EASY: Efficient Arbiter SYnthesis from Multi-threaded Code

Jianyi Cheng
Imperial College London
London, UK
jianyi.cheng17@imperial.ac.uk

Shane T. Fleming
Imperial College London
London, UK
s.fleming06@imperial.ac.uk

Yu Ting Chen
University of Toronto
Toronto, Ontario
joyuting.chen@mail.utoronto.ca

Jason H. Anderson
University of Toronto
Toronto, Ontario
janders@ece.toronto.edu

George A. Constantinides
Imperial College London
London, UK
g.constantinides@imperial.ac.uk

ABSTRACT

High-Level Synthesis (HLS) tools automatically transform a high-level specification of a circuit into a low-level RTL description. Traditionally, HLS tools have operated on sequential code, however in recent years there has been a drive to synthesize multi-threaded code. A major challenge facing HLS tools in this context is how to automatically partition memory amongst parallel threads to fully exploit the bandwidth available on an FPGA device and avoid memory contention. Current automatic memory partitioning techniques have inefficient arbitration due to conservative assumptions regarding which threads may access a given memory bank. In this paper, we address this problem through formal verification techniques, permitting a less conservative, yet provably correct circuit to be generated. We perform a static analysis on the code to determine which memory banks are shared by which threads. This analysis enables us to optimize the arbitration efficiency of the generated circuit. We apply our approach to the LegUp HLS tool and show that for a set of typical application benchmarks we can achieve up to 87% area savings, and 39% execution time improvement, with little additional compilation time.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Logic synthesis**; **Modeling and parameter extraction**;

KEYWORDS

High-Level Synthesis; Memory optimization; Formal methods

ACM Reference Format:
Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason H. Anderson, and George A. Constantinides. 2019. EASY: Efficient Arbiter SYnthesis from Multi-threaded Code. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293899

1 INTRODUCTION

FPGAs are beginning to achieve mainstream adoption for custom computing, particularly as FPGAs are deployed in datacentres, through, for example, the Microsoft Project Catapult [14] and the Amazon EC2 F1 instances [3]. However, to use such FPGA devices, familiarity with detailed digital design at a low abstraction level is required, hindering their use by engineers without a hardware background. High-level synthesis (HLS) aims to bring the benefits of custom hardware to software engineers by translating a language they are familiar with, such as C, into a hardware description. This process can significantly reduce the design time compared to manual RTL implementations. Various HLS tools have been developed by both academia and industry, such as LegUp from University of Toronto [1], Bambu from Politecnico di Milano [17], Xilinx Vivado HLS [19] and Intel's HLS Compiler [8].

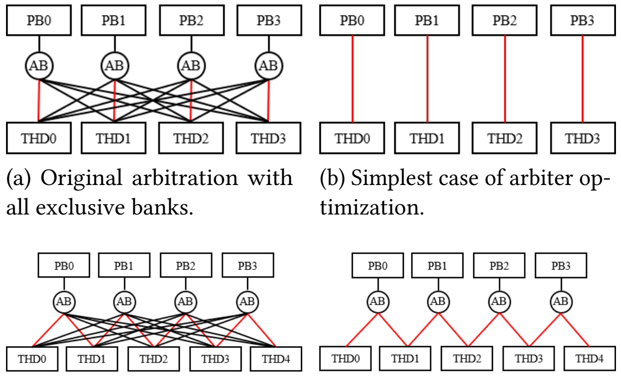
The input to HLS for parallel hardware synthesis can be either single-threaded sequential code or multi-threaded concurrent code. Our work relates to multi-threaded input, which commonly involves three challenges in HLS design. Firstly, while FPGA devices provide large amounts of compute, its effective utilization is often limited by the *memory bandwidth*. Additionally, to increase the memory bandwidth, partitioning schemes can be used to split memory into smaller distributed memories or banks. This allows for parallel accesses to data items, but to ensure *correctness*, arbitration logic needs to be used to serialize accesses to each individual partition. Finally, as the number of memory partitions or compute threads increases, so do the overheads of arbitration resulting in challenge of the *system scalability*.

HLS tools, such as LegUp, often address the memory bandwidth and correctness challenges by performing automated memory partitioning and using a crossbar arbiter to ensure global accessibility. However, this approach has scalability challenges, 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 code to specify disjoint regions of memory, which would enable the optimization 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, and such an approach is counter to the fully automated design philosophy of HLS tools.

In this paper, we propose an approach called Efficient Arbiter SYnthesis (EASY) based on automated translation of a multi-threaded program into a related sequential program in the formal verification

```
 1 void *assign(void *threadarg) {
 2   int arg = *threadarg;
 3
 4   // assign element values
 5   for (i = arg; i < arg+1024; i++)
 6     A[i] = B[f(i)];
 7
 8   pthread_exit(NULL);
 9 }
10
11 int main() {
12   ...
13
14   // initialize arguments to pass into threads
15   for (i = 0; i < 2; i++)
16     data[i] = i*1024;
17
18   // create the threads
19   for (i = 0; i < 2; i++)
20     pthread_create(&threads[i], NULL, assign, data[i]);
21
22   ...
23 }
```

Figure 1: Example of a simple multi-threaded C source using pthreads.



(a) Original arbitration with all exclusive banks.
(b) Simplest case of arbiter optimization.
(c) Example of original arbitration with overlapped banks.
(d) Arbiter simplification for overlapped banks.

PB: partitioned bank; THD: thread; AB: arbiter.

