Synthesis of Statically Analyzable Accelerator Networks From Sequential Programs

Shaoyi Cheng
EECS, University of California, Berkeley
sh_cheng@berkeley.edu

John Wawrzynek
EECS, University of California, Berkeley
johnw@eecs.berkeley.edu

ABSTRACT

This paper describes a general framework for transforming a sequential program into a network of processes, which are then converted to hardware accelerators through high level synthesis. Also proposed is a complementing technique for performing static deadlock analysis of the generated accelerator network. The interactions between the accelerators' schedules, the capacity of the communication channels in the network and the memory access mechanisms are all incorporated into our model, such that potential artificial deadlocks can be detected and resolved *a priori*.

An algorithm optimized for FPGA implementation is developed and applied through our transformation framework. A set of irregular computation kernels are converted into networks of FPGA accelerators. Compared to hardware accelerators generated without our transformation, the accelerator networks achieve significantly better performance.

Keywords

process network, high level synthesis, system design flow, hardware accelerator

1. INTRODUCTION

As the growth of processor clock frequency came to a stop in the middle of the last decade, the performance boost provided by the continuous advancement of hardware technology for sequential programs could no longer be sustained. The benefits of user transparent CPU architecture optimizations were also diminishing. With the recent advent of FPGA SoCs [1], offloading computation to hardware accelerators becomes a promising alternative for achieving better system efficiency. However, there is a large gap between the inherently parallel mode of operation in hardware accelerators and the intuitive imperative programming paradigm. Even with tool flows like high level synthesis (HLS), substantial user inputs are still required to create efficient implementations [2]. To address this problem, we envision a two step process, where the sequential specification provided by the user is first transformed to a more parallel representation, from which conventional HLS can easily create a high performance compute engine.

In this paper, we explore this approach by using a modified version of the process network (PN), a parallel model of computation, as the target representation for our transformation. Using FPGA as the computing device, our final implementations are accelerator networks optimized to achieve high throughput in the presence of long data access latency. Our major contributions include the following.

- We devise a general approach for converting a sequential program to a network of concurrent processes. Different algorithms can be inserted into the flow, splitting computations between processes in different ways, optimized for the underlying compute substrates.
- With FPGA as our target platform, processes are to be converted to hardware accelerators. We propose an analysis framework to statically detect and resolve deadlocks in the accelerator networks, ensuring the correctness of the execution.
- We develop an algorithm to generate instances of process networks particularly amenable to hardware acceleration. Using high level synthesis (HLS), we map them onto FPGAs to demonstrate the benefits of our approach.

The rest of the paper is organized as follows: Section 2 looks at background and related work in process networks and accelerator generation using HLS. Section 3 describes the main steps in transforming sequential programs to networks of processes. When implemented on FPGAs, these networks may experience deadlock issues, which can be statically analyzed and resolved with the method developed in section 4. To make our general approach more concrete, we present a specific algorithm which fits into our PN generation framework in section 5. This algorithm is developed with FPGA as the target platform, thus the generated networks boast great performance when mapped onto the intended compute substrate (section 6). Finally, we conclude the paper in section 7.

2. BACKGROUND AND RELATED WORK

2.1 Process Networks

The process network, or Kahn process network [3], is an inherently parallel model of computation where processes communicate with each other through unbounded FIFO channels. The processes block when they read from empty channels but never block when writing. The PN model is deterministic in the sense that the scheduling of the process execution does not alter the final results. This gives great flexibility in implementing/controlling each process.

A characteristic of KPN is that all data exchange are performed over the FIFOs and the notion of a shared global memory does not exist in the model. Previous work converting sequential programs to process networks [4][5] were

ICCAD '16, November 07 - 10, 2016, Austin, TX, USA

ISBN 978-1-4503-4466-1/16/11...$15.00

DOI: http://dx.doi.org/10.1145/2966986.2967077

able to convert original memory operations to explicit data communication at the boundary of the generated hardware as the targeted kernels are highly regular. In this study however, we target kernels with irregular memory accesses and the final implementations are to be used along side a general purpose processor. We therefore preserve the shared memory, but conceptually model data accesses as inter-process communications. More specifically, the memory, and its associated interconnect can be seen as a special process which takes in address/data tokens and produces data/acknowledge tokens. In the case where multiple memory ports are exposed, this "process" does not block on read, as it can choose to serve any port with valid requests. This behavior is similar to the "select" function in YAPI [6], which introduced non-determinism. In our framework however, the non-determinism is addressed by a set of special mechanisms, as explained in section 3.1.

Another necessary deviation from the ideal PN model lies in the capacity of the channels. In any practical implementations of process networks, the FIFOs cannot be unbounded. Consequently, in cases with circular dependencies, when buffers fill up writes may block and deadlocks can occur. These are so-called artificial deadlocks, as their occurrence is introduced by the nonideality of the implementation. In a process network, these deadlocks, if left unresolved, would cause a mismatch between the resultant behavior of the original code and the network. To tackle this problem, many approaches were proposed over the years, most of which involve monitoring execution and dynamically incrementing FIFO sizes when deadlock occurs [7][8][9], or only target highly regular kernels. In our flow, we ensure, by construction, each FIFO is only read/written by a single instruction in each process, which makes the generated PN easier to analyze than a generic PN. We can therefore use a static analysis framework to find the needed FIFO sizes.

