-loops with strict or non-strict inequalities as loop exit conditions. Therefore, it is sufficient to automatically derive the loop invariants from the exit conditions of `for` loops for most HLS applications. We implement a simple lookup table of proposed loop invariants, shown in Table 2. The loop invariants are derived from loop conditions, but not the same as loop conditions. The difference is the non-strict inequality in the upper bound due to the final iteration. EASY checks the properties of the loop iterator, *i.e.* the striding behaviour and the exit bound, and generates the guessed loop invariants for any `for` loop in the input code. We note that it is safe to *guess* loop invariants without being concerned about whether they are correct, because we place an obligation on Boogie to verify that a guessed invariant actually does hold.

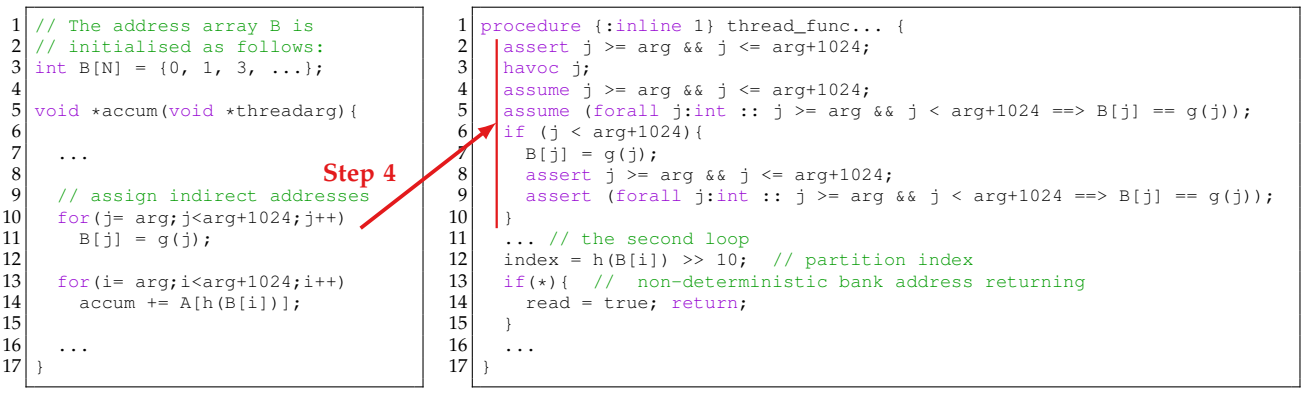
4.5 Indirect Array Addressing

In the preceding sections, we explained how to reason about the memory behaviour of a directly-addressed array. A directly-addressed array is one where the expression of an array index only depends on register values and does not depend on any array data. For instance, in Fig. 4, `f` is independent of any array data but only depends on `i`. An indirectly-addressed array is one where the value of its array index is dependent on data in one or more arrays. Indirect array indexing is used in applications such as sparse matrix multiplication and 3D rendering [33]. In this section, we show how to prove the memory behaviour of an indirectly-addressed array with additional step 4 listed in Section 4.3.

Our verification procedure is sound but not complete, *i.e.* we can prove absence of memory accesses but not their presence. However, the lack of completeness means that the verification may be less precise, as there may theoretically still be an absence of accesses unprovable by our procedure. For instance, Fig. 6(a) shows a similar example to Fig. 4 but it contains two arrays `A` and `B`. In the first loop, the elements in array `B` are assigned with a function `g` of the loop iterator `j` in a directly-addressed format. Subsequently, in the second loop, the values of elements in array `A` are summed. Array `A` is partitioned and indirectly-addressed by the data stored in array `B` in the form of `h(B[i])`. Unlike Fig. 4, the index of array `A` depends on both the loop iterator `i` and the data in another array `B`. Hence proving the memory behaviour of array `A` requires not only the expression of function `h` but the data stored in array `B`. Therefore, the verifier to assume that `h(B[i])` in any iteration can be an arbitrary integer, which could be a smaller set of values like `h(B[i]) == i`.

Step 4: We solve this by including the information of the address array for the indirectly-addressed partition array. Firstly, with a given partitioned array, EASY identifies its memory accesses in the thread function and uses the *slicing* tool to extract all the related instructions. If the address of a memory access depends on any *load* instruction, the memory access is indirectly addressed. Knowing that, we include information about the loaded data in the array in the Boogie program. In this example, we find the index of array `A` depends on the data in array `B`, so we describe the data in array `B` in the Boogie program, as shown in Fig. 6(b).