Figure 2: Examples of arbiter simplifications by the proposed work.

language Boogie [12], together with assertions. We then apply the Boogie tool-flow to automatically generate satisfiability modulo theory (SMT) queries, the results of which our tool interprets as directions to simplify arbitration logic in the original multi-threaded program. Our work is able to address the scalability challenge by extending the current LegUp tool flow with a fully automated static analysis flow that supports arbitrary input code.
The main contributions of this work are as follows:

1) A general technique that uses formal methods to prove memory bank exclusivity for arbitrary multi-threaded input code.

2) A technique for translating generic LLVM intermediate representation (IR) code for multi-threaded programs into the Microsoft Boogie (single threaded) verification language, suitable for proving the absence of memory bank contention.

3) 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.

4) Analysis and results showing that the proposed approach can achieve up to 87% area saving (geo. mean 48%) and 39% wall-clock time improvement (geo. mean 21%) over a number of benchmarks.

2 MOTIVATION

Fig. 1 gives an example of multi-threaded C code using pthreads. In each thread, a loop assigns elements of array `A` from data stored in array `B`. The element of `B` that is selected for each assignment uses the loop iterator, `i`, and the pure function `int f(int i)`.

Each thread will use the function to decide which portions of the shared memory they are going to access. If this function is sufficiently complicated, it may be non-trivial for a developer to know how the memory should be partitioned for parallel access.

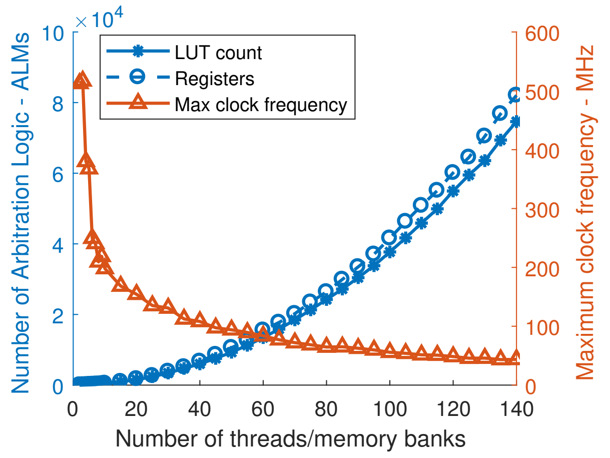


Figure 3: Evaluation on performance and chip area of original arbiters.

For example, the simplest case is `f(i) = i`. In this case, it is clear that the array indices accessed in each thread never overlap with the other threads, by combining knowledge of `f` with Line 16. For instance, thread 0 only ever accesses data at addresses 0 to 1023 of array `B`; while thread 1 only touches elements with addresses from 1024 to 2047. As these threads are both accessing mutually exclusive regions of `B`, a block partitioning strategy can be applied, where the array `B` of size 2048 can be divided into two sub-arrays of size 1024. Each thread accesses a unique partitioned memory bank during execution, so that the bank selection representing memory bank arbitration in hardware can be avoided.

In the architecture shown in Fig. 2(a), the LegUp HLS tool implements arbiters for each partitioned memory bank connecting to *all* threads. With many threads and banks, the size and delay of the arbitration logic becomes quite considerable. Fig. 3 shows how the arbitration logic hardware utilization and maximum clock

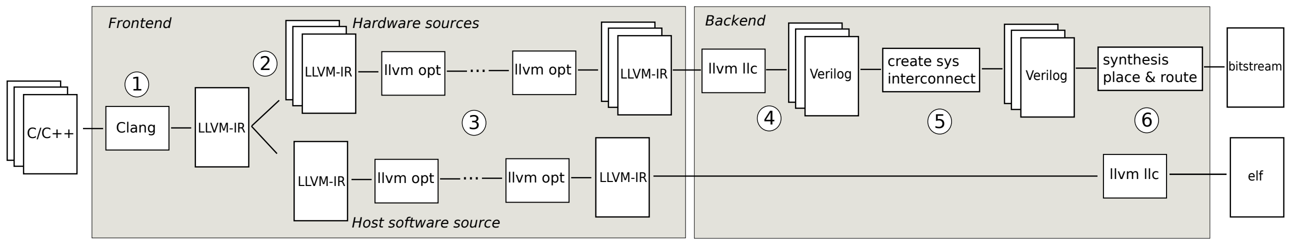


Figure 4: A coarse overview of the LegUp multi-threaded tool-flow.

frequency scales as the number of memory banks and threads are increased, where the number of threads is equal to the number of banks. As expected, increasing the number of threads causes a decrease in maximum clock frequency and increases hardware utilization. Observe that the maximum clock frequency of the arbiter drops sharply for a relatively small number of threads, eventually approaching a low sub-50MHz frequency. The figure also shows that both the logic utilization and number of registers used by the arbiters increases dramatically with the number of threads, leading to significant portion of total available resources. As the LegUp HLS tool conservatively assumes that any thread can access any bank, an arbiter port is needed for each thread. As the number of threads increases, so do the number of arbiter ports in the arbitration logic, resulting in a long propagation delay between memory banks and threads, decreasing the maximum achievable frequency.

The main objective of this work is to simplify the memory arbitration by statically proving (using the semantics of the input) that some threads are incapable of accessing some memory partitions, and optimizing the arbitration logic accordingly. With our approach, we are able to analyze the input source and through the use of formal methods, prove which thread never accesses a memory bank. Often, only the red wires shown in Fig. 2(a) are ever used during the whole execution, while the black wires are never used, demonstrating the source of conservatism and the potential benefits of arbiter simplification. After applying our approach, we achieve the connectivity in Fig 2(b), where all arbitration logic can be safely removed. A more general case is shown in Fig. 2(c), where a memory bank may be accessed by more than one thread. In this case the arbiters cannot be completely stripped out, but can still be simplified resulting in a more area-efficient arbitration architecture in Fig. 2(d).