2.2 High Level Synthesis of Accelerators

HLS for FPGAs are being actively developed in both industry [10] and academia [11]. These HLS tools attempt to capture parallelism in a function written with high level languages like C/C++. With the help of various user provided guidance, these tools generate hardware in the form of RTL. The synthesized circuits usually follow a Finite State Machine with Datapath (FSMD) paradigm. Activation of a particular operator in the datapath is associated with a certain clock cycle and the execution of the computation engine is orchestrated by a synthesized central controller.

With this static scheduling of operations, the accelerators' runtime behavior is rather simple. Different parts of the generated circuit run in lockstep with each other, no dynamic dependency checking mechanisms, such as scoreboarding or load-store queueing, are needed. This rigid scheduling of operators, while producing simpler hardware with smaller area, is also vulnerable to stalls introduced by cache misses or variable latency operations. This often results in suboptimal performance when the accelerators are mapped onto FPGAs. The algorithm in section 5, was developed to specifically deal with this issue. Thus the networks generated with the algorithm, when implemented physically, have much better performance.

3. FROM SEQUENTIAL PROGRAM TO PROCESS NETWORK

Our PN generation flow is built on top of the LLVM framework [12], which uses a single static assignment (SSA) intermediate representation (IR). A typical function in a sequential program is represented with a control flow graph (CFG). This is a two level hierarchical structure. Instructions are contained within basic blocks which are connected by directional edges representing control transfers. Meanwhile, every instruction defines a new variable in the SSA representation, thus the data dependencies between instructions are explicitly defined. To generate the process network from this representation, a few main steps are performed:

- *Dependency Annotation:* The hierarchy in the IR is flattened by branch predication. Ordering of memory operations is also enforced by the addition of special dependency edges into the graph.
- *Instruction Cluster Formation:* The instruction nodes, with various dependencies between each other, are partitioned into multiple sets. This is the core part of the flow. Different algorithms targeting compute substrates of different characteristics can be applied in this step.
- *CFG Reconstruction:* From each set of instruction nodes, a CFG is reconstructed to form a self-contained process. Dependencies between the instruction nodes are realized by the communication between the recreated CFGs.

3.1 Dependency Annotation

To facilitate the instruction partitioning, we perform predication to eliminate the basic block boundaries, creating a unified graph of nodes with the addition of control dependency edges. The simplest way to insert these edges is to have every instruction dependent on the last branch of the immediate predecessors of its container basic blocks. However, we take a more aggressive approach, looking for the earliest control transfer which necessarily leads to the execution of an instruction. Given an instruction i and its container basic block bb, our flow looks for the nearest set of basic blocks BB' who are not properly post-dominated by bb, and insert the dependency edge between the branch instructions ending each member of BB' and i. Compared to the naive approach, our method uses a single control edge to replace a sequence of edges traversing multiple branches, reducing the overall communication overhead in the network. Also with the elimination of unnecessary edges, the subsequent steps can be more flexible, potentially leading to a more efficient final implementation.

Another type of dependency is implicitly carried by memory accesses. When two memory accesses target the same location and one is a store, their order in the original program execution needs to be preserved—a dependency edge is added between them. We try to avoid adding unnecessary edges as they can potentially affect the instruction partitioning negatively. The flow currently relies on alias analysis to perform partitioning of the memory space. No edges are inserted for accesses to disjoint memory partitions. Meanwhile, for code harder to analyze, the flow can take in user annotations, complementing the overly conservative compile time analysis. With the addition of these edges, the ordering of requests sent to the memory subsystem is constrained to be consistent with the original program execution. The

non-determinism caused by memory selecting among multiple requests no longer impacts the final outcome. In general, for any pair of instructions separated into two sets, the necessary ordering between them is enforced by the cut edges, which will eventually be converted to communication channels between processes.

3.2 Instruction Cluster Formation

3.2.1 Partitioning Instructions

To create multiple processes, the instruction nodes are to be distributed into multiple sets. The actual partitioning of instruction nodes can use different algorithms, targeting compute substrate of different characteristics. In our implementation, a predefined interface is created, which takes in a fully annotated dependency graph and outputs separate collections of instructions. These can then be fed to the next stage of our flow. This is a convenient way to experiment with various techniques, leveraging other parts of the framework to reconstruct processes, detect/resolve deadlocks and generate concrete implementation. For instance, the decoupling method described in [13] can be easily plugged into the flow, generating network optimized for multicore processors. We have also developed our own algorithm targeting FPGAs, which is detailed in section 5. There may even be scenarios where work can be divided up amongst FPGA accelerators and CPUs, requiring the partitioning routine to consider characteristics of multiple platforms simultaneously. For any of these cases, our infrastructure can be used with minimal change.

3.2.2 Optimizing Communication vs. Computation

Regardless of exactly how the instructions are partitioned, as each process performs its computation locally, the necessity to communicate with its peers can incur implementation overheads. For each communication channel, buffer space is to be allocated, and sender/receiver primitives also need to be inserted. Therefore, it is sometimes beneficial to duplicate some nodes after the instruction partitioning. Given sets of instruction nodes created after applying a platform-specific algorithm, for each individual set G, the problem of whether to copy other instruction nodes into G can be formulated as an integer programming problem.

