Resource and Memory Management Techniques for the High-Level Synthesis of Software Threads into Parallel FPGA Hardware

Jongsok Choi, Stephen Brown, and Jason Anderson
ECE Department, University of Toronto, Toronto, ON, Canada
legup@eecg.toronto.edu

***Abstract*—Recent work has proposed the high-level synthesis of parallel software programs (specified using Pthreads or OpenMP) into concurrently operating parallel hardware modules [6]. In this paper, we describe resource and memory management techniques for improving performance and area of hardware generated by such software thread synthesis. One direction investigated pertains to how modules in the HLS-generated parallel hardware should connect to one another: 1) with a *nested* topology, or 2) with a *flat* topology. In the nested topology, hardware modules are created in a hierarchical manner: modules are instantiated *inside* within modules that use them. Conversely, the flat topology instantiates all hardware modules at the *same* level of hierarchy. For the flat topology, we describe a system generator that automatically generates the required interconnect between all hardware modules, as well as flexibly shares *or* replicates functions, functional units, and memories. We also explore methods to reduce memory contention among hardware units that operate in parallel, by investigating three different memory architectures which use: 1) a *global* memory controller, 2) *local* memories, and 3) *shared-local* memories. Local and shared-local memories are dedicated RAM blocks for a single or a set of hardware modules, and help to increase memory bandwidth by allowing concurrent memory accesses. We also consider memory replication to localize memories in hardware modules, and convert small memories to registers to further improve performance and memory usage. Finally, we describe implementing locks and barriers in HLS hardware: synchronization constructs used in parallel programming. We show that with our resource and memory management techniques, we can improve the geomean performance, area, and area-delay product of parallel HLS-generated hardware up to 41.6%, 38.3%, and 63.3%, respectively, for a set of 15 benchmarks.**

I. Introduction

High-level synthesis (HLS) is an up-and-coming design methodology for FPGAs, which allows a user to automatically generate an RTL circuit description from a high-level software specification. The advantage of HLS is that a circuit designer can work more productively at a higher level of abstraction, reducing time-to-market vs. hand-coded RTL. With the input specification in software, HLS ultimately aims to make the performance and power advantages of FPGA hardware accessible to those with only software skills.

State-of-the-art FPGAs have high computational capacity afforded by abundant logic cells, memories, and hard blocks. For example, Xilinx recently announced the Virtex UltraScale XCVU440, a 20-nm device with 4.4 million logic cells [20], containing more than 20 billion transistors and making it the worlds densest IC. Leveraging the available FPGA resources is an important factor in meeting the performance requirements of a hardware system, and by using HLS, we can exploit an FPGA's spatial parallelism more easily than when manually designing in RTL. The LegUp high-level synthesis tool from the University of Toronto [5] offers a unique feature for this purpose: it is able to synthesize parallel software threads (specified using the Pthreads or OpenMP standards) into parallel-operating hardware modules [6]. Each software thread is synthesized into a concurrently operating hardware instance. Such an approach is attractive as it enables software engineers to leverage hardware parallelism through a programming paradigm they are already familiar with. However, the synthesis of software threads by HLS, as described in [6], has two limitations: 1) It synthesizes memory architectures that can result in high contention among threads, and 2) It is unable to allow threads to share resources (for example, to share a particular functional unit). It is precisely these limitations we address in this paper. Throughout this paper, we use *thread* to mean both the software thread, and the hardware module arising from the HLS of a software thread.

In the context of concurrently operating hardware threads, we investigate two crucial aspects which affect circuit performance and area: *circuit topology* and *memory architecture*. For circuit topology, we first examine the *nested* topology, wherein hardware modules are created and connected in a hierarchical manner, such that a module or a functional unit is instantiated *inside* the module where it is used. The nested architecture is intuitive and easy to implement, and is also used by Xilinx's Vivado HLS [19]. We compare this to a flat topology, where all modules are created at the *same level* of hierarchy. For the flat circuit topology, we describe a system generator, which automatically creates the needed interconnect between all components in the system. The system generator can also flexibly share or replicate functions, functional units, and memories. In the presence of parallel threads (parallel hardware), it automatically inserts arbitration and deadlock-prevention circuitry.

Together with the circuit topology, we also investigate a number of different memory management techniques, with the goal of reducing memory contention between parallel threads. We explore three different memory architectures using: 1) A shared *global* memory controller, 2) *local* memories, and 3) *shared-local* memories. A global memory controller allows pointer aliases to be resolved at circuit *run-time*, but has limited memory bandwidth. Points-to analysis can be used to determine at *compile time*, which array a pointer can reference. With this, we implement *local* and *shared-local* memories, which are directly connected to the accessing modules and permit concurrent accesses. With parallel threads, we also investigate using memory replication to localize memories to each thread, to further reduce memory contention. In parallel programming, synchronization constructs can be required to ensure proper execution of a program. We show how the points-to analysis can be applied to implement locks and barriers efficiently. Lastly, we describe memory-to-registers conversion, which helps to reduce memory usage and latency. We evaluate the proposed topology

978-1-4673-9091-0/15/$31.00

and memory architecture changes directly with LegUp HLS.

The contributions of this work are:

1) Analyzing the merits/disadvantages of the nested and the flat circuit topologies. We compare this to the hierarchical circuit topology used by Vivado HLS.
2) Presenting a system generator that can flexibly share/replicate functions, functional units, and memories between functions/threads, as well as automatically inserting hardware to prevent deadlocks. We contrast this with Qsys, the system integration tool from Altera.
3) Investigating the use of points-to analysis to create different memory configurations, in addition to handling thread-synchronization constructs.
4) Quantitatively evaluating the impact of circuit topologies, and together with resource management techniques, on the performance and area of parallel HLS hardware arising from software thread synthesis.