In the context of the existing simulation-based approach [22], let T be the set of threads, and S be the subset of those observed to be accessing a given bank in simulation. Let F be the set of threads that have been formally proven by our tool to not be accessing a given block of memory. The original arbitration method for partitioned memory using S observed by [22] is to build an arbiter with $|T|$ connections conservatively, while our work builds an arbiter with $|T-F|$ connections. We know that every simulation-observed access pattern is possible, so $S \subseteq T - F \subseteq T$. Most of the time, we find $S = T - F$, which can be interpreted as 'it is safe to consider only those banks touched in simulation' (as in Fig. 2(c)). The 'best case' is $|S| = |T - F| = 1$ such as Fig. 2(a). The objective of our work is to find F for each bank.

3 BACKGROUND

3.1 The LegUp HLS tool

For this work, we have chosen to use the LegUp HLS tool, as it is the only tool that supports multi-threaded inputs, in the form of C style `pthreads`, with all other HLS tools only supporting single-threaded inputs [10, 19]. 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 optimize the generated memory interface for concurrently executing hardware threads, which is realized 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 [4] compiler framework. LLVM consists of a frontend, optimization stage, and backend. The frontend converts the input program into LLVM-IR, which can then be optimized using a collection of preexisting optimization passes, and the backend receives the final optimized 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 now summarize the key stages in LegUp's multi-threaded synthesis flow.

1) The input C is transformed into LLVM-IR via Clang [6].

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 optimization passes: some LegUp specific, such as bitwidth minimization, 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 [15].

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 synthesizing the Verilog using FPGA vendor tools.

Fig. 4 shows a labeled tool-flow diagram 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. In this work, the LLVM-IR is analyzed directly after the frontend, and the output of our analysis is used to optimize the RTL code generation in stages 4 and 5.

In LegUp, each hardware thread is synthesized into a hardware circuit with an FSM and datapath. The hardware circuits corresponding to threads operate independently. That is, there is no global schedule requiring data-synchronization on memories between threads (which can instead be achieved by using locks and synchronization barriers common in other HLS tools [18]).

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. LegUp also provides an optimization pass to partition shared arrays automatically into multiple smaller BRAM-based memories, enabling more data to be accessed concurrently [22] (c.f. Section 3.2).

As mentioned above, LegUp is unable to determine conclusively which thread will access which of the shared memories, forcing it to take a conservative approach, where it assumes any thread can access any shared memory. This assumption requires the construction of expensive crossbar-based interconnects and arbitration logic between all threads and every shared memory as illustrated in Fig. 2(a). 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 optimization that can split a shared memory into multiple smaller memories [22]. Memory partitioning allows for 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 (i.e. stalled cycles) can now occur. 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.

Memory partitioning cannot be performed blindly and requires care. If partitioning is well balanced, with threads accessing disjoint regions of the address space concurrently, then performance is improved. However, if some of the partitions are very "hot" with frequent accesses and others are very "cold" with infrequent accesses, then the overheads of the partitioning logic may outweigh the benefit seen by the increased throughput. To carefully 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 smallest memory contention frequency.

Table 1: Program analysis approaches for memory partitioning of HLS applications.

		Approaches for memory partitioning	
		Polyhedral analysis	Simulations + Formal methods
Input code	Single-threaded	[20], [2], [9]	[21]
	Multi-threaded		[22]+our work

While this approach greatly improves the throughput for multi-threaded shared memory applications, it cannot guarantee that multiple threads never access the same partition. This exacerbates the scalability challenge discussed in Section 3.1 and highlighted in Fig. 3, 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 partitions per shared memory. Our analysis alleviates this scalability challenge, as it is able to prove which threads can access which partitions of a shared memory through a static analysis of the code. This enables us to safely optimize the arbitration logic connecting memory partitions to threads, reducing its complexity and increasing performance.

3.3 Program Analysis Tools

Program analysis has been an active research area for optimizing circuit generation with HLS tools. Table 1 shows a comparison between our work and the following related works. Polyhedral techniques have been used in HLS since the work of Liu *et al.* [11]. Recent work such as Wang *et al.* [20] on polyhedral analysis for memory partitioning has shown promising performance through performing cyclic-like partitioning, exploring the translation 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 [2]. Winterstein *et al.* [7] 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 arbitrary code but currently only work with on-chip memory. The most recent work by Escobedo and Lin [9] propose a graph-based approach based on formal methods to compute minimally required memory banks for any given pattern to avoid memory contention for stencil-based computing kernels. In summary, these works are not built based on multi-threaded input, but are rather directed towards partitioning of arrays with multiple accesses in a loop body. These techniques are not applicable in multi-threaded cases, and often require code to have a particular structure (e.g. polyhedral loop nests).

Satisfiability Modulo Theories (SMT) solver-based approaches are relatively new to the HLS community, but have shown potential strengths in hardware optimization. The closest piece of work to this paper is by Zhou *et al.* [21], which proposes an SMT-based checker for verification of memory access conflicts based on parallel execution with banked memory, resulting in a highly area-efficient memory architecture. We both use simulation traces as a starting point for formal analysis, an approach referred to as 'mining' in [21]. However, they optimize a banking function with a model of area, while our work proves the absence of conflict for a banking function derived by [22]. Moreover, [21] does not take the whole program, with its control structures into account and instead formulates an SMT query on the banking functions themselves. Hence, it does not support input-dependent memory traces, which can be analyzed by our work.