Let T be the set obtained with nodes duplicated into G, and the set of remaining nodes be S. Each node i can be associated with a binary variable p_i, where $p_i = 0$ if $i \in T$ and $p_i = 1$ if $i \in S$. Meanwhile, each edge ij is associated with a binary variable d_{ij} which takes the value 1 if $i \in S$ and $j \in T$, and 0 otherwise. Finally, w_{ij} is used to represent the cost of edge ij while c_i represents the cost of the node i.

$$\text{Minimize} \quad \sum_{ij \in E} w_{ij} d_{ij} + \sum_{j \in T} c_j \tag{1}$$

$$\begin{aligned} \text{subject to:} \quad & d_{ij} - p_i + p_j \geq 0, && ij \in E \\ & p_i \in \{0,1\}, && i \in V \\ & d_{ij} \in \{0,1\}, && ij \in E \\ & p_g = 0, && g \in G \end{aligned}$$

Depending on the implementation platform, the cost of the edges and nodes can be assigned differently. This formulation however, is platform-neutral and generally applicable. With the CPLEX [14] optimization software, this ILP can be solved heuristically and the sets of instructions can be updated accordingly.

3.3 CFG Reconstruction

From each set of instructions generated, an executable process is created by constructing an independent control flow graph. This step recreates basic blocks to in the new CFG, while inserting communication primitives to send and receive tokens through FIFO channels.

- For every instruction i in the current set, recreate its container basic block if it's not already in the CFG.
- For every instruction j on which i depends, recreate its container basic block if it's not already in the CFG.
- If j is assigned to another set of instructions, a load operation is created locally as a placeholder in its container basic block.
- If i is producing operands for instructions assigned to other sets, a store operation is inserted after i.

The set of recreated basic blocks B are thus associated with either the instructions assigned to the current set, or the placeholder instruction supplying them with operands. The insertion of load and store operations is to accommodate various dependencies between the processes. Note that each pair of these inserted communication primitive is associated with a single instruction in the original CFG. The flow of tokens between them ensures the execution path are synchronized across different CFGs and the right operands are supplied for computations distributed across the process network.

To form execution paths in the new CFG, we find the nearest common dominator d of all the recreated basic blocks from the original CFG. It is added to B and used as the new entry block. Again from the original CFG, we trace all the paths between member basic blocks in B, and add the points of divergence into B. These points are basic blocks which cause the control flow to go into different members in B, or to basic blocks not in B at all. After all member basic blocks are connected, we have a self-contained process. During this construction process, for the branch instructions not already assigned to the current CFG, a load operation is created to accept a branch target token from another process. The local control flow can then be performed according to this received token. Finally, if the entry block d was inside a loop in the original control flow, any control transfer out of the set B is redirected to d. We are essentially enclosing the process with a *while(true)* loop and the execution of the new CFG will be repetitively activated by the availability of the proper tokens.

In figure 1, the function below is converted from its SSA form to a process network. A naive instruction partitioning algorithm is used in this example. The loop control flow, the memory accesses and the computation are separated into three different processes.

```
float foo(float* x, float* prod, int* ind)
{
  float curProd = 1.0;
  for(int i = 0; i < N; i++){
    int curInd = ind[i];
    float curNum = x[curInd];
```

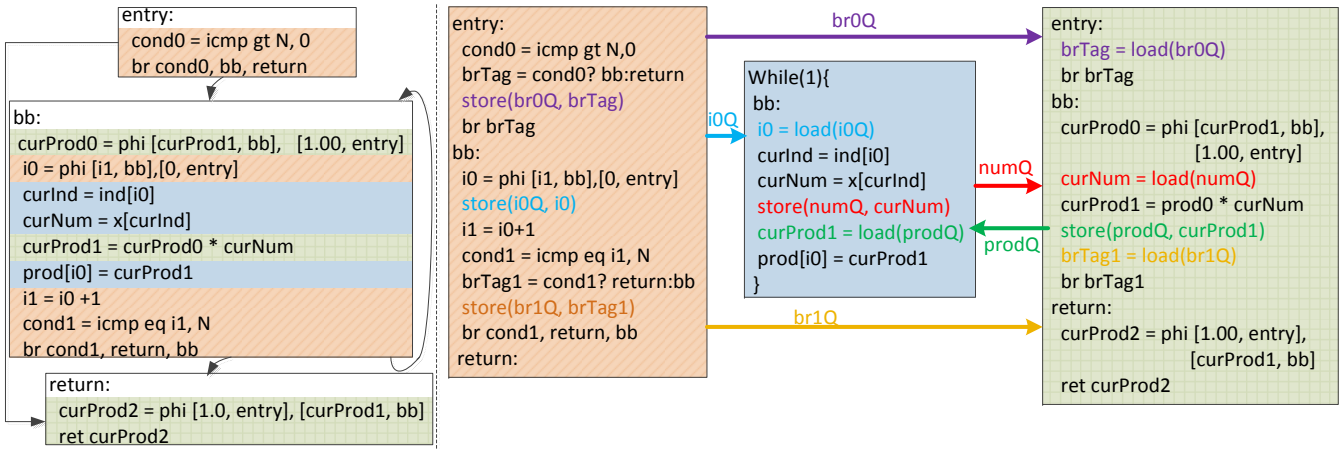

Figure 1: From A Single CFG to A Process Network

```
    curProd = curProd * curNum;
    prod[i] = curProd;
  }
  return curProd;
}
```

When this network runs, it writes the same data into the memory, and eventually return the same value as the original program. As it completes execution, its processes will either return or be blocked on an empty FIFO. However, whether it can successfully complete hinges on the absence of artificial deadlocks—which we look at next.