Our work represents a step towards improving the performance and area of parallel HLS hardware.

The remainder of this paper is organized as follows: Section II discusses related work. Section III provides an overview of the two different circuit topologies and Section IV presents the system generator. Section V discusses the different memory architectures with techniques to improve memory bandwidth. An experimental study is described in Section VI and conclusions are presented in Section VII.

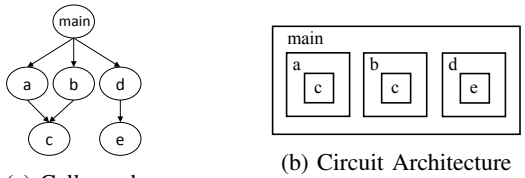
II. Related Work

A number of prior works have focused on the architecture of HLS-generated circuits. [28] and [14] investigated implementing efficient pipelined hardware for multi-threaded kernels. [11, 9, 6], and [27] implement parallel hardware architectures with Pthreads and OpenMP using HLS. Altera's OpenCL compiler [18] also creates deeply pipelined hardware from massively parallel OpenCL kernels. Vivado [19] provides knobs for pipelining both entire functions and loops. [8, 29, 4], and [23] investigate implementing efficient loop pipelining hardware. Such prior works pertain mainly to the micro-architecture of the data-path produced by HLS. Conversely, our work considers the system-level architecture, specifically, how functions, memories, and functional units can be connected together within a larger surrounding circuit and how they can be shared *or* replicated between parallel modules. In fact, the techniques proposed in this paper are compatible with prior work on synthesis of pipelined hardware modules.

In terms of resource sharing, [21] discusses sharing across *call hierarchies* using the flat topology, but not in the context of parallel-operating hardware. [7, 13] investigate sharing resources but within a module. In [22], resources are shared between loops for optimizing throughput. In contrast, our work investigates resource sharing between parallel threads. To our knowledge, no other work has analyzed the impact of circuit topology together with function, memory, functional unit sharing and replication on the area and speed of parallel HLS-generated hardware.

III. Circuit Topology

Unlike software compilers, which target fixed processor architectures, HLS offers the freedom to evaluate and choose the best architecture for a specific application. The circuit topology considered here is a dimension along which such a design's architecture may be optimized.



(a) Call graph (b) Circuit Architecture

Fig. 1: A call graph and its circuit architecture using nested topology.

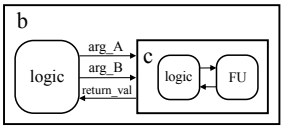


Fig. 2: Internal architectures of module `b` and `c`.

In this section, we describe two circuit topologies, the nested topology and the flat topology. In the nested topology, each hardware module is self-contained, meaning that, aside from data in memories, it does not rely on other hardware modules outside of its own module hierarchy. Fig. 1a shows a call graph of a program, and Fig. 1b shows the corresponding nested circuit architecture. As depicted, the architecture is hierarchical, with `main` being the top-level module, and its hardware modules recursively instantiated inside. Note that due to the hierarchical structure, there are *two* copies of module `c`. This is the default architecture used by both LegUp and Vivado HLS. The hierarchical approach thus precludes the sharing of modules by other modules, potentially leading to high area consumption.

Fig. 2 shows the internal architectures of modules `b` and `c`. Observe that module `c` has a functional unit (FU) instantiated within. In the nested topology, the arguments of a function become input ports of its hardware module, and the return value (if any) becomes an output port of the module.

The advantage of the nested topology lies in its simplicity: connectivity between modules is entirely local. Any modules used by a module are directly instantiated within the module itself and connected inside. The hardware module interface is aligned to that of software (i.e. arguments *passed in* become input ports, any data *returned* become output ports). Each hardware module is also self-contained, so if one needs to re-use a particular module in a new hardware system, simply instantiating that one module is sufficient.

However, the nested architecture is inefficient in a number of ways. In the input software, if a function is called by multiple different functions, the nested topology replicates hardware, as was shown in Fig. 1b. Likewise, since functional units are instantiated within hardware modules, sharing is precluded. Given that dividers or float-pointing units are generally large, this can considerably increase overall circuit area, particularly if such functional units are used in many different modules.

Fig. 3 shows the circuit architecture using the flat topology,

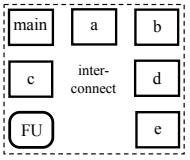


Fig. 3: Circuit in Fig. 1 with flat circuit topology.

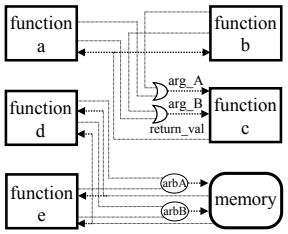
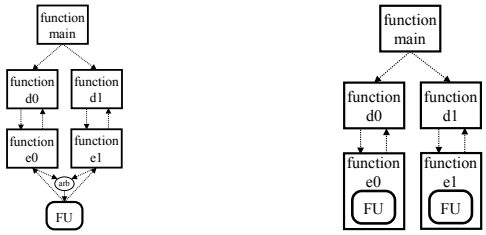


Fig. 4: An example interconnect generated by system generator.



(a) With functional unit sharing.

(b) Without functional unit sharing.

Fig. 5: Parallel hardware with/without functional unit sharing.

for the same circuit shown in Fig. 1. In the flat architecture, all modules reside at the same level of hierarchy, which enables sharing of functions and functional units. As shown, only one instance of module `c` is created, which is shared by modules `a` and `b`. The system generator, described in the next section, automatically creates the interconnect to directly connect or share functions, functional units, and memories in both sequential and parallel-execution modes.