Finally, [21] proves the existence of bank conflicts, allowing each parallel hardware unit to access different memory banks concurrently. For instance, they allow that at clock cycle 0, instance 0 accesses bank 0 and instance 1 accesses bank 1, while at clock cycle 1, parallel instance 0 accesses bank 1 and parallel instance 1 accesses bank 0. Conversely, we identify the memory banks *never* accessed by certain threads over the whole execution, because we target arbiter removal. For example, we can improve the arbitration logic in an instance where thread 0 never accesses bank 0 and bank 1 at any point, while thread 1 never accesses bank 2 and bank 3.

3.4 Microsoft Boogie

Boogie is an automatic program verifier from Microsoft Research, built on top of SMT solvers [12]. Boogie uses its own intermediate verification language (IVL) to represent the behavior of the program being verified. Instead of executing the verification code, an SMT solver is applied to reason about program behavior, including the values that variables may take. Encoding of verifications as SMT queries is automatically performed by Boogie 'behind the scenes', hidden from the user. Other works have proposed the automated translation of an original program to Boogie IVL, such as Smack [13], a automated translator of LLVM-IR code into equivalent Boogie code.

In addition to all the commands one would expect in a standard programming language, Boogie contains a number of verification-specific language constructs, which we use in our work, as detailed 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* values of the variable, unlike simulation-based testing which will only check assertions for particular test vectors.

2) `assume c`: The `assume` command adds 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.

4 METHODOLOGY

We convert the multi-threaded HLS input code into single-threaded Boogie code to verify the number of arbiter ports needed for the each partitioned bank. The Boogie code represents only those expressions from the original code that could impact on the memory bank partition index accessed by each thread. Formal techniques are applied to prove that certain arbiter ports are unnecessary. The subsections below highlight the main steps in the process, including how we translate the multi-threaded C into a Boogie program with a proper specification of program states, the construction of assertions that check whether arbitration is needed, and how we handle loops.

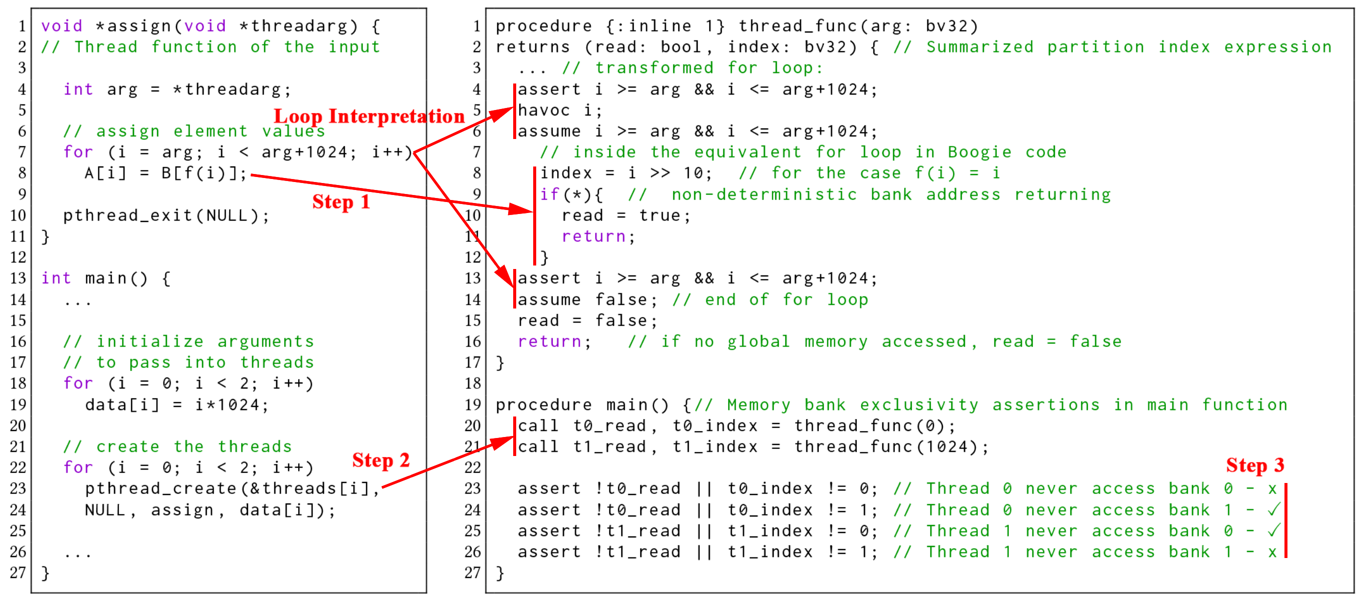
4.1 Multi-Threaded C and Boogie Program

Using the Boogie primitive operations outlined in Section 3.4, we can generate a single-threaded 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 transformed into a single-threaded Boogie program as part of the compiler pass. This source translation consists of three main steps:

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 code with the same inputs.
Step 3 : Within each Boogie thread instance we generate Boogie `assert` statements that are used to test for memory bank exclusivity.