For simplicity, we assume `g(i) = i` and `h(i) = i`, but they can be any arbitrary functions in the general case. Fig. 6(b) shows the Boogie program generated from the




```
1  // The address array B is
2  // initialised as follows:
3  int B[N] = {0, 1, 3, ...};
4
5  void *accum(void *threadarg){
6
7    ...
8
9    // assign indirect addresses
10   for(j= arg;j<arg+1024;j++)
11     B[j] = g(j);
12
13   for(i= arg;i<arg+1024;i++)
14     accum += A[h(B[i])];
15
16   ...
17 }
```

(a) Input multi-threaded C code, M

```
1  procedure {:inline 1} thread_func... {
2    assert j >= arg && j <= arg+1024;
3    havoc j;
4    assume j >= arg && j <= arg+1024;
5    assume (forall j:int :: j >= arg && j < arg+1024 ==> B[j] == g(j));
6    if (j < arg+1024){
7      B[j] = g(j);
8      assert j >= arg && j <= arg+1024;
9      assert (forall j:int :: j >= arg && j < arg+1024 ==> B[j] == g(j));
10   }
11   ... // the second loop
12   index = h(B[i]) >> 10;  // partition index
13   if(*){  //  non-deterministic bank address returning
14     read = true; return;
15   }
16   ...
17 }
```

(b) Output Boogie program, $transform(M)$.

Fig. 6. The Boogie program $transform(M)$ is generated from the indirect addressing behaviour for partitioned arrays in the C program M.

input code in Fig. 6(a). The second loop that contains the accesses to the partitioned array `A` is represented as a non-deterministic returning form in an arbitrary loop iteration like Section 4.3. This representation allows `thread_func` to arbitrarily return one of all possible bank addresses if the access is valid (`read == true`). The additional work for the indirectly-addressed array is the description of the address array.

We use another approach to describe the address array, as the non-deterministic representation does not provide enough information regarding the address array. The non-deterministic choice only describes one element in the address array at a particular index `j`. For instance, in this example, the values of `j` and `i` selected in the two loops can be different. Hence when `i != j`, knowing `B[j] == j` does not help proving the access to array `A` at bank address of `h(B[i]) >> 10`.

We summarise the array data by the use of `forall` constructs as an additional loop invariant. In the example, the expression of array `B` is regular, where `B[i] = g(i)`. The additional loop invariant describes the expression of the elements in array `B` that are overwritten in the first loop. In the Boogie program shown in Fig. 6(b), lines 11 and 12 state that at the exit of the first loop, for any values of loop index `j` between `arg` and `arg+1023`, the value of array data `B[j]` is always function `g` of the corresponding loop index `j`, *i.e.* `g(j)`. This summarises all the values of `B[j]` for the verification of the following accesses to array `A` in the second loop.

The expression defining the data in the address array can also contain more constructs, and our approach still works. For instance, there may be conditional branches in the loop, such that the array elements can have various expressions. EASY also verifies the memory conflicts among conditional reads and writes by including these in the description automatically. Therefore, the Boogie representation generated by EASY both describes the array data pattern and proves that a certain set of memory banks cannot touched by a certain thread.

4.6 Summary

We have shown how to use the Boogie verifier to prove the memory arbitration problem in parallel hardware with memory partitioning. EASY automatically extracts the partitioned memory behaviour and generates a Boogie program to prove threads do not access certain memory banks. The memory access patterns can be arbitrary, and the proposed technique remains valid and correct. The verification result is a conservative over-approximation, where the tool may not be able to prove a connection is removable yet is actually unnecessary. For instance, the values of address arrays should be known at the entry of the loop, initialised by the code. However, the approach is guaranteed to not make the hardware worse.

5 Integration Into the LLVM Framework

We describe the technical details on how EASY is integrated within the LegUp HLS framework as shown in Algorithm 1. At top level, EASY takes the input program M and generates a Boogie program B. Then the verification of B guides EASY to transform M to a more efficient architecture M'.

Before generating B, the analysis extracts the following parameters: the set of threads T, the partitions for partitioned arrays A, the loop invariants for loops L and the observed memory accesses by simulation S (line 4-7). S observed through simulation provides an under-approximation of access relation. With these parameters, EASY extracts all the partitioned memory accesses, where $i \rightarrow a$ means that an instruction i touches array a, and constructs (the left arrow) a set I (line 8). Then it removes all the irrelevant instructions to I, and constructs a sliced program M_S (line 9). The remaining program M_S is used to generate B.