4. ARTIFICIAL DEADLOCK ANALYSIS

4.1 Boundedness of the Generated PNs

As mentioned in section 2.1, it is impossible to have unbounded communication FIFOs between processes in any real PN implementations. For a PN to execute without experiencing artificial deadlocks, it has to be bounded, i.e. certain schedule of the network only requires finite FIFO space. For a general PN, its boundedness is an undecidable problem [7]. However, for the process networks generated using our scheme, we can easily show their boundedness by constructively creating a schedule of operations.

The main observation here is that for every token sending instruction, there is a corresponding token consuming instruction. Furthermore, they both derive from the same instruction in the CFG before partitioning. Therefore, we can create a schedule H for our network following the execution of instructions in the original program. For every instruction executed in the original CFG, we schedule the PN's instructions derived from it in H. If these instructions are involved in communication, the source of the data tokens is scheduled first, immediately followed by the token sinks. It should be apparent that only one slot is needed in each communication channel for the process network to execute, since the produced tokens are promptly consumed.

4.2 Instruction Schedules and Deadlocks

Now let the schedule of instructions in each process p be $G(p)$, we say $G(p)$ is *locally consistent* with H if and only if $\overline{i_a} \prec \overline{j_b}$ in $H \Rightarrow i_a \prec j_b$ in $G(p)$, where $\overline{i_a}$, $\overline{j_b}$ are invocations of instructions in H, and i_a, j_b are their equivalences in p.

LEMMA 1. *Assuming all FIFOs are of size one, as long as $G(p), \forall p \in PN$ are locally consistent with H, artificial deadlock will not occur in PN.*

Assuming there is an artificial deadlock, we can go around the dependency cycle and examine the blocked processes $P_b \subseteq PN$. For a process p_b to be blocked at instruction invocation j_0, j_0 is either reading from an empty FIFO, or writing to a full FIFO. For the former case, the FIFO is empty because the token's producer instruction $j'_0 \in p'_b$ cannot execute, which indicates an earlier instruction i'_k in $G(p'_b)$ is blocked. For the later case, the FIFO is full because the token produced by an earlier invocation j_{-1} has not been taken by its corresponding consumer instruction $j''_{-1} \in p''_b$. This also indicates an earlier instruction l''_n in $G(p'')$ is blocked. Applying this reasoning recursively around the circle of dependencies, we should have a chain of precedence among instructions invocations $j_0 \succ i'_k \succ ... \succ j_0$ or $j_0 \succ j''_{-1} \succ l''_n \succ ... \succ j_0$ in H. Both chains are self-contradictory and therefore, the scenario for artificial deadlock can never occur.

Since H is derived from the original CFG, with Lemma 1 proven, we can conclude that if each of the generated process is executed strictly according to the original program order, we will have a network free from artificial deadlocks. This will guarantee, for instance, when processors are used as the compute substrate, the network will produce the correct result as each process is executed according to locally consistent $G(p)$. On the other hand, when HLS is used to create hardware accelerators from processes, this guarantee may not hold anymore since aggressive parallelization and reordering of instructions would violate the consistency between $G(p)$ and H.

Figure 2 is an example PN we can use to illustrate this problem. The original loop is converted to a two-process network with two single slot communication channels. Figure 3 then shows the scheduling and communication of the processes. Each instruction invocation is prefixed with a vector, representing its place in the (j,i) iteration space, and three types of dependency edges are used to show the necessary precedence among these invocations. There is no cycle of dependency in (a), and therefore no deadlocks, when both processes execute in program order. However, in (b), a common HLS technique, loop pipelining, is applied to *Process 1*, where iterations of the loop are aggressively over-

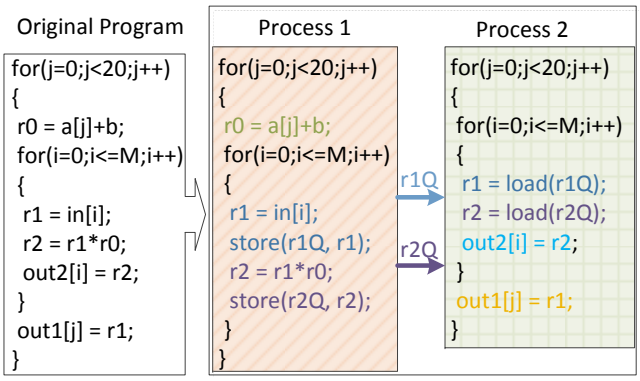

Figure 2: Example PN for Deadlock Analysis

lapped. Every cycle, a new iteration is started. Because of the long latency of load from *in[i]* and the multiplication, the sending of the first *r2* token, $_{(0,0)}$store(r2Q,r2), is scheduled after many instruction invocations for later iterations. Due to the static scheduling, it can not execute until after $_{(0,2)}$store(r1Q,r1), whose writing to the FIFO r1Q cannot happen until the previous token is consumed by $_{(0,1)}$load(r1Q). Because of the scheduling in process 2, this load from r1Q occurs after $_{(0,0)}$load(r2Q), which cannot complete unless $_{(0,0)}$store(r2Q,r2) can send the token. An artificial deadlock is therefore created.