First, a procedure named `thread_func` represents the memory behavior of the original thread function (named `assign` in the example code). From the example, it is intuitive that one thread accesses the array index range from `data[i]` to `data[i]+1023` in a `for` loop when `f(i) = i`. In the equivalent loop in `thread_func`, the partition index - determined using [22] - is required for verification instead of the exact data value or array index. For simplicity of demonstration, we assume here that the partition index is bit 10 of the 11-bit array index under the block partitioning scheme, but the expression can be arbitrary in general. A non-deterministic choice `if(*)` is used to model the fact that we need to capture the partition index accessed by *any* loop iteration; taking the if branch causes the verifier to flag that a read has happened with index `index` - if the branch is not taken, this corresponds to skipping this particular memory access in the original code. For this example, thread 0 can only return 0 as the set of all possible partition index is {0} with accessed array index ranging from 0 to 1023. Similarly thread 1 only returns 1.

The arbitrary partition index returning in Boogie is transformed from the extracted multi-threaded memory behavior, which consists of a number of *sliced* partitioned memory accesses. A sliced partitioned memory access is defined as a list of LLVM-IR instructions relating to the partition index, disregarding all other irrelevant instructions in the thread function [16]. We must translate these instructions for analysis of the partition index of accessed memory bank. When the data in an array is accessed such as `A[i]` in Fig. 1, the corresponding instruction in LLVM-IR code is represented as "`index = partition_index; if (*) {read = true; return;}`" in Boogie.



```
void *assign(void *threadarg) {
// Thread function of the input

  int arg = *threadarg;

  // assign element values
  for (i = arg; i < arg+1024; i++)
    A[i] = B[f(i)];

  pthread_exit(NULL);
}

int main() {
  ...

  // initialize arguments
  // to pass into threads
  for (i = 0; i < 2; i++)
    data[i] = i*1024;

  // create the threads
  for (i = 0; i < 2; i++)
    pthread_create(&threads[i],
    NULL, assign, data[i]);

  ...
}
```

(a) Input multi-threaded C code.

```
procedure {:inline 1} thread_func(arg: bv32)
returns (read: bool, index: bv32) { // Summarized partition index expression
  ... // transformed for loop:
  assert i >= arg && i <= arg+1024;
  havoc i;
  assume i >= arg && i <= arg+1024;
    // inside the equivalent for loop in Boogie code
    index = i >> 10;  // for the case f(i) = i
    if(*){  //  non-deterministic bank address returning
      read = true;
      return;
    }
  assert i >= arg && i <= arg+1024;
  assume false; // end of for loop
  read = false;
  return;   // if no global memory accessed, read = false
}

procedure main() {// Memory bank exclusivity assertions in main function
  call t0_read, t0_index = thread_func(0);
  call t1_read, t1_index = thread_func(1024);

  assert !t0_read || t0_index != 0; // Thread 0 never access bank 0 - x
  assert !t0_read || t0_index != 1; // Thread 0 never access bank 1 - ✓
  assert !t1_read || t1_index != 0; // Thread 1 never access bank 0 - ✓
  assert !t1_read || t1_index != 1; // Thread 1 never access bank 1 - x
}
```

(b) Output Boogie program.

Figure 5: Example of a 4-partition 4-thread case for bank exclusivity verifications.

Two further parts of the Boogie model appear in the `main` procedure. Instead of executing threads concurrently, the generated Boogie program calls each thread procedure in a sequential manner. These call instructions are considered as separate and independent modules for memory access analysis. Each call instruction returns a read state and an arbitrary partition index among accessed banks. Thus, the called procedure has either accessed the partitioned memory with a valid partition index, or does not access any partitioned memory, i.e. all none of the `if(*)` blocks have been executed.

The last part of the verification code is the list of final assertions. The final assertions are automatically generated by enumerating connections between each partitioned banks and each threads. Each single assertion states that a specific thread does not 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 arbiter port in hardware. With Boogie code containing a list of final assertions, the Boogie verifier is automatically called to filter all failed assertions. Based on the successful assertions left in Boogie code, it formally proves that arbitration is not required between certain combinations of memory banks and program threads, stripping out or radically simplifying the arbiters as a result. This approach supports any memory access pattern, and the verification results can correctly identify where arbitration is required in the HLS-generated hardware.

4.2 Loop Interpretation

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 code, in order to be able to support general `while` loops and also to keep the size of the verification code small. A loop in Boogie code typically 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 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 [5].

```
while (c)
invariant ϕ;
{
  B;
}
```

(a)

```
assert ϕ;   //(base case)
havoc modset (B);
assume ϕ;
//inductive hypothesis
if (c) {
  B;
  assert ϕ;  // (step case)
  assume false;
}
```

(b)

Figure 6: Loop summary in verification language using loop-cutting transformation [5].

Fig. 6 shows the general case of our loop transformation process: in Fig. 6(a) a general structure of a `while` loop is described, while Fig. 6(b) shows the equivalent transformed loop in Boogie. In Fig. 6(a), a `while` loop contains a conditional check on variable `c` and a loop body `B`. Additionally, ϕ represents the loop invariant. In Fig. 6(b), the invariants for the loop are established inductively. At the entry point of loop, also known as the base case, the first assertion asks Boogie to verify that the loop invariant holds. The next few lines skip through an arbitrary number of loop iterations,

Table 2: Reference table for invariants of *for* loops of the form `for( i=start; cond; st )`

.

		st	
		`i++`	`i--`
cond	`i < end` or `i > end`	`start <= i` `i < end`	`start >= i` `i > end`
	`i <= end` or `i >= end`	`start <= i` `i <= end`	`start >= i` `i >= end`

havocing the variables that might be changed by the loop body (`modset(B)`), and only `assuming` the induction hypothesis of the loop invariant in order to prove that the invariant still holds. We note that using this transformation, we can *guess* any loop invariant ϕ without being concerned about the correctness of the resulting memory structure, because we place an obligation on Boogie to verify that a guessed invariant actually does hold.