The generation of B contains two parts. First, for the `main` procedure (line 11-14), each instruction in M_S is translated into an equivalent Boogie instruction and added to B. At the end of the procedure, EASY enumerates the assertions for all the access relation between the threads T and the partitioned arrays A except those that have already been observed in S. These assertions are also added to B like Fig. 4(b). Second, for the thread functions (line 16-22),

Algorithm 1: Algorithm of EASY.

```
1  M: Input program module ;
2  B: Boogie program ;
3  M': Output program module ;
4  Extract the set of threads T in M ;
5  Extract partitioned arrays A ;
6  Extract L, the set of basic blocks of M corresponding to
     loop entries and exits ;
7  Load observed access relation S ⊆ T × A ;
8  I ← {i ∈ M | ∃a ∈ A. i → a}
9
10 M_S ← sliceProgram(M, I) ;
11 for each function f in M_S do
12     if f ≠ main then
13         // Step 2 from Fig. 4
14         for each basic block bb in f do
15             // Step 4 from Fig. 4
16             if bb ∈ L then
17                 B.addLoopInvariants(bb) ;
18             for each instruction i in bb do
19                 B.addInstr(i) ;
20                 // Step 1 from Fig. 4
21                 if i → array a ∈ A then
22                     B.addNonDeterministicChoice(a) ;
23     else
24         // Step 3 from Fig. 4
25         for each instruction i in f do
26             B.addInstr(i) ;
27         B.addAssertions(T × A − S) ;
28
29 F ← verify(B) ;
30 M' ← removeArbiterPorts(M, F) ;
31 return M'
```

at the entry of the basic blocks, EASY checks whether the current block is a loop header or exit as stated in Section 4.4 and Section 4.5. If true, it inserts the corresponding loop invariants to B. Also, in each basic block, every instruction is translated into an equivalent Boogie instruction. If an instruction accesses a partitioned array, a non-deterministic choice is also added to the Boogie program (line 21-22).

Once B is generated, EASY verifies B by calling Boogie verifier and constructs a set of access relations F for all the banks that can be safely removed (line 23). Then it takes the original program M and removes these accesses F, constructing a more efficient program M'. An example of how these accesses are removed is shown in Fig. 7. The left side of the figure is part of the code in M, and the right side is the corresponding hardware from the code on the left. In hardware, a multiplexer used to access the partitioned memory, which is represented as a number of `select` instructions in M. For the case of the current thread being proven to not access sub-arrays 1 and 2, EASY removes the corresponding selecting instructions in the input code to construct M'. The final hardware is then simplified, where red '×'s show the arbitration wires that are safely removed.

6 Experimental Results

The FPGA family and synthesis tool that we used for result measurements including Fig. 2 are Cyclone V (5CSEMA5F31C6) and Quartus II 15.0.0, as this FPGA is one of the devices supported by the LegUp HLS tool. We evaluate the arbitration simplification process on a set of benchmarks, assessing its impact on both circuit area and speed: F_{max}, latency in cycles, and wall-clock time (latency in cycles $\times$ $1/F_{max}$). We also discuss its impact on CAD tool run-time.

6.1 Benchmarks

The goal of HLS is to automatically produce architectures from software descriptions. Our benchmarks focus on the support for explicit parallelism via threads. When combined with other HLS transformations, advanced architectures like systolic architectures can be synthesised from code expressed through thread-level parallelism. We apply our approach to the nine multi-threaded benchmarks with directly addressed partitioned arrays:

matrixadd sums one integer matrix of size 128 $\times$ 128 by blocking into groups of row summations, each performed by a different thread.

histogram reads an input integer array of size 32768 and counts the number of elements in five distinct ranges, storing the final element distribution in a result array.

matrixmult multiplies two integer matrices of size 128 $\times$ 128. The element operations are divided into groups of row multiplications for parallelism.

matrixmult (cyclic) is the same as `matrixmult` but use the cyclic memory partitioning scheme, grouping rows with matching LSBs to be operated on by a single thread.

matrixtrans computes the transpose of an input matrix of size 128 $\times$ 128 following the cyclic scheme.

matrixtrans (block cyclic) is the same as `matrixtrans`, but the row allocation to different threads is based on both MSBs and LSBs of the index in a block-cyclic partitioning scheme, where for instance, a thread transposes rows at addresses of 0-3, 32-35, 64-67 and 96-99.

substring searches for a string pattern of size 3 within an input string of size 2048, counting the number of occurrences of this pattern. The input string has been divided into several continuous substrings for multi-threaded execution. The arbitration complexity is relatively high due to there being multiple memory statements in a single loop accessing the same partitioned array.

los analyses the presence of an obstacle between the elements in a predefined obstacle map of size 64 $\times$ 64 and centre of the map, wherein an element with value 1 indicates an obstacle, while an element with value 0 represents free space. The analysis of the elements is distributed to several threads for parallelism and the resultant output is a map, where elements having value 0 represent the presence of obstacles between the test coordinates and centre point, while 1s are verified line-of-sight cases. This benchmark has a loop-carried dependency at the thread level, and an infinite `while` loop is used with two conditional breaks, leading to more complex partition index expressions.

fft performs the fast Fourier transformation for each row in a 16 $\times$ 256 matrix.