The classic approach to resolve artificial deadlock is by increasing the size of the FIFO where write is blocked [7]. In this particular example, if the size of r1Q is increased from 1 to 2, then the execution of $_{(0,2)}$store(r1Q,r1) would be dependent on $_{(0,0)}$load(r1Q) instead of $_{(0,1)}$load(r1Q). The cycle of dependence is broken as there will be no path of dependence from $_{(0,0)}$load(r1Q) to $_{(0,0)}$store(r2Q,r2). The deadlock no longer exists.

Unfortunately, in a network implemented in hardware, the FIFO size cannot be easily increased. To determine the required buffer space in the channels *a priori*, simulations are sometimes used [15]. It relies on using representative datasets as input andmay not provide any guarantees.In this section, we analyze the interaction between FIFO sizing and intra-process instruction scheduling to devise a non-simulation based deadlock prevention algorithm.

4.3 Precedence Graph for Deadlock Detection

To directly detect cycles from the schedules of processes as we have done in figure 3 is unrealistic as each schedule may contain potentially infinite instruction invocations. However, HLS tools generate a short, repeatable schedule for each process. We can leverage this fact to create a more concise representation—a precedence graph.

An example of a precedence graph is shown in figure 4, the schedules of instruction invocations are condensed into a small graph whose edges carry vector weights. The weights of the *DepS* edges are simply the worst case differences between the vector prefixes of invocations of a pair of instructions. Similarly, the weights of *DepF* edges correspond to the differences in prefixes between the consumer and producer instruction invocations. Note this is directly related to the number of slots in the FIFO. For our example, the edge from store(r1Q,r1) to its consumer load(r1Q) carries a weight of (0,-1) when the channel has just one slot. It means the previous invocation of the consumer instruction needs to complete before the producer instruction can write more tokens. The *DepD* edges, on the other hand, always have weight $\vec{0}$ as the consumer depends on the producer with the same iteration vector. In essence, this graph summarizes the necessary precedence between instruction invocations.

In the precedence graph, every simple cycle has a weight sum which is also a vector. A deadlock manifests as a cycle with weight being the zero vector ($\vec{0}$) or vectors whose first non-zero elements (leading elements) are positive. They indicate an instruction invocation depending on itself or a later invocation in the execution, and thus a deadlock. The computation to find deadlocks may potentially be expensive. In the worst case, the number of simple cycles in the graph can be exponential in $|V|$, the number of nodes in the graph. However, practically, the number of nodes involved are approximately the same as the number of channels needed, as we only need to look at the loads and stores from/to FIFOs. The graph is also rather sparse in the connectivity between these nodes. Generic graph algorithms such as [16] can be used for efficient enumeration of the cycles and the subsequent identification of the deadlocks.

Resolving the deadlock involves choosing one of the *DepF* edges in the cycle and adding enough slots (making the weight more negative) such that the weight of the cycle gets a negative leading element. For our example in figure 4, if the capacity of r1Q is increased by 1, the weight on its corresponding *DepF* becomes (0,-2). Consequently, the identified cycle in the figure would have a total weight (0,-1), when the original artificial deadlock disappears. In general, for every deadlock causing cycle, the *DepF* edge whose channel has the smallest width can be selected for capacity expansion, so the cost of these extra slots is minimized. This heuristic does not necessarily produce an absolute global optimal solution in the presence of multiple deadlock causing cycles, but suffices for our use cases.

4.4 Deadlocks in Memory Accesses

To exploit the memory bandwidth more efficiently, burst mode memory accesses are often used in accelerators. Even though the memory "process" does not block on reads, if the response data it sends to other normal processes is not consumed promptly, it can blocked on writes. The head-of-line blocking can then create a dependency cycle, resulting in a deadlock. Our precedence graph can be used to compute the necessary buffer size to resolve this situation as well.

Figure 5 illustrates how multiple burst mode memory accesses can create a deadlock. The reaction of the memory is modelled by a pop action responding to the incoming request, and a push action producing one response data token. Meanwhile, the sending of memory requests and receiving of responses are decoupled and assigned to different slots in the process schedule. This is often done in HLS because of the long latency of memory accesses. For burst mode requests associated with a loop, the request sending is moved outside of the loop as shown in the figure. The symbolic variable B in the weight of the *DepM* edge signifies how many tokens have to be consumed/buffered before the response for the next request (*data*2) become accessible for the response receiver. Assume the arbiter for the memory subsystem implements a policy which can potentially take in and serve multiple (K) consecutive requests from one stream, B can be as big as $BurstSize_{max} * K$. In the case of a round-robin arbiter, $B = BurstSize_{max}$ and for our experiment plat-