The selection of an appropriate invariant ϕ is key to verification success. However, for the HLS benchmarks we have considered, the loops have a simple structure of increasing or decreasing single-strided for loops with strict or non-strict inequalities as loop exit conditions. We therefore implement a simple table lookup of proposed loop invariant, following Table 2. For instance, if the loop index is incrementing with an exit condition of the index being equal to the end bound, the loop invariant would be that the loop index is greater or equal to start index value, and also less than the exit bound value. For the case of an inequality check as the exit condition, the loop index is less or equal to end-bound value. Similar results can be found with inversed signs in the cases where the loop index shows decrementing behaviors.

On the other hand, for loops with dependencies, more specific invariants are required to precisely describe the loop behavior to achieve precise results. For instance, the loop bounds can be dynamic, or the code following the loop could make use of knowledge of program variables at loop exit not captured in by the loop exit condition alone. We solve this by loop peeling typically for `for` loops, which is fully automated in our tool-flow; inferring invariants for general `while` loops still needs human guidance. Fig. 7 shows an example of code transformation applied by loop peeling typically to perform a `for` loop with exit condition of `i < N`. Different from Fig. 6(b), this code performs two iterations for one loop. The iterations except final iteration are represented using identical arbitrary *teleporting* method, which also results in invalidation of loop body summary. After that, loop iterator is set to `N-1` representing end of `(N-1)`th iteration following by the final iteration of the loop. The final iteration is presented as the second iteration which simulates the loop behavior of last iteration maintaining the loop body information in the last iteration valid for further assertions after the loop. Therefore, this can solve the case when the information in the loop body is required for further verification. However, it is only compatible for `for` loops at current stage. Since the number of the iterations in more general loop is unknown, the generation of modified set configurations of loops before final iteration is challenging and to be solved.

```
for(i=0;i<N;i++)
{

   b = i*2;

}

assert b==2*(N-1);
```

(a) Original C source.

```
i = 0;
assert i <= N-1 && i >= 0;
havoc i, b;
assume i <= N-1 && i >= 0;
if (i<N-1) {
  b = i*2;
  i++;
  assert i<=N-1 && i>=0;
  assume false;
}
i = N-1;
if(i < N) {
   b = i*2;
   i++;
}
assert b==2*(N-1);
```

(b) Loop peeling to maintain information of b.

Figure 7: Example of loop peeling in Boogie.

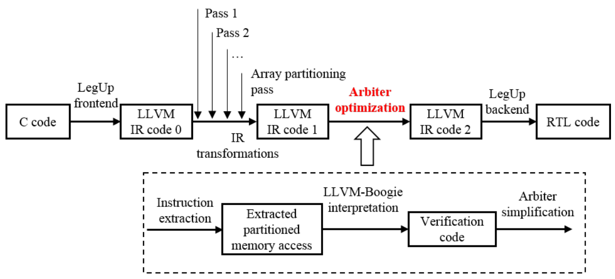


Figure 8: LegUp tool flow with proposed work integrated.

5 INTEGRATION INTO LLVM FRAMEWORK

The automated arbitration optimization process is implemented as a series of LLVM passes. Fig. 8 shows the LegUp HLS design flow with the work of arbiter optimization integrated. One of the LegUp frontend passes performs memory partitioning [22]. This pass divides selected arrays into a number of sub-arrays, where each array is also assigned with a unique partition index. Our work is carried out after the execution of this pass. Our work can be divided into three parts. Firstly, the multi-threaded memory behavior is extracted and formulated into a mathematical expression for bank mapping. Secondly, code transformation is carried out resulting in Boogie code for memory conflict verification. Finally, we remove the unused memory ports of arbiters in all threads based on the verification results given by the Boogie code. This results in a newly optimized LLVM-IR code with efficient memory arbitration.

5.1 Partitioned Memory Access Extraction

The slicing tool [16] allows us to extract only those pieces of the code that could affect the memory banks accessed by each thread. It radically simplifies the code, so that only those operations affecting memory access patterns are retained. For example, in the motivational example of Fig. 1, all instructions that may modify the value of `f(i)` are retained, while other instructions in the thread function are removed. This avoids wasted runtime in analysis of instructions that are not relevant to memory access behavior.

5.2 Compilation into Boogie

In this step, the sliced LLVM-IR code is mapped into a Boogie program. The translation of individual instructions from LLVM-IR code to Boogie code is straightforward. Since the original integer type in C is 32-bit, the equivalent data type in Boogie is a 32-bit bitvector (Boogie integers are unbounded). First, we iterate through the LLVM-IR instructions to extract variable names, which are declared at the beginning of the Boogie function. Then, the variables assigned arbitrary values are configured with `havoc` commands after the variable declarations. Then, LLVM-IR instructions can be directly replaced by available operators in Boogie or translated into a small number of equivalent instructions. The non-deterministic choice `if(*)` returning the partition index is inserted at the location where the pointer to the element in partition array is obtained. In other words, instead of accessing the requested data, the Boogie code returns the partition index of the requested bank. During interpretation, the assertions and assumptions for the transformed loops are inserted in Boogie code locations that correspond to the beginning of the entry block and the exit block of the loops in LLVM-IR code. The summarization of invariants for complex loops is achieved by recognizing `phi` instructions in the LLVM-IR code that are used for data reuse, where the necessary modified set for the current loop can be sliced and transformed.