We also obtained the results from four other benchmarks in which the partitioned arrays are indirectly addressed.

```
1  %par_idx = ashr i32 %i, 13
2  %pred_0 = icmp eq i32 %par_idx, 0
3  %pred_1 = icmp eq i32 %par_idx, 1
4  %pred_2 = icmp eq i32 %par_idx, 2
5  %pred_3 = icmp eq i32 %par_idx, 3
6  %addr = and i32 %i, 8191
7  %GEP_0 = getelementptr * input_sub_array_0, i32 %addr
8  %GEP_1 = getelementptr * input_sub_array_1, i32 %addr
9  %GEP_2 = getelementptr * input_sub_array_2, i32 %addr
10 %GEP_3 = getelementptr * input_sub_array_3, i32 %addr
11 %Load_0 = load i32* %GEP_0
12 %data_0 = select i1 %pred_0, i32 %Load_0, i32 0
13 %Load_1 = load i32* %GEP_1
14 %data_1 = select i1 %pred_1, i32 %Load_2, i32 0
15 %Load_2 = load i32* %GEP_2
16 %data_2 = select i1 %pred_2, i32 %Load_2, i32 0
17 %Load_3 = load i32* %GEP_3
18 %data_3 = select i1 %pred_3, i32 %Load_3, i32 0
19 %Or_0 = or i32 %data_0, %data_1 i32 0
20 %Or_1 = or i32 %data_2 i32 0, %data_3
21 %Or_2 = or i32 %Or_0, %Or_1
```

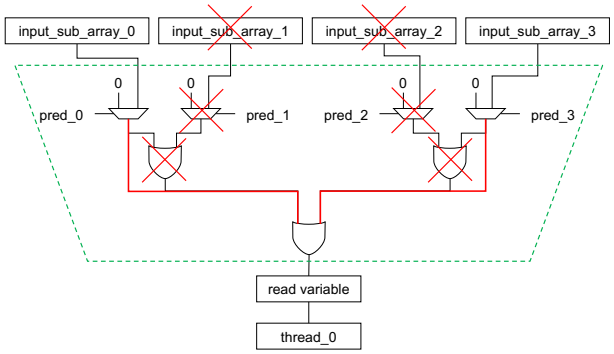


Fig. 7. EASY automatically modifies the source at LLVM-IR level on the left, which results in a simplified hardware architecture on the right.

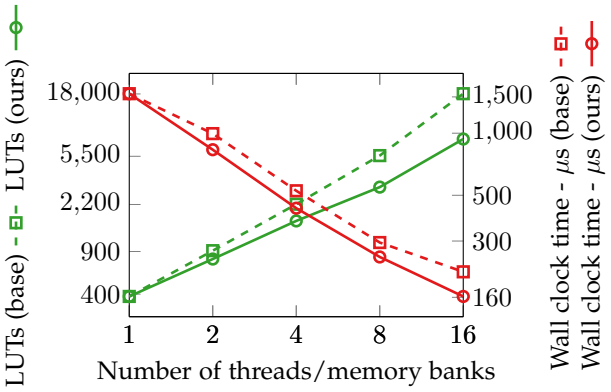


Fig. 8. Evaluation of area and performance on `histogram` hardware with equal numbers of threads and banks.

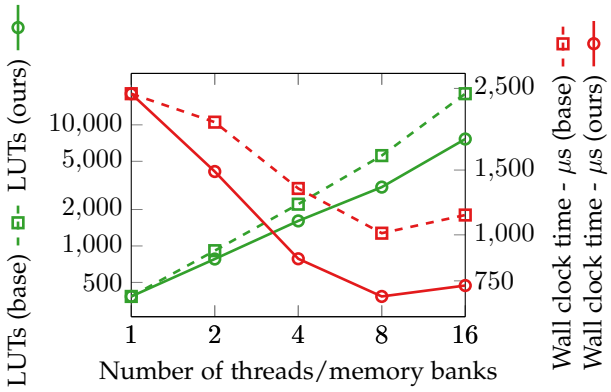


Fig. 9. Evaluation of area and performance on `ida_histogram2` hardware with equal numbers of threads and banks.

ida_histogram1 has a memory pattern of `b[i]*T + k` for the partitioned array, where `T` is number of threads and `k` is the thread ID. So regardless of the values stored in array `b`, each thread always accesses a certain partitioned bank if a cyclic partitioning scheme is applied.

ida_histogram2 has memory behaviour depending on `b[i]` with a single expression.

ida_histogram3 is similar to `ida_histogram2`, but `b[i]` has a conditional expression.

sparse_matrixmult performs sparse matrix multiplication. The index of the partitioned array is also indirectly addressed but through two pointer arrays, of the form `JA[IA[k]]`.