IV. System Generator

For the flat circuit topology, we built a system generator to automatically connect all communicating hardware components in the system. The system generator handles both sequential and parallel execution, generating a different interconnect optimized for each case. It is similar to other system generators, such as Qsys (the system integration tool from Altera), except that it is completely integrated into the HLS framework. As such, it requires no additional input from the user; whereas, with Qsys, the user must specify which components connect to which other components, through which type of interface.

Our system generator automatically creates the interconnect by traversing the function call graph of the input program. All connections are point-to-point, allowing concurrent independent transfers. Each connection is composed of a pair of interfaces, a *master* interface and a *slave* interface. A master interface initiates a transfer, and a slave interface responds to the transfer. For instance, a function accesses a memory through its master interface (composed of address, enable/write enable, read/write data ports), and the memory responds through its corresponding slave interface. A single module can have multiple interfaces, allowing concurrent transfers (i.e. a function can have multiple interfaces to access multiple memories simultaneously, as well as interfaces to call other functions, and access functional units). When multiple master interfaces are connected to a single slave interface, with the master components executing sequentially, the system generator creates a simple `OR` gate to handle contention efficiently. The `OR` suffices in this case, as long as inactive masters output logic-`0` to their corresponding `OR` inputs. When multiple *parallel* masters are connected to a slave, a round-robin arbiter is automatically created. This differs from Altera's Qsys, which creates a round-robin arbiter regardless of whether the components are executing concurrently or sequentially, negatively impacting area and Fmax. An example interconnect generated by the system generator is shown in Fig. 4. In this case, function `a` and `b` execute sequentially and share function `c`, and functions `d` and `e` run in parallel and share a memory. Each memory is dual-ported, so we create a separate arbiter for each port to maximize memory bandwidth.

The system generator is also responsible for selectively sharing or replicating common hardware modules, based on a user's performance vs. area requirements. If a function is parallelized with threads, then the HLS tool creates as many hardware instances of the function as the number of threads in the input program. If the threaded function has descendant functions, then by default, the tool also replicates the descendant functions in hardware to maximize throughput. For example, Fig. 5 shows the circuit architecture where `main` forks two threads to execute function `d`, which has a descendant function `e`[1]. Sharing (Fig. 5a) vs. replication (Fig. 5b) of functional units or memories is controlled by a configuration parameter. In the case of replication, the component is instantiated *inside* the module which uses it (Fig. 5b), creating a dedicated component for that module. Replication of memories is further discussed in Section V.

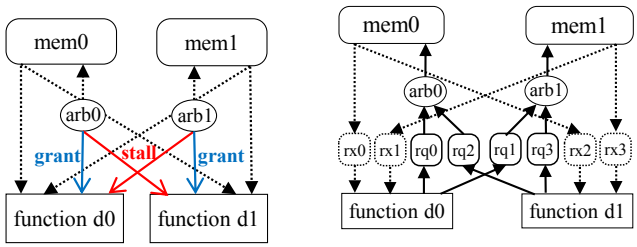
A. Automatic Deadlock Prevention

As shown in Fig. 4, arbiters are generated to allow concurrent access by multiple masters to a shared resource. However, when multiple masters request to access multiple common slaves at the same time, a deadlock can occur. This is illustrated in Fig. 6a, where functions `d0` and `d1` request to access *both* `mem0` and `mem1` at the same clock cycle (with each function have a dedicated memory port to each memory). In the example, `arb0` grants access to function `d0`, and `arb1` grants access to function `d1`. Both functions are not able to continue as they are both waiting to receive a grant for the "other" memory – a deadlock[2]. To prevent deadlocks, our system generator automatically inserts deadlock prevention modules where necessary.

There are two parts to the deadlock prevention module, the *request* module and the *data-receiver* module, denoted as `rq` and `rx` in Fig. 6b. This set of deadlock prevention module is created for each dedicated memory interface for each function (`rq/rx` 0 and 1 are for function `d0`, and `rq/rx` 2 and 3 are for function `d1` in Fig. 6b). The request module handles the request for a master interface to its connecting slave arbiter. It ensures that once a master interface has received a grant from its arbiter (which allows the master to access the slave in the same cycle), it does not make the same request again (requests are *state* dependent, thus being stalled in the same state would keep the request signal high continuously). Returning to the deadlock scenario described earlier, after the grants have been received (and the respective memories have been accessed), the request modules lower the requests from function `d0` to `arb0` and from function `d1` to `arb1`. Then, in the next cycle, only the requests from function `d0` to `arb1` and from function `d1` to `arb0` remain for arbitration, which are independent to each other, and thus a deadlock does not occur. The circuit for the request module is

[1] If the descendant function is only called *once*, it may be beneficial to *inline* the function, to allow additional compiler optimizations. This can be controlled by the user through a configuration parameter.

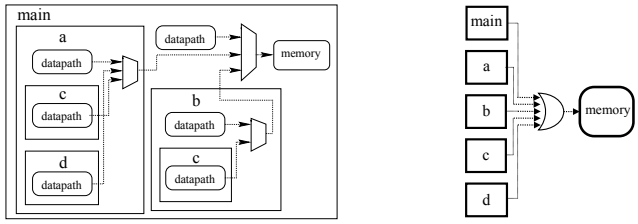
[2] A function's FSM remains stalled as long as its stall signal is asserted. Its stall signal is a logical `OR` of all stalls for its master interfaces from their arbiters.



(a) A deadlock situation.

(b) With deadlock prevention modules.

Fig. 6: Circuit architecture with/without deadlock prevention modules.



(a) Sharing a memory in nested topology.