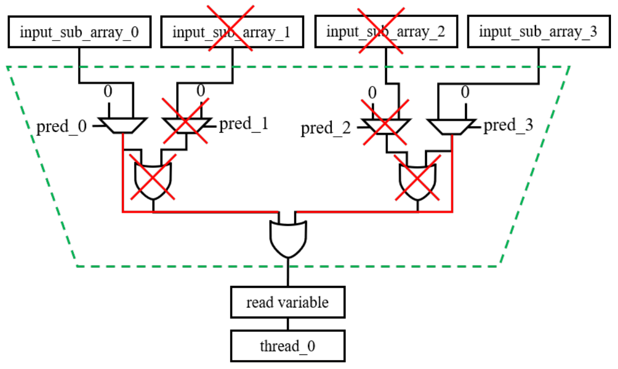
In the `main` procedure, after listing the separate function calls in the form of procedures in Boogie, the final assertions are constructed based on the partition indices and thread indices. If N is the number of threads, and M is the number of memory partitions, the Boogie program has $N \times M$ assertions corresponding to the individual interconnections between the threads and banks. Since the thread function calls are inlined in the `main` procedure, recursive function calls are not possible, however, recursion is also generally not supported by HLS tools.

5.3 Arbiter Optimization Process

Based on which Boogie assertions hold, memory arbiters are simplified without affecting execution correctness. This step is carried out in the LLVM-IR code level frontend. In each thread, the bank multiplexer used to access the partitioned memory is simplified by removing ports to banks that are never accessed. An example is shown in Fig. 9, for the case of thread 0 being proven to not access sub-arrays 1 and 2. The red '×'s show hardware that is safely removed in the final circuit.

6 EXPERIMENTAL RESULTS

The FPGA family we used for results measurements is Cyclone V (5CSEMA5F31C6) in Quartus II 15.0.0. The reason is that this FPGA



ik - local address of array *subX_ga*. bk - partition index.

Figure 9: LLVM-IR modification for efficient bank selection.

is one of the devices supported by 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}, cycle latency, and wall-clock time (cycle latency $\times\ 1/F_{max}$). We also discuss its impact on CAD tool run-time.

6.1 Benchmark Descriptions

We apply our approach to the eight multi-threaded benchmarks from [22]: *matrixadd*, *histogram*, *matrixmult*, *matrixmult (cyclic)*, *matrixtrans*, *matrixtrans (block cyclic)*, *substring* and *los*. In *matrixadd*, two integer matrices of size 128×128 are summed by blocking matrix operations 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. Matrix multiplication is implemented with two matrices of size 32×32 in *matrixmult*. Similarly, the element operations are divided into groups of row summations for parallelism. In *matrixmult (cyclic)*, the matrix row allocation has been rearranged in a cyclic scheme, grouping rows with matching LSBs to be operated on by a single thread. *Matrixtrans* computes the transpose of an input matrix of size 128×128 following the cyclic scheme. In *matrixtrans (block cyclic)*, the row allocation to different threads is based on both MSBs and LSBs of the index in a block-cyclic partitioning scheme, where a thread transposes rows at addresses of 0-3, 32-35, 64-67 and 96-99, for instance.

Benchmark *substring* searches for a string pattern of size 3 within an input string of size 16384, 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 expressions for partitioned bank access within a single thread. The line of sight example in *los* analyzes the presence of an obstacle between the elements in a predefined obstacle map of size 64×64 and center 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 center

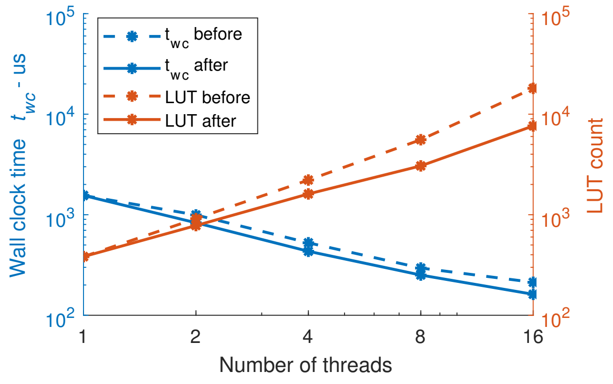


Figure 10: Evaluation of absolute parameters on *histogram* hardware with equal numbers of threads and banks.

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.

6.2 Case Study: Histogram

Overall computational performance is maximized 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. 10 shows the wall-clock time and LUT utilization of the optimized 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 linear 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 utilization increases quadratically, which appears to have doubled increasing rate compared to the wall-clock time. 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 optimized, 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 arbiter ports may be removed.

6.3 Results for All Benchmarks

The post P&R results for all benchmarks are given Table 3 for the case of 16 threads and memory banks[1]. 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 behavior. Significant improvements in benchmarks such as *matrixmult* 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 *matrixtrans* benchmark, the *cyclic* partitioning scheme has been applied by default, which has significantly benefited from the proposed work reaching improved clock period by 50.5% and logic by 87.4%. However, when applying $block - cyclic$ scheme, it appears to have the worst improvements. This attributed to the fact that each bank is touched by all threads during execution, the arbiters cannot be simplified. Hence appropriate memory partitioning is required to perform the most efficient arbitration solution. Across all benchmarks, LUT count was reduced by up to 87%, and wall-clock time was improved by up to 39%. Greater improvements are expected for devices with more threads. On average, wall-clock time is improved by 21%, and LUT count is reduced by 58%.