These benchmarks present some common program structures in modern HLS applications such as indirect array addressing, array overwriting, and conditional branches.

6.2 Case Study 1: Directly Addressed Histogram

Overall computational performance is maximised when the degree of computational parallelism matches the degree of parallelism provided by the memory system. In the simplest cases, this is often achieved when there is an equal number of partitioned banks and threads, with each thread operating on a private portion of the data. Fig. 8 shows the wall-clock time and LUT utilisation of the optimised design (with efficient memory arbitration) as compared to the original architecture, for the `histogram` benchmark. Observe that the wall-clock time appears to have a reciprocal relationship with number of threads, where more threads with sufficient memory bandwidth and no memory contention results in faster hardware execution. However, due to the increased number of parallel hardware units, hardware utilisation increases quadratically. After arbiter simplification, both performance and chip area are generally improved with an increasing gap as the number of threads increases. Although the total number of clock cycles is not noticeably reduced, the critical path is optimised, improving wall clock time by up to 27%. The chip area also decreases compared to the original design by up to 58%. Since this work modifies the arbitration hardware alone, memory block usage is unchanged. However, the hardware resources for the arbitration circuit, namely LUTs and registers, are reduced appreciably. More importantly, with more threads, the improvements in performance and chip area are more effective as more arbitration wires may be removed.

TABLE 3
Arbitration simplification evaluation for 16 memory banks and 16 threads. The baseline (written 'base') is the hardware from the tool flow without our work, *i.e.* it has the memory arbitration like Fig. 2

Benchmark	LUT Count			Register Count			Max Clock Frequency (MHz)			Total clock cycles			Wall Clock Time (μs)			Speedup v.s. 1 thread 1 bank	
	base	ours	×	base	ours	×	base	ours	×	base	ours	×	base	ours	×	base	ours
histogram	18.1k	7.64k	0.42×	22.3k	13.5k	0.61×	71.8	94.5	1.32×	15.3k	14.7k	0.96×	213	155	0.73×	7.3×	10.0×
matrixadd	14.3k	2.26k	0.16×	14.6k	4.18k	0.29×	73.7	107	1.45×	2.12k	2.12k	1.00×	28.8	19.8	0.69×	7.5×	10.9×
matrixmult	21.0k	3.64k	0.17×	15.9k	5.18k	0.33×	74.0	85.9	1.16×	2.13M	2.13M	1.00×	28.8k	24.8k	0.86×	2.1×	2.5×
matrixmult(cyclic)	20.9k	11.2k	0.54×	15.9k	10.5k	0.66×	73.8	83.1	1.13×	2.13M	2.13M	1.00×	28.8k	25.6k	0.89×	2.1×	2.4×
matrixtrans	18.6k	2.36k	0.13×	14.4k	3.55k	0.25×	61.3	92.3	1.51×	37.2k	36.2k	0.97×	607	392	0.65×	1.2×	1.9×
matrixtrans(blockcyclic)	10.6k	9.10k	0.86×	9.00k	7.40k	0.82×	75.5	75.5	1.00×	62.9k	61.7k	0.98×	833	817	0.98×	0.9×	0.9×
substring	11.0k	2.56k	0.23×	12.0k	4.72k	0.39×	77.4	125	1.62×	443	439	0.99×	5.72	3.50	0.61×	7.0×	11.5×
los	23.8k	20.6k	0.87×	31.3k	25.1k	0.80×	73.8	81.5	1.10×	46.5k	45.4k	0.98×	630	557	0.88×	5.7×	6.4×
fft[3]	25.6k	23.0k	0.90×	36.3k	33.1k	0.91×	108	122	1.13×	413k	413k	1.00×	3.84k	3.39k	0.88×	1.7×	2.0×
ida_histogram1	23.4k	5.88k	0.25×	31.7k	8.05k	0.25×	73.1	91.4	1.25×	33.9k	33.8k	1.00×	464	370	0.80×	4.6×	5.7×
ida_histogram2	17.5k	6.45k	0.37×	22.2k	8.49k	0.38×	58.7	91.1	1.55×	66.4k	66.4k	1.00×	1.13k	728	0.64×	2.0×	3.2×
ida_histogram3	17.5k	6.62k	0.38×	22.4k	8.56k	0.38×	58.1	87.8	1.51×	66.4k	66.4	1.00×	1.14k	756	0.66×	2.0×	3.0×
sparse_matrixmult	16.8k	6.03k	0.36×	18.9k	8.70k	0.46×	87.3	96.6	1.11×	5.62k	5.58k	0.99×	64	57.8	0.90×	2.9×	3.2×
geom. mean	-	-	**0.43**×	-	-	**0.50**×	-	-	**1.30**×	-	-	**0.99**×	-	-	**0.78**×	-	**4.90**×

TABLE 4
Results of Table 3 for arbitration logic only.