(b) Sharing a memory in flat topology.

Fig. 7: Memory sharing in nested/flat topology.

simple, it contains a register which: 1) stores a `0` when the grant is given for a master interface, but the stall is still asserted for the function (due to stalls for its other master interfaces), and 2) otherwise stores a `1`. This register output is simply `AND`'ed with the request from the master interface, preventing it from keeping the request high once its grant is received. The request modules essentially ensure that after all of the requests have been granted once, the function is able to continue to execute, and it can work for any number of requests.

Note that if concurrent requests from the same function reach their corresponding memories at different clock cycles due to contention, the data from the memories also returns at different cycles. This results in incorrect execution, since the function expects both data items at the same time. The purpose of the data receiver module is to ensure that the data returned to the master is received correctly, by buffering the data, as appropriate. If there were no stalls, the data receiver passes through the returned data directly, otherwise it returns the buffered data. The circuit for the data receiver is also straightforward: It contains a shift register, with its size equal to the latency of the slave, and it shifts in `1` into the LSB when the grant is given from the arbiter. When the MSB of the shift register contains a `1`, it indicates that the slave is returning its data in that clock cycle. At this time, the data receiver stores the data in its registers, and also passes it through directly to the master (if the master was not stalled, the data is needed in that clock cycle), and at other times it returns the stored data. The data receiver is parametrized to allow connecting to slaves with any latency and data width. For instance, it can be connected to a divider, which has a latency equal to its bitwidth, as well as a multiplier or a memory, which have shorter latencies. It can also work for variable latency operations (i.e, off-chip memory access), by enabling the shift register only when the `valid` signal[3] from the variable latency operation is received.

In summary, our system generator creates an efficient interconnect completely automatically, benefiting from the integration within the HLS framework and access to the program's call graph in the compiler. It handles arbitration for sequential and parallel execution modes, allows flexible sharing or replication of functions and functional units, and inserts dead-lock prevention modules when necessary. We believe these are unique features of our work.

B. Advantages of Flat Topology with the System Generator

The flat circuit topology enables the efficient sharing of modules. In the nested topology, in order to share a memory between two modules, the accessing modules need connectivity to the module where the memory is instantiated. If an accessing module is deep in the function hierarchy, memory ports must be created and signals passed across all intermediate modules, as shown in Fig. 7a. In Vivado HLS, and when using the nested topology in LegUp HLS, multiplexers are created at each level of hierarchy (in the sequential case; i.e. when modules sharing the memory are not operating concurrently). The size and the depth of the multiplexers grows linearly with the number of functions that access the memory and the depth of the call hierarchy. Modules instantiated multiple times due to the nested topology also increase the multiplexer size unnecessarily, as shown in Fig. 7a for function `c`. This leads to poor performance and area. Functions can be inlined to remove some multiplexers, but this may increase circuit area. In the flat topology, all shared modules are instantiated at the same hierarchy level, and we *zero* out all memory signals when they are not being used, so that our system generator can simply connect them through an `OR` gate (in the sequential case), as shown in Fig. 7b. In the parallel case, `OR` gates are replaced with arbiters, as is done with functional unit sharing described earlier. In Vivado HLS, we were not able to share functional units across different functions[4]. As for Altera's OpenCL Compiler, it simply *inlines* all descendants functions of a kernel, eliminating the option to share functions or functional units.

V. Memory Architectures

Memory architecture can play a critical role in any hardware system and memory bandwidth is often the limiting factor for performance. A key architectural feature of FPGAs is the availability of on-chip block RAMs which provide low-latency memory accesses. Block RAMs are distributed throughout the chip, and can be accessed in parallel. There are also an abundant number of registers, which can also be used to store data. We therefore examine the different ways we can make use of the block RAMs and registers to reduce memory contention, in the presence of parallel operating hardware. We first consider three different memory architectures which use: 1) a *global* memory controller, 2) *local* memories, and lastly 3) *shared-local* memories. We use points-to analysis to designate arrays for implementation in global, local, and shared-local memories. Shared-local memories are shared by multiple modules in a system, hence require logic to handle contention, which may increase circuit area and latency. To mitigate this, we investigate replicating constant (read-only) shared-local memories across parallel modules to eliminate the overhead of arbitration logic.

[3]IP cores for variable latency operations, such as off-chip memory, have a `valid` signal to indicate that the data being returned in *valid* in that cycle.

[4]The Vivado HLS user manual shows that the `config_bind` configuration with the `min_op` option can be used to share functional units globally in a program. However, when we tried to share a divider in two separate functions, Vivado HLS created a divider inside each of the functions.

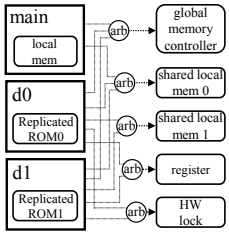


Fig. 8: Circuit using the different types of memories.

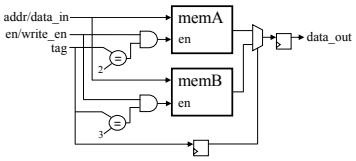


Fig. 9: Global memory controller architecture.

We also show how points-to analysis can be used to efficiently implement locks and barriers for thread synchronization. Lastly, we consider converting small memories to registers to lower memory usage and latency. An example circuit containing all of these features is shown in Fig. 8. In the figure, the functions `main`, `d0` and `d1` execute in parallel, and they share the global memory controller, shared-local memories 0 and 1, a register module, as well as a hardware lock module. A local memory that is used only by `main` is instantiated inside the function, and `d0` and `d1` have replicated constant memories instantiated inside. For simplicity, only one port of memory is shown and the deadlock prevention modules are also not shown. Each component is accessed through a set of dedicated ports, allowing concurrent accesses.