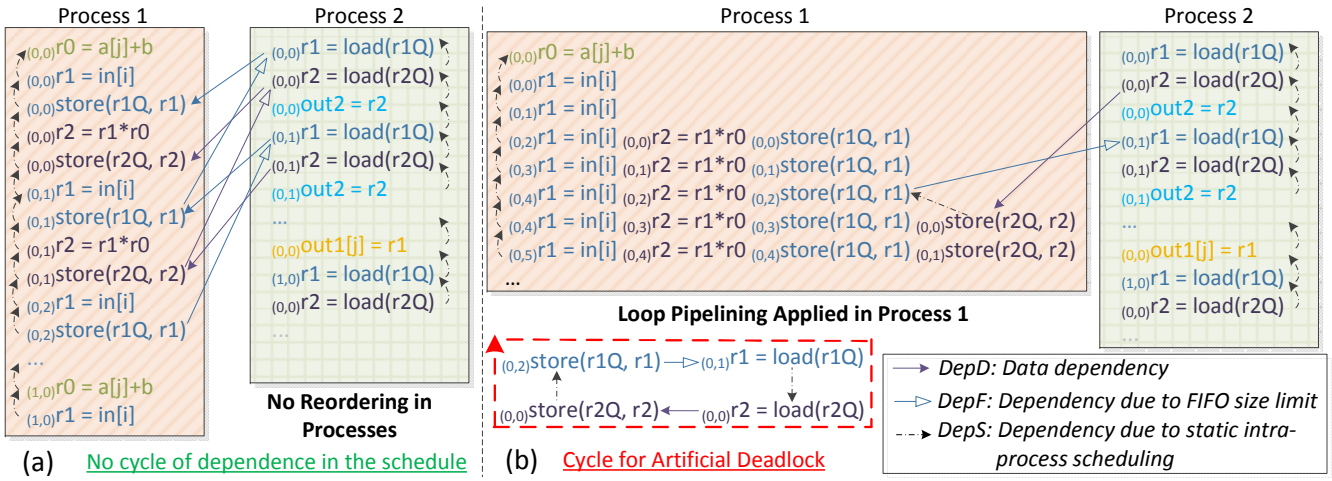

Figure 3: (a) No Reordering, No Deadlocks (b) Loop Pipelining Introduces Deadlock

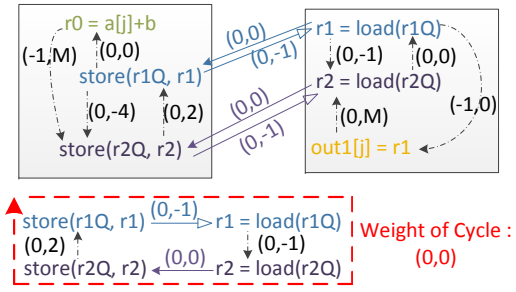

Figure 4: Precedence Graph for Process Network

form which uses the AXI bus protocol, this number is 256. With the precedence graph, we can easily compute that the buffering for the two memory response channels needs to be increased to B for the deadlocks to be resolved.

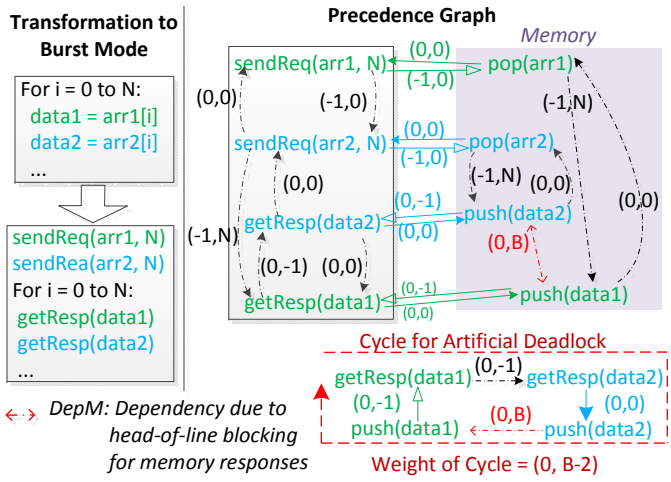

Figure 5: Deadlock in Burst Memory Accesses

5. A MEMORY CENTRIC INSTRUCTION PARTITIONING ALGORITHM

So far, we have developed a framework to generate process networks from sequential programs and a static analysis method to ensure sufficient FIFO space is allocated to prevent artificial deadlocks. This section provides a concrete application of this framework to generate high performance compute engines. The discussion in section 2.2 revealed the weakness of HLS generated FPGA accelerators, which can be alleviated by distributing memory accesses into different processes, each running its own schedule. Off-chip communication and computation are naturally overlapped and the overall throughput of the compute engine can be greatly improved.

More specifically, several factors are considered in this instruction partitioning algorithm. The starting point is a graph G thoroughly annotated with various dependency information. The group of nodes which forms dependency cycles are first identified. They are associated with loop carried dependencies and are the limiting factors for how aggressively loop iterations can be overlapped. The initiation interval (II) of loops are dictated by the latency of these cycles. As the communication channels will always add latency, it is undesirable to have these cycles spanning multiple sets. Secondly, we want to have memory operations separated from dependency cycles involving long latency computation, such that the stalls caused by cache misses no longer affect the computations running in another process. Lastly, to localize the effects of stalls introduced by cache misses, the number of memory operations in each set should be minimized, especially when they address different parts of the memory space.

The steps taken to achieve the aforementioned requirements are detailed in algorithm 1. The SCCs are collapsed into new nodes, which together with the original instruction nodes, are topologically sorted. The obtained directed acyclic graph is traversed and a new set is created whenever a memory operation or an SCC with long latency computation is encountered. Here, long latency operations are those which cannot be completed within one clock cycle, and their categorization ultimately depends on the target frequency of the final implementation on the FPGA. Currently, we leverage Xilinx's Vivado HLS to generate latency estimates for various compute operations. With a target clock frequency of 150MHz, for instance, floating point multiply takes four clock cycles while a 32 bit integer addition can be completed within a cycle. As Vivado HLS is eventually used as the backend for our HDL generation, it provides accurate annotations for our flow.