Benchmarks	LUT Count			Register Count		
	base	ours	×	base	ours	×
histogram	12.0k	1.51k	0.13×	13.1k	4.29k	0.33×
matrixadd	12.4k	389	0.03×	10.9k	528	0.05×
matrixmult	19.4k	2.00k	0.10×	10.8k	144	0.01×
matrixmult(cyclic)	15.8k	6.20k	0.39×	7472	2079	0.28×
matrixtrans	18.4k	2.12k	0.12×	12.5k	1.68k	0.13×
matrixtrans(blockcyclic)	6.99k	5.48k	0.78×	3.76k	2.17k	0.58×
substring	9.60k	1.13k	0.12×	8.22k	974	0.12×
los	14.0k	10.8k	0.77×	16.6k	10.4k	0.63×
fft[3]	20.6k	18.0k	0.88×	28.1k	24.9k	0.89×
ida_histogram1	20.6k	3.11k	0.15×	25.8k	2.08k	0.08×
ida_histogram2	14.3k	3.22k	0.23×	15.6k	1.96k	0.13×
ida_histogram3	14.2k	3.38k	0.24×	15.8k	2.02k	0.13×
sparse_matrixmult	14.4k	3.53k	0.25×	11.9k	1.70k	0.14×
geom. mean	-	-	**0.32**×	-	-	**0.27**×

6.3 Case Study 2: Indirectly Addressed Histogram

When dealing with indirectly addressed array partitioning, our work also achieves promising results. Fig. 9 shows the results for benchmark `ida_histogram2`, where each thread accesses a data array `A` at `b[i]`. The optimal solution of memory partitioning for this benchmark is to partition both array `A` and array `b` that controls the accesses to the former array. To demonstrate that our work is also effective on indirect array addressing, only array `A` is partitioned.

In Fig. 9, the increase in total area as the number of threads/banks increases is similar to that in Fig. 8. The wall clock time in seconds is defined as the number of cycles multiplied by the clock period, that is, divided by the maximum clock frequency. The wall clock time is also optimised as illustrated by the increasing gap between the red lines. When the number of threads reaches 16, the wall clock time increases compared to that with 8 threads. This is due to the fact that the number of clock cycles of the benchmark are not reducing significantly but the maximum clock frequency drops by 16%. So the total wall-clock time increases.

3. The 16-thread 16-bank fft cannot fit in CycloneV, so here are the results for the maximum thread/bank number of 2 for CycloneV.

6.4 Results for All Benchmarks

The post Place & Route results for the whole hardware of all benchmarks are given in Table 3 for the case of 16 threads and 16 memory banks.[4] The table shows LUT and register count, F_{max}, cycle latency, and wall-clock time of the whole benchmarks. We observe that all benchmarks are improved in area and performance, however, the extent of the improvement varies, depending on benchmark-specific memory-access behaviour. Significant improvements in benchmarks such as `matrixmult`, `fft` and `substring` are due to multiple accesses to partitioned arrays in one iteration, or to partitioning of multiple arrays, where the original arbitration circuits are larger leading to greater improvements. We also observe that the same benchmark with different memory partitioning schemes can have dramatically different results. For the `matrixtrans` benchmark, the *cyclic* partitioning scheme has been applied by default, which has significantly benefited from the proposed work with clock period improved by 51% and logic utilisation by 87%. However, when applying a block-cyclic scheme, the improvements are more modest. This attributed to the fact that each bank is touched by all threads during execution, so the arbiters cannot be simplified.

For the benchmarks with indirect array addressing, we only partition the array that is indirectly addressed. The results demonstrate that EASY can still prove that certain arbitration wires are unnecessary, even when the memory behaviour depends on the data in other arrays. Whether the data stored in an address array is initialised with constants or conditionally overwritten, the tool can automatically prove the data is within a certain range, so unused arbitration wires can be identified as well as directly-addressed memory. The results for benchmark `sparse_matrixmult` also show that our work supports multiple levels of indirect array addressing, where 64% of LUTs and 54% of registers are saved. Across all benchmarks, LUT count was reduced by up to 87%, and wall-clock time was improved by up to

4. Full dataset DOI: 10.5281/zenodo.1523170.

39%. Greater improvements are expected for these benchmarks with more threads. On average, wall-clock time is improved by 22%, and LUT count is reduced by 57%. The results for the arbitration logic only are shown in Table 4.