A. Points-to Analysis

To intelligently designate arrays for implementation in *global*, *local*, *shared-local* memories, we use a points-to analysis, which determines which memory locations a pointer can reference. There have been many points-to analysis algorithms developed by the compiler community. Andersen [2] described the most accurate of these approaches, which formulates the points-to analysis problem as a set of inclusion constraints for each program variable, which are then solved iteratively. Steensgaard [26] presented a less accurate points-to analysis, which used a set of type constraints modeling program memory locations that can be solved in linear-time. In this work, we use the points-to analysis described by Hardekopf [16] and implemented in the LLVM compiler by [12]. This algorithm speeds up Andersen's approach by detecting and removing cycles that can occur in the constraints graph. For each memory access in a program, the points-to analysis returns a set, which contains all the arrays the address can possibly point to. If it returns a set of size 1, it indicates that the address can only point to a single array, which will be located in one logical hardware RAM by the HLS tool (possibly split across several physical block RAMs by the vendor's RTL synthesis). Otherwise, the address points to multiple arrays, which needs to be resolved at run-time. Points-to analysis algorithms have varying levels of accuracy and may be overly conservative, but for programs without dynamic memory, recursion, and function pointers, *most* pointers are can be resolved at compile time [25].

B. Global Memory Controller

The purpose of the global memory controller is to automatically resolve pointer ambiguity at run-time. The global memory controller is only created if there are pointer references that cannot be resolved at compile-time with the points-to analysis (i.e. pointers pointing to multiple arrays). Its architecture is shown in Fig. 9. For clarity, some of the signals are combined together in the figure. Even though the figure depicts a single-ported memory, all memories are dual-ported by default. The memory controller steers memory accesses to the correct RAM, by using a tag, which is assigned to each array in the program by the HLS tool. A tag is set to be the top 9-bits (which can address up to 512 global memories; this bitwidth is easily configurable) of an incoming address, and it is used to determine which memory block to enable, with all other memory blocks disabled. The same tag is used to select the correct output data between all memory blocks. The lower bits of the address are used to get the offset into the RAM. Each block RAM has latency of one cycle, and the output of the multiplexer is also registered (to improve Fmax), making a memory access two cycles by default.

The advantage of this memory controller is that it can support generic pointers and resolves pointer references at run-time (by using the tags). This permits the support of a wider range of input programs, including those which may not be amenable to pointer analysis. However, there are a number of drawbacks to this memory architecture, in terms of its performance and area. First, any memories in the memory controller must be accessed sequentially. This is because the memory being accessed is determined at run-time, and hence needs to be accessed through a shared set of memory ports. This limits memory accesses to two per cycle (owing to the underlying dual-ported memories). In addition, the output multiplexer of the memory controller grows linearly in size with the number of RAMs. Increasing the number of global memories can hurt both the Fmax of the circuit as well as its area. The performance and area deterioration becomes worse when using the global memory controller with the nested circuit topology, as was shown in Fig. 7a, due to the large amount of multiplexing required to connect to the memory controller. Despite this, the memory controller ensures that the circuit can handle all types of pointer accesses, and may be needed for some programs.

C. Local and Shared-local Memories

Using the points-to analysis, we can designate arrays in the program to implement in *local* and *shared-local* memories. An array is designated into a local or a shared-local memory if the points-to analysis can determine that it is never referenced by a pointer that points to multiple arrays. If such an array is only accessed by a single function, it is designated as local memory. Otherwise, if it is referenced in multiple functions, it becomes a shared-local memory. Each local and shared-local memory is accessed through a dedicated set of memory ports, allowing concurrent memory accesses between memories. A local memory is instantiated *inside* the module which accesses it, hence connected directly, and a shared-local memory is instantiated *outside* the module, with an arbitration unit created to handle memory contention between its users. Because local and shared local memories have limited numbers of accessors, the memory latency is set to *one* clock cycle. We have empirically found that this improves the overall performance (the latency can also be easily configurable by the user). In Vivado HLS, memory access latencies are set to *two* cycles for all memories. Within local and shared-local memories, we perform a number of optimizations

to improve performance. As described below, we replicate read-only memories, and convert memories to registers. We can also implement synchronization constructs efficiently with points-to analysis.

1) Constant Array Replication: Constant arrays are implemented in read-only memories (ROMs), and as such, can safely be replicated in each accessing module. Although replication increases memory usage, for memory-intensive applications, where many threads contend for the same memories, it can be beneficial to create a dedicated memory for each thread. By localizing the memory to each thread, we can improve performance by reducing stalls due to contention, and also decrease area by removing the arbitration logic. Enabling this feature can be controlled through a configuration parameter in our work.

2) Memory to Register Conversion: By default, LegUp HLS implements all arrays in block RAMs. However, for small arrays with few elements, implementing them in registers may be beneficial to reduce memory usage. In LLVM, global variables are treated the same way as global arrays. Thus, we detect when an array has a single element, or if it is a global variable, and we store it in a register module. LLVM has an existing compiler pass called *mem2reg*, which promotes memory to registers [17], however we found this to work only in a limited number of cases, necessitating this optimization.

If the converted register is used by a single function, we create the register inside the function, or if it is used by multiple functions, we create a register module which is connected to all of user functions by our system generator, which in turn automatically creates the arbitration. Similar to how we had set the memory latency *1* clock cycle for local and shared-local memories, we can actually set the memory load latency for these registers to *0* clock cycle, further reducing memory latency. In addition, since the load latency is `0`, the register outputs are directly connected to the accessing modules, and do not need to connect through the data receivers, which also reduces area.