6.4 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. In addition, the increase in the number of threads also leads to more assertions, as well as duplicated thread procedure calls. The average runtime for all the benchmarks was 13 seconds. This is directly related to the number of constructed assertions, which in turn is related to two issues: the complexity of partition memory accesses and the number of threads. 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 verifications times are insignificant compared to Synthesis/P&R 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 proposals for memory banking, using them as formal specifications for memory exclusivity which, if verified, guarantee that arbiters can be removed or simplified without impacting on program correctness. Across a range of benchmarks, the execution time of the circuits has been improved by up to 39% (avg. 21%) combined with an area saving of up to 87% (avg. 58%). The performances of hardware with more memory architecture are also promising from our measurements.

The novelty of this work is in the automated procedure for optimization of the arbitration circuits. We have shown that the behavior of typical concurrent multi-threaded code can be over-approximated using non-deterministic choice in sequential Boogie code, 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.

[1]Full dataset DOI: 10.5281/zenodo.1523170.

Table 3: Arbitration simplification evaluation for 16 memory partitions and 16 threads.

benchmark	LUT count			Register count			Max clock frequency (MHz)			Total clock cycles			Wall clock time (μs)		
	before	after	%	before	after	%	before	after	%	before	after	%	before	after	%
histogram	18081	7639	58%	22331	13552	39%	71.79	94.54	32%	15269	14660	4%	212.7	155.1	27%
matrixadd	14286	2261	84%	14594	4176	71%	73.73	107.01	45%	2125	2122	0%	28.8	19.8	31%
matrixmult	21005	3641	83%	15872	5184	67%	74.02	85.88	16%	2130200	2130199	0%	28778.7	24804.4	14%
matrixmult(cyclic)	20880	11227	46%	15872	10479	34%	73.84	83.08	13%	2130200	2130199	0%	28848.9	25640.3	11%
matrixtrans	18653	2358	87%	14352	3548	75%	61.31	92.30	51%	37194	36167	3%	606.7	391.8	35%
matrixtrans(blockcyclic)	10606	9100	14%	8996	7402	18%	75.48	75.55	0%	62900	61715	2%	833.3	816.9	2%
substring	11029	2558	77%	11959	4718	61%	77.39	125.58	62%	443	439	1%	5.7	3.5	39%
los	23785	20610	13%	31283	25122	20%	73.81	81.46	10%	46514	45384	2%	630.2	557.1	12%
geom. mean	-	-	**58%**	-	-	**48%**	-	-	**29%**	-	-	**2%**	-	-	**21%**

One of the key advantages we have retained over more structured approaches, such as polyhedral methods, is the ability to deal with arbitrary code. That being said, although our tool 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 multi-threaded behavior induced, or execution time. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

8 ACKNOWLEDGEMENTS

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

REFERENCES

[1] A. Canis et al. 2013. LegUp: An Open-source High-Level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *TECS* 13, 2 (2013).

[2] A. Cilardo and L. Gallo. 2015. Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning. *TACO* 11, 4 (2015).

[3] Amazon EC2 F1 instances. 2018. (2018). https://aws.amazon.com/

[4] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE, San Jose, CA.

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

[6] Clang. 2018. (2018). https://clang.llvm.org/

[7] F. Winterstein, K. Fleming, H.J. Yang, S. Bayliss and G. Constantinides. 2015. MATCHUP: Memory Abstractions for Heap Manipulating Programs. In *FPGA*. ACM, Monterey, CA.

[8] Intel HLS Compiler. 2017. (2017). https://www.altera.com/

[9] J. Escobedo and M. Lin. 2018. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. In *FPGA*. ACM, Monterey, CA.

[10] J. Villarreal, A. Park, W. Najjar and R. Halstead. 2010. Designing modular hardware accelerators in C with ROCCC 2.0. In *FPGA*. IEEE, Charlotte, NC.

[11] Q. Liu, G.A. Constantinides, K. Masselos, and P.Y.K. Cheung. 2007. Automatic On-chip Memory Minimization for Data Reuse. In *FCCM*.

[12] M. Barnett et al. 2005. Boogie: a modular reusable verifier for object-oriented programs. In *FMCO*. ACM, Amsterdam, The Netherlands.

[13] M. Carter, S. He, J. Whitaker, Z. Rakamarić and M. Emmi. 2016. SMACK Software Verification Toolchain. In *ICSE-C*. ACM, Austin, Texas.

[14] Microsoft Project Catapult. 2018. (2018). https://www.microsoft.com/

[15] P. Coussy, M. Meredith, D.D. Gajski and A. Takach. 2009. An Introduction to High-Level Synthesis. *DTC* 26, 4 (2009).

[16] S. T. Fleming and D. B. Thomas. 2017. Using Runahead Execution to Hide Memory Latency in High Level Synthesis. In *FCCM*. IEEE, Napa, CA, USA.

[17] V.G. Castellana, A. Tumeo and F. Ferrandi. 2014. High-level Synthesis of Memory Bound and Irregular Parallel Applications with Bambu. In *HCS*. IEEE, Cupertino, CA, USA.

[18] Xilinx. 2016. SDAccel Development Environment - User Guide (v206.2). (2016).

[19] Xilinx Vivado HLS. 2017. (2017). https://www.xilinx.com/

[20] Y. Wang, P. Li and J. Cong. 2014. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *FPGA*. ACM, Monterey, CA.

[21] Y. Zhou, K.M. Al-Hawaj and Z. Zhang. 2017. A New Approach to Automatic Memory Banking using Trace-Based Address Mining. In *FPGA*. IEEE, Monterey, CA.

[22] Y.T. Chen and J.H. Anderson. 2017. Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software. In *FPL*. IEEE, Ghent, Belgium.