6.5 Runtime analysis

While the theoretical worst-case runtime of the approach we present is exponential in program size, in practice, the runtime of the verification process is reasonably short. The average runtime for all the benchmarks was 13 seconds. The increase in the number of threads also leads to more assertions, as well as duplicated thread procedure calls. This is directly related to the number of assertions constructed, which in turn is related to two issues: the complexity of partition memory accesses and the number of threads. For instance, if the address of the array is simple like `f(i) == i` in Section 4.3, it is easy for Boogie verifier to prove; if `f(i)` is complex enough, *e.g.* functions having various arithmetic operations and conditional branches, the verification time is longer. The longest verification time was 70s for `substring`, which has multiple memory accesses in one iteration using different partition index values resulting in multiple assertions for each access. Such verification times are insignificant compared to Synthesis/Place & Route time.

7 Conclusions

In this work, we propose an automated process to simplify and/or remove memory arbiters in HLS-generated circuits for multi-threaded software code. Our flow uses previously-proposed simulation trace-based memory banking as formal specifications for memory exclusivity which, if verified, guarantee that arbiters can be removed or simplified without impacting program correctness. Across a range of benchmarks, the execution time of the circuits has been improved by up to 39% (avg. 22%) combined with an area saving of up to 87% (avg. 57%). The performance of hardware with different numbers of threads and banks is also promising from our measurements.

The novelty of this work is in the automated procedure for optimisation of the arbitration circuits. We have shown that the behaviour of typical concurrent multi-threaded code can be over-approximated using non-deterministic choice in a sequential Boogie program, allowing existing verification tools to represent and check the verification conditions required. The runtime of the proposed compiler pass is 13 seconds, on average, across a set of 8 multi-threaded benchmarks.

One of the key advantages our approach over more structured approaches, such as polyhedral methods, is the ability to deal with arbitrary code. However, although EASY can accept arbitrary code as input, we can certainly contrive examples where it fails to prove the necessary properties without human guidance, due either to the over approximation of concurrent behaviour induced, or execution time.

Our future work will explore the fundamental limits of this approach, both theoretically and practically. For instance, the automated invariant guessing only works with `for` loops at the current stage. Inferring invariants for general `while` loops still needs human guidance. Automatically guessing the loop invariants with proper triggers can be another direction for future work, perhaps using the Daikon tool [34] and Dafny tool [35].

Acknowledgments

The authors wish to thank Jingming Hu for his assistance with result measurements. This work is supported by EPSRC (EP/P010040/1, EP/R006865/1), the Royal Academy of Engineering and Imagination Technologies.

References

[1] Microsoft Project Catapult, 2019. [Online]. Available: https://www.microsoft.com/en-us/research/project/project-catapult/
[2] Amazon EC2 F1 instances, 2019. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/
[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.
[4] V. G. Castellana, A. Tumeo, and F. Ferrandi, "High-level synthesis of memory bound and irregular parallel applications with Bambu," in *2014 IEEE Hot Chips 26 Symposium (HCS)*. Cupertino, CA, USA: IEEE, Aug 2014, pp. 1–1.
[5] Xilinx Vivado HLS, 2019. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[6] Intel HLS Compiler, 2019. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html
[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-oriented Programs," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, ser. FMCO'05. Amsterdam, The Netherlands: Springer-Verlag, 2006, pp. 364–387.
[8] J. Cheng, S. T. Fleming, Y. T. Chen, J. H. Anderson, and G. A. Constantinides, "EASY: Efficient Arbiter SYnthesis from Multi-threaded Code," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. Seaside, CA, USA: ACM, 2019, pp. 142–151.
[9] J. Cheng, S. Fleming, C. Yuting, J. Anderson, and G. Constantinides, "Dataset for EASY: Efficient Arbiter SYnthesis from Multi-threaded Code," Dec. 2020.
[10] EASY, 2020. [Online]. Available: https://github.com/JianyiCheng/EASY
[11] Intel Quartus Prime Software Suite, 2020. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/overview.html
[12] Catapult High-Level Synthesis, 2020. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
[13] Stratus High-Level Synthesis, 2020. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html
[14] Intel FPGA SDK for OpenCL Software Technology, 2020. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html
[15] V. W. Freeh, "A comparison of implicit and explicit parallel programming," *Journal of Parallel and Distributed Computing*, vol. 34, no. 1, pp. 50 – 65, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731596900453
[16] J. Choi, S. D. Brown, and J. H. Anderson, "From Pthreads to Multicore Hardware Systems in LegUp High-Level Synthesis for FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2867–2880, Oct 2017.
[17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673
[18] Clang, 2019. [Online]. Available: https://clang.llvm.org/
[19] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, July 2009.