As described, we can flexibly adjust the memory latencies of the different types of memories to optimize performance, with global memories having `2` cycles, local/shared-local memories having `1` cycle, and memories converted to registers having `0` cycle.

3) Handling Synchronization: In parallel programming, synchronization of threads can be required to ensure proper execution of the program. In Pthreads, locks are used to ensure atomic access across threads, and barriers are used to synchronize all threads at a certain point in a program. Both constructs require the use of synchronization variables; a *mutex* variable and a *barrier* variable. With points-to analysis, we can treat the synchronization variables as memory variables, and classify them as a shared-local memory (since they are accessed by multiple modules). Points-to analysis returns a list of functions which use the synchronization variable. Then, we create dedicated ports from each function to each lock/barrier variable. When multiple locks are used in a program, they each have dedicated ports and can be accessed concurrently. Again, the system generator automatically creates arbiters and dead-lock prevention modules for each lock/barrier variable. Each lock and barrier variable is replaced with a hardware lock module and a hardware barrier module, respectively [6]. Communicating with a lock module is achieved through memory loads and stores. To obtain the lock, a thread polls on the lock module until it return a `1`, which indicates that the lock is unlocked. After it returns a `1` the lock module becomes locked (returns a `0`). To release a lock, the owner writes back to the hardware lock module, which unlocks it. The barrier hardware is implemented as a counter similarly through memory loads/stores. Previously in [6], without the points-to analysis, all locks and barriers needed to be accessed through shared memory ports (also shared with memories), and were created inside a central synchronization controller. Similar to the global memory controller, the synchronization controller steered accesses to the correct lock/barrier module at run-time but limited concurrent accesses. With points-to analysis, we can access multiple locks and barriers concurrently, independent of other memories.

VI. Experimental Study

In this section, we study the impact of the the different circuit and memory architectures on the performance and area of parallel hardware. We consider in total 8 different architectures:

1) Nested topology with a global memory controller.
2) Flat topology with a global memory controller.
3) Architecture 2 *plus* divider sharing across threads.
4) Architecture 3 *plus* multiplier sharing across threads.
5) Architecture 4 *plus* local, shared-local memories (all memories have latency of `2` cycles).
6) Architecture 5 *plus* memory to register conversion (latency of local/shared-local memories set to `1` cycle, register module has latency of `0` cycle).
7) Architecture 6 *plus* constant memory replication across threads.
8) Architecture 7 *minus* multiplier sharing across threads.

With each successive architecture, we enable/disable a feature, allowing us to analyze its impact in isolation. Architectures 1, 2, 3, and 4 have no local or shared-local memories. A global memory controller can be used in any of the 8 architectures, but is only created for benchmarks which require it. Comparing architectures 1 and 2 reflects the utility of the flat architecture vs. the nested architecture. With architectures 3 and 4, we can examine the impact of sharing functional units. Architecture 5 shows the impact of having dedicated memories with local and shared-local memories, and Architecture 6 illustrates the effect of reducing the memory access latencies. With, Architecture 7, we can investigate the effect of memory replication on memory contention, and lastly, with Architecture 8, we analyze the area/performance impact of disabling multiplier sharing across threads. For the rest of this paper, each architecture is referred to by its number (i.e. Arch. 1 = nested with global memory controller).

A. Benchmarks

We use a total of 15 benchmarks, each of which is parallelized with Pthreads. The benchmarks are described below. Each benchmark includes built-in inputs and golden outputs, with the computed result checked against the golden output at the end of the program to verify correctness.

- Alphablend: Alphabends two images.
- Barrier: An accumulation benchmark which uses a barrier.
- Blackscholes: Options pricing via a Monte Carlo approach.
- Box Filter: A convolution filter commonly used in image processing. C implementation of the filter adopted from [1].
- DF: Adopted from the CHStone [15], it performs double-precision floating-point operations using 64-bit integers.
- Division: Integer division of two arrays.
- Dot Product: Dot product of two arrays.
- Hash: Four different hashing algorithms, with the number of collisions compared at the output.

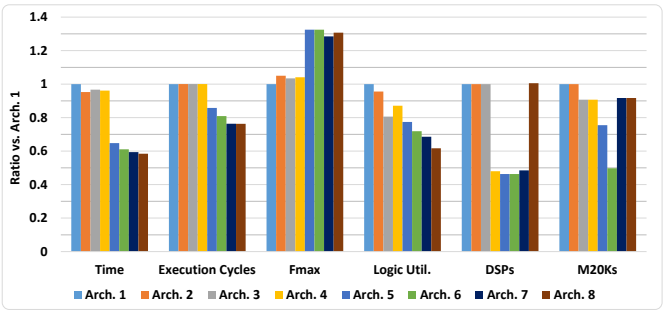


Fig. 10: Geomean performance and area results for each architecture.

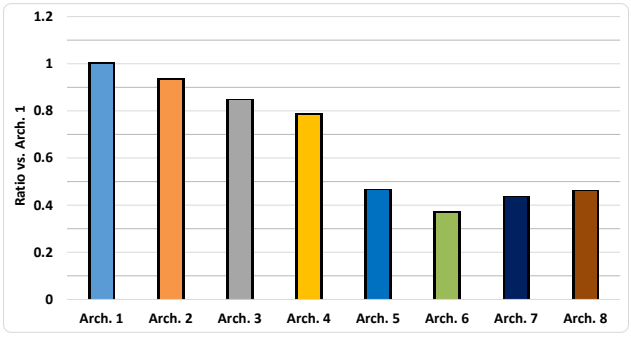


Fig. 11: Geomean area-delay product for each architecture.