Algorithm 1 Instruction Partitioning

```
1: procedure PartitionInstNodes(G)
2:     SCCs ← allStronglyConnComps(G)
3:     DAG ← collapse(SCCs,G)
4:     topoSortedNodes ← topologicalSort(DAG)
5:     longSCCs ← getSCCWithLongOp(SCCs)
6:     memNodes ← findLdStNodes(G)
7:     memLongSCC ← LongSCCs ∪ memNodes
8:     allSets ← {}
9:     curSet ← {}
10:    while topoSortedNodes ≠ ∅ do
11:        curNode ← topoSortedNodes.pop()
12:        curSet ← curSet ∪ curNode
13:        if curNode ∈ MemLongSCC then
14:            allSets ← allSets ∪ curSet
15:            curSet ← {}
16:        end if
17:    end while
18:    return allSets
19: end procedure
```

6. EXPERIMENTAL EVALUATION

To measure the benefits our approach, a set of benchmarks are processed by our flow. The partitioning algorithm described in section 5 is applied together with the optimization step formulated in part 3.2.2. These benchmarks are kernels with non-regular control flow and memory access patterns. Their descriptions are listed in table 1. As the input datasets are too large for on-chip buffers, and the memory accesses being too irregular for simple DMA, these are cases where our flow can play a significant role in improving the overall performance.

The FPGA device used for our evaluation is Xilinx's Zynq-7000 XC7Z020 FPGA SoC. It contains two parts: an ARM-based processing system (PS), and the programmable logic (PL). The baseline for our evaluation is the performance of each kernel executing on the ARM core, which is an out-of-order, dual-issue hard processor running at 667MHz. Zynq also provides two options for the accelerators in PL to access the main memory: through the accelerator coherence port (ACP), or the high performance (HP) port. The ACP connects to the snoop control unit in the PS, and thus accesses the on-chip cache of the PS. On the other hand, the HP port connects directly to the memory controller, thus often incurs longer latency for data requests.

Table 1: Benchmark Descriptions

Benchmark	Description	Data Size
Floyd-Warshall	Graph Algo. Dynamic Programming Mem. Accesses Depend on Computation	≈ 8 MB
Knapsack	Optimization, Dynamic Programming Mem. Accesses Depend on Computation	≈ 5 MB
Depth-First Search	Stack-based Graph Algo. Irregular Memory Access Pattern	≈ 3 MB
SpMV Multiply	Matrix in Compressed Row Storage Indirect Memory Addressing	≈ 16 MB

In our study, Vivado HLS, a state-of-the-art high level synthesis tool provided by Xilinx, is used to create the accelerators for the original sequential code, as well as the processes generated by our flow. With the target clock period set to 8ns during HLS, the tightest timing constraints post place & route implementations managed to meet range from 111 to 150MHz. All design points shown in this section use the highest achievable frequency as the actual operating clock frequency.

6.1 Performance Comparisons

In figure 6, performance of the different implementations are presented. Conventional accelerators and accelerator networks with different memory subsystem configurations are compared. All the numbers are normalized to the baseline. Design points with caches use the Xilinx System Cache IP, configured to be 64KB and 2-way associative.

In all four benchmarks, conventional accelerators directly generated from software kernels actually have lower performance than the hard processor. Even with on-PL caches, they can only achieve throughput less than 50% that of the baseline. The superscalar, out-of-order ARM core is capable of exploiting instruction level parallelism to a good extent and also has a high performance on-chip cache. The additional parallelism exploited by the HLS flow is evidently not enough to compensate for the clock frequency advantage the hard processor core has over the programmable logic and the longer data access latency from the reconfigurable array.

With our methodology, the accelerator networks generated are rather competitive against the hard processor, even without a reconfigurable cache. For Floyd-Warshall, knapsack and SpMV multiply, when the networks are directly connected to the memory through the ACP, the average performance is 2.3x that of the baseline—an 8.4x gain over the conventional accelerators. With the addition of caches, the average runtime of the accelerator networks was reduced by 18.7%, while that of the conventional accelerators was cut by 45.4%. The gap between their performance is thereby reduced from 8.4x to 5.6x. This difference in improvement is due to conventional accelerators' sensitivity to the latency of data accesses, which is also manifested by its performance degradation of 40% when the uncached HP port is used instead of ACP.

Our partitioning algorithm also has its limitations, as demonstrated by its ineffectiveness in the depth first search benchmark. The kernel performs very little computing but lots of memory accesses. The use of a stack in DFS also creates a dependence cycle through the memory. Consequently, the performance is fundamentally limited by the latency of memory access. Thus there were only small differences between the performance of the conventional accelerator and the accelerator network, both of which achieve throughput far below that of the baseline.

Overall, for kernels suitable for FPGA acceleration, there is a significant performance advantage in the accelerator networks as our algorithm effectively addresses the weakness of conventional accelerators. If we compare the best results achieved with the accelerator networks to that of the conventional accelerators, we see improvement of 3.3 to 9.1 times, with an average of 5.6.

6.2 Area comparison

To quantify the impact of our proposed methodology on area, we have compared the FPGA resouce usage of conventional accelerators and the accelerator networks. Table 2 shows the results, where each implementation is complemented with two memory subsystem configurations.