[20] Xilinx, "SDAccel Development Environment - User Guide (v206.2)," 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf

[21] Y. T. Chen and J. H. Anderson, "Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Ghent, Belgium: IEEE, Sep. 2017, pp. 1–8.

[22] Y. Wang, P. Li, and J. Cong, "Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. Monterey, California, USA: ACM, 2014, pp. 199–208.

[23] A. Cilardo and L. Gallo, "Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 45:1–45:25, Jan. 2015.

[24] J. Escobedo and M. Lin, "Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CALIFORNIA, USA: ACM, 2018, pp. 199–208.

[25] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A New Approach to Automatic Memory Banking Using Trace-Based Address Mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 179–188.

[26] F. Winterstein, K. Fleming, H.-J. Yang, S. Bayliss, and G. Constantinides, "MATCHUP: Memory Abstractions for Heap Manipulating Programs," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. Monterey, California, USA: ACM, 2015, pp. 136–145.

[27] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[28] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic On-chip Memory Minimization for Data Reuse," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. Napa, CA, USA: IEEE, April 2007, pp. 251–260.

[29] A. Islam and N. Kapre, "LegUp-NoC: High-Level Synthesis of Loops with Indirect Addressing," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO, USA: IEEE, April 2018, pp. 117–124.

[30] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, "SMACK Software Verification Toolchain," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. Austin, Texas: ACM, 2016, pp. 589–592.

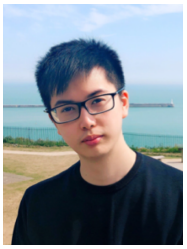
[31] S. T. Fleming and D. B. Thomas, "Using Runahead Execution to Hide Memory Latency in High Level Synthesis," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Napa, CA, USA: IEEE, April 2017, pp. 109–116.

[32] N. Y. S. Chong, "Scalable Verification Techniques for Data-Parallel Programs," Doctoral Thesis, Imperial College London, London, UK, 2014.

[33] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 269–278.

[34] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[35] K. R. M. Leino and C. Pit-Claudel, "Trigger Selection Strategies to Stabilize Program Verifiers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 361–381.



Jianyi Cheng (S'20) received an MSc in Analogue and Digital Integrated Circuit Design from Imperial College London in 2018 and a BEng in Electrical and Electronic Engineering from University of Nottingham in 2017. Currently, he is a PhD student in Electrical and Electronic Engineering at Imperial College London. His research interests include reconfigurable computing, high-level synthesis, program analysis and formal verification. He is a Student Member of the IEEE.

Shane Fleming received a PhD from Imperial College London in 2018. Since then he has worked as a research associate at Imperial College on the POETS project and in 2019 he joined Microsoft Research Cambridge as a researcher. He is now a lecturer at the Department of Computer Science at Swansea University. His research interests are in developing tools to make hardware development more agile and in developing custom hardware for datacenter infrastructure.

Yu Ting Chen (S'15) received a B.A.Sc. and M.A.Sc. in electrical and computer engineering from the University of Toronto. Her research area focused on improving memory performance in high-level synthesis generated hardware for parallel programs. She currently holds the position of senior software engineer at Tenstorrent Inc.

Jason Anderson (S'96, M'05, SM'20) received the Ph.D. degree from the University of Toronto (U of T), Toronto, ON, Canada. He joined the Field-Programmable Gate Array (FPGA) Implementation Tools Group, Xilinx, Inc., San Jose, CA, USA, in 1997, where he was involved in placement, routing, and synthesis. He is currently a Professor with the Department of Electrical and Computer Engineering, U of T. He has authored over 100 papers in refereed conference proceedings and journals, and holds 29 issued U.S. patents. He co-founded LegUp Computing in 2015 – a startup to commercialize high-level synthesis research. LegUp was acquired by Microchip Technology in 2020. His current research interests include computer-aided design, architecture, and circuits for FPGAs.

John Wickerson (M'17, SM'19) received a Ph.D. in Computer Science from the University of Cambridge in 2013. He is a Lecturer in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages, and software verification. He is a Senior Member of the IEEE and a Member of the ACM.

George A. Constantinides (S'96, M'01, SM'08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Royal Academy of Engineering / Imagination Technologies Research Chair, Professor of Digital Computation, and Head of the Circuits and Systems research group. He has served as chair of the FPGA, FPL and FPT conferences. He currently serves on several program committees and has published over 150 research papers in peer refereed journals and international conferences. Prof Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.