- Histogram: Accumulates integers into 5 equally-sized bins.
- Line of Sight: uses the Bresenhams line algorithm [3] to determine whether each pixel is visible from the source.
- Mandelbrot: An iterative mathematical benchmark which generates a fractal image.
- Matrix Multiply: matrix multiplication of two arrays.
- MCML: Adopted from the Oregon Medical Laser Centre [24], it simulates light propagation from a point source in an infinite medium with isotropic scattering.
- Mutex: An accumulation benchmark which uses a lock.
- Vector Add: Performs vector addition of two arrays.

Other than the *Mutex* benchmark, *Barrier* and *MCML* benchmarks also use Pthread locks. Each benchmark was synthesized, placed and routed into the Altera Stratix V FPGA (5SGSMD8K1F40C2) with Quartus 15.0.

B. Results

Table I shows the geometric mean results across all benchmarks for Arch. 1, and Fig. 10 shows the geometric mean results for each architecture relative to Arch. 1. There are three performance metrics (total wall-clock time (# cycles $\times$ clock period), total number of clock cycles and Fmax of the circuit) and three area metrics (logic utilization, DSP blocks, and M20K blocks). The logic utilization metric reported by Altera Quartus II is an estimate of how full the device is, calculated from the number of ALMs (adaptive logic modules) used in the design. M20Ks are Altera's on-chip RAMs that can each hold up to 20 Kbits of data.

The general trend is that, as we progress from Arch. 1, towards the Arch. 8, results improve in terms of both performance and logic utilization. Comparing Arch. 1 and 2, both logic utilization and Fmax improve slightly, owing to the previously described efficiency of the flat topology vs. the nested topology. With the Fmax improvement, geomean wall-clock time improves by 4.7%. With divider sharing in Arch. 3, logic utilization and M20K usage drop. Altera's divider cores use M20Ks within, thus memories are also saved in sharing dividers. There is virtually no impact on execution cycles (0.1% increase). This is because in our system, threads are started in a staggered manner (i.e. one after another), and dividers are pipelined (to the depth equal to the operand's bitwidth). Thus, when sharing dividers across threads, stalls caused by divider contention among threads are minimal. When sharing multipliers in Arch. 4, DSP usage drops as expected, and execution cycles are again affected minimally. Logic utilization does increase, however, due to multiplexers required on the inputs of the multipliers. In Arch. 3, logic utilization decreased when sharing dividers only, since the decrease from sharing dividers exceeds the increase from the added input multiplexers. Performance significantly improves in Arch. 5, owing to the local and shared-local memories. Compared to Arch. 4, execution cycles and total execution time improve by 14.3% and 32.6%, respectively. The local/shared-local memories also shrink the expensive multiplexers in the global memory controller (as described in Section V-B), improving logic utilization and Fmax by 11.1% and 27.3%, respectively. M20K usage drops with local memories, since Quartus is able to perform more optimizations, such as reducing a RAM block to registers, when memories directly connected to the data-path. When RAMs are created inside the global memory controller, behind multiplexers, Quartus is not able to perform such optimizations. However, Quartus cannot automatically convert all small memories to registers, which we handle in Arch. 6.

Time	Execution Cycles	Fmax	Logic Util.	DSPs	M20Ks
262.86	45562.53	173.34	3727.72	11.03	39.38

TABLE I: Geomean baseline results (Arch. 1).

With memory-to-register conversion, M20K usage decreases by 34.1% from Arch. 5, and logic utilization decreases by 7.1%. We also reduce the memory-load latencies to 1 clock cycle for local/shared-local memories and to 0 clock cycles for memories converted to registers. This improves both execution cycles and wall-clock time by an additional 5.7%, compared to Arch. 5. In Arch. 7, we localize ROMs to each thread through replication to reduce memory contention between threads. With this, execution cycles and total execution time improve by 5.7% and 3.6% respectively, relative to Arch. 6. M20K usage increases, however, by 1.97$\times$ due to replication. In Arch. 8, we disable multiplier sharing across threads, as the input multiplexers can increase circuit area and lower Fmax. The execution cycles improves minimally, and the logic utilization improves by 10%, compared to Arch. 7. As expected, DSP usage also increases significantly by 2$\times$. Overall, Arch. 8 yields the best performance and logic utilization results out of all architectures, with an improvement of 41.6% (wall-clock time) and 38.3% (logic utililization), compared to Arch. 1.

Area-delay product is a well-known metric to gauge circuit efficiency. However, modern FPGAs contain different types of blocks, which include logic blocks, memory blocks, DSP blocks, and routing, each of which consumes a different amount of area on the FPGA. To account for the different types, we use the area data from [10], which gives the chip tile area for each type of block[5]. With the total area accounting for the different types of blocks, we multiply this with wall-clock time to obtain the

[5]Although [10] provides detailed area data for the types of blocks in Stratix III. Stratix V contains similar types of blocks, so we believe the data can be used for this relative area comparison. We calculate the M20K area from the given M9K area by using the relative memory capacity ratio.

area-delay product for each architecture.

Fig. 11 shows the geometric mean area-delay product for each architecture. As seen in the figure, the area-delay product improves from Arch. 1 up to Arch. 6, at which points it starts to become worse. This is because the performance generally improves up to Arch. 6, with the area also significantly reduced by sharing functional units, reducing multiplexing logic, and converting RAMs to registers. From Arch. 7, the performance continues to improve slightly, however, area is increased significantly when replicating memories in Arch. 7 and disabling multiplier sharing in Arch. 8. Overall, Arch. 6 shows the best area-delay product, with an improvement of 63.3% over Arch. 1.