The difference in area between the accelerator network and the conventional accelerators is effected by two factors. The accelerator networks always have additional costs associated with the FIFOs and communication primitives. Also, we have sized the FIFOs to be larger than required to intro-

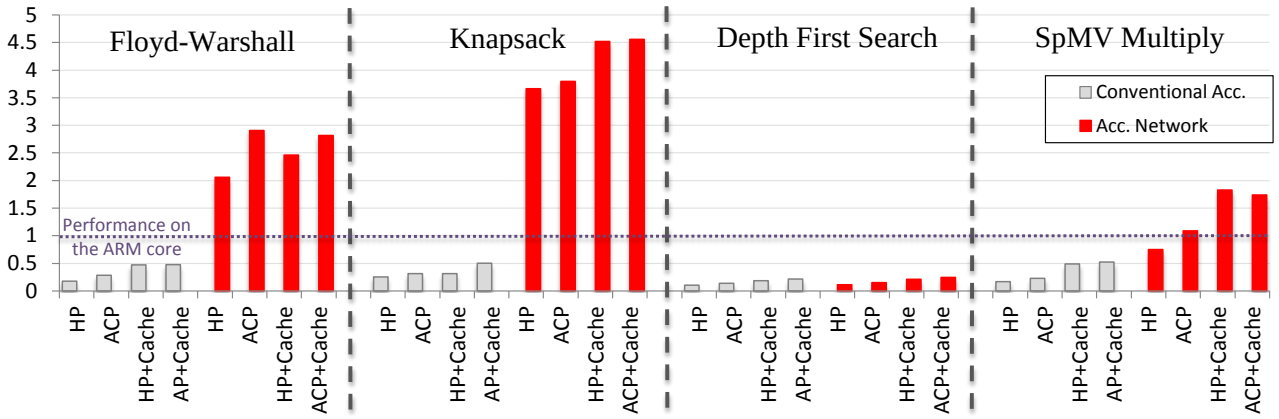

Figure 6: Performance of Conventional Accelerators and Accelerator Networks

Table 2: Resource Usage of Accelerators.

Benchmark		ACP			ACP + 64KB Cache		
		LUT	FFs	BRAM	LUT	FFs	BRAM
Floyd-Warshall	Con.Acc	2491	3528	0	3806	4629	19
	Acc. Network	7659	7210	0	8995	8309	19
	% change	+207.5	+104.3	0	+104.4	+79.5	0
Knapsack	Con.Acc	7672	7490	8	6573	5885	21
	Acc. Network	8089	8787	8	6970	7256	21
	% change	+5.4	+17.3	0	+6.0	+23.3	0
DFS	Con.Acc	4810	4929	4	4931	4594	21
	Acc. Network	8509	7813	4	7436	6298	21
	% change	+76.9	+58.5	0	+50.8	+37.1	0
SpMV Multiply	Con.Acc	9873	9116	10	7918	6792	21
	Acc. Network	8577	8837	10	6718	6788	21
	% change	-13.1	-3.1	0	-15.2	-0.1	0

duce more elasticity into the network. On the other hand, the accelerator generated for each individual process in the network has reduced pipeline depth, resulting in area savings. The overall change therefore depends on which factor plays a larger role, and is ultimately application specific.

7. CONCLUSIONS

In this paper, we presented a framework with which sequential programs can be transformed to networks of processes. With HLS, they can be implemented as accelerator networks on FPGA. Compare to accelerators directly synthesized from the original program, and software running on a hard CPU core with much higher clock frequency, our implementations can achieve significantly better performance.

8. ACKNOWLEDGMENTS

This research is supported by the Berkeley Wireless Research Center and the ASPIRE Lab. The ASPIRE Lab is funded by DARPA Award HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and industrial sponsors and affiliates: Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

9. REFERENCES

[1] Xilinx Inc., *Zynq-7000 All Programmable SoC Overview*, 2013.

[2] G. Inggs, S. Fleming, D. Thomas, and W. Luk, "Is high level synthesis ready for business? a computational finance case study," in *FPT*, pp. 12–19, Dec 2014.

[3] K. Gilles, "The semantics of a simple language for parallel programming," *In Information Processing*, vol. 74, pp. 471–475, 1974.

[4] T. Harriss, R. L. Walke, B. Kienhuis, and E. F. Deprettere, "Compilation from matlab to process networks realized in fpga.," *Design Autom. for Emb. Sys.*, vol. 7, no. 4, pp. 385–403, 2002.

[5] S. van Haastregt and B. Kienhuis, "Automated synthesis of streaming c applications to process networks in hardware," DATE '09, pp. 890–893, 2009.

[6] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf, "Yapi: Application modeling for signal processing systems," in *37th DAC*, pp. 402–405, 2000.

[7] T. M. Parks, *Bounded scheduling of process networks*. PhD thesis, UC Berkeley, California, 1995.

[8] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," ESOP'03, pp. 319–334, 2003.

[9] G. E. Allen, P. E. Zucknick, and B. L. Evans, "A distributed deadlock detection and resolution algorithm for process networks," in *ICASSP '07*, vol. 2, pp. II–33–II–36, April 2007.

[10] Xilinx Inc., *Vivado Design Suite High-level Synthesis*, 2015.

[11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, pp. 24:1–24:27, Sept. 2013.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," CGO '04, pp. 75–, 2004.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," MICRO 38, pp. 105–118, 2005.

[14] ILOG, Inc, "CPLEX: High-performance software for mathematical programming and optimization."

[15] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," 1994.

[16] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.