Overall, we observed that local/shared-local memories and memory-to-register conversion, together with reduced access latencies, significantly improve performance and area. Constant memory replication also helps to reduce memory contention further, but degrades area-delay product. Sharing functional units across threads had little impact on performance degradation, while producing considerable area savings. This is because threads are invoked at different clock cycles, making them slightly "out-of-step" with one another, thereby reducing contention.

VII. Conclusions and Future Work

In this paper, we analyzed the impact of two circuit topologies with different memory management techniques on the performance and area of parallel HLS-generated hardware. We considered the nested topology, where hardware modules are instantiated in a hierarchical manner, and the flat topology, other where modules are instantiated at the same level of hierarchy. The flat architecture enables greater sharing of hardware modules and functional units. We described a system generator, integrated within HLS, to automatically create efficient interconnect between hardware modules, with the ability to share/replicate functions, functional units, memories between functions/threads, as well as inserting deadlock-prevention modules for parallel operating hardware. Three different memory architectures were also investigated: the global memory controller, and local/shared-local memories. The global memory controller handles memory accesses that are not amenable to points-to analysis. Local and shared-local memories improve memory bandwidth by providing concurrent direct memory accesses, and decrease area by reducing the multiplexing logic. Additional memory management techniques, memory replication and memory-to-register conversion, were explored to reduce memory contention between threads and also to reduce memory usage and latency.

In geomean results, the flat circuit topology with local/shared-local memories, divider sharing, memory-to-register conversion, and replication of constant memories (Arch. 8) yielded the best result with 41.6% and 38.3% improvement in wall-clock time and logic utilization, respectively, relative to the baseline. Replication of constant memories and multipliers increased area, thus the best area-delay product was observed with Arch. 6, which used the flat circuit topology with local/shared-local memories, divider/multiplier sharing, and memory-to-register conversion, offering an improvement of 63.3% vs. the baseline.

For future work, we would like examine the proposed techniques in a scenario where the HLS of threads is *combined* with function pipelining; i.e. wherein parallel threads execute in concurrent pipelined hardware modules.

References

[1] Algorithm and Programing. *Box Filtering. (http://tech-algorithm.com/articles/boxfiltering/)*.

[2] L. O. Andersen. Program analysis and specialization for the c programming language. In *Ph.D. Thesis*. University of Cophenhagen, 1994.

[3] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4, 1965.

[4] A. Canis, S.D. Brown, and J.H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *IEEE FPL*, pages 1–8, Sept. 2014.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S.D. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA FPGA*, pages 33–36, 2011.

[6] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *IEEE FPT*, pages 270–277, December 2013.

[7] J. Cong and W. Jiang. Pattern-based behavior synthesis for fpga resource reduction. In *ACM/SIGDA FPGA*, pages 107–116, 2008.

[8] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-enabled loop pipelining for high-level synthesis. In *DAC*, pages 1–6, San Francisco, CA, June 2014.

[9] D. Cabrera et. al. OpenMP extensions for FPGA accelerators. In *IEEE Systems, Architecture, Modeling and Simulation (SAMOS)*, pages 17–24, 2009.

[10] H. Wong et. al. Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. In *ACM Int'l Symnp. on FPGAs*, pages 5–14, 2011.

[11] Y.Y Leow et. al. Generating hardware from OpenMP programs. In *IEEE FPT*, pages 73–80, 2006.

[12] G. Q. Silva, https://code.google.com/p/addrleaks/. *Static Detection of Address Leaks.*, 2013.

[13] S. Hadjis, A. Canis, J.H. Anderson, J. Choi, K. Nam, S.D. Brown, and T. Czajkowski. Impact of FPGA architecture on resource sharing in high-level synthesis. In *ACM/SIGDA FPGA*, pages 111–114, 2012.

[14] Robert J. Halstead and Walid Najjar. Compiled multithreaded data paths on fpgas for dynamic workloads. In *IEEE Compilers, Architectures and Synthesis for Embedded Systems*, 2013.

[15] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[16] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Programming language design and implementation*, pages 290–299, 2007.

[17] http://llvm.org/docs/Passes.html. *LLVMs Analysis and Transform Passes*, 2015.

[18] http://www.altera.com/products/software/ opencl/opencl-index.html. *OpenCL for Altera FPGAs*, 2013.

[19] http://www.xilinx.com/products/design_ tools/vivado/vivado-webpack.htm. *Xilinx: Vivado Design Suite*, 2013.

[20] http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf. *Xilinx: Xcell Journal, Issue 86*, 2014.

[21] D. Ku and G. D. Micheli. *High Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Norwell, MA, 1992.

[22] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. Resource-aware throughput optimization for high-level synthesis. In *ACM/SIGDA FPGA*, pages 200–209, 2015.

[23] A. Morvan, S. Derrien, and P. Quinton. Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion. In *IEEE FPT*, pages 1–10, Dec. 2011.

[24] Oregon Medical Laser Center. *Monte Carlo Simulations. (http://omlc.ogi.edu/software/mc/)*.

[25] L. Semeria and G. De Micheli. Spc: synthesis of pointers in c application of pointer analysis to the behavioral synthesis from c. In *IEEE/ACM ICCAD 98. Digest of Technical Papers.*, pages 340–346, November 1998.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 32–41, 1996.

[27] G. Stitt and F. Vahid. Thread warping: a framework for dynamic synthesis of thread accelerators. In *IEEE/ACM CODES+ISSS*, pages 93–98, 2007.

[28] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded pipeline synthesis for data-parallel kernels. *ICCAD*, 2014.

[29] T. Yuki, A. Morvan, and S. Derrien. Derivation of efficient fsm from loop nests. In *IEEE FPT*, pages 286–293, Kyoto, Japan, December 2013.