Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware

Zhaoshi Li[1] Leibo Liu[1*] Yangdong Deng[2] Shouyi Yin[1] Yao Wang[1] Shaojun Wei[1]
[1]National Laboratory for Information Science and Technology, Tsinghua University
[2]School of Software, Tsinghua University
li-zs12@mails.tsinghua.edu.cn,{liulb,dengyd,yinsy,wangyao_alucard,wsj}@tsinghua.edu.cn

ABSTRACT

CPU-FPGA heterogeneous platforms offer a promising solution for high-performance and energy-efficient computing systems by providing specialized accelerators with post-silicon reconfigurability. To unleash the power of FPGA, however, the programmability gap has to be filled so that applications specified in high-level programming languages can be efficiently mapped and scheduled on FPGA. The above problem is even more challenging for irregular applications, in which the execution dependency can only be determined at run time. Thus over-serialized accelerators are generated from existing works that rely on compile time analysis to schedule the computation.

In this work, we propose a comprehensive software-hardware co-design framework, which captures parallelism in irregular applications and aggressively schedules pipelined execution on reconfigurable platform. Based on an inherently parallel abstraction packaging parallelism for runtime schedule, our framework significantly differs from existing works that tend to schedule executions at compile time. An irregular application is formulated as a set of tasks with their dependencies specified as rules describing the conditions under which a subset of tasks can be executed concurrently. Then datapaths on FPGA will be generated by transforming applications in the formulation into task pipelines orchestrated by evaluating rules at runtime, which could exploit fine-grained pipeline parallelism as handcrafted accelerators do.

An evaluation shows that this framework is able to produce datapath with its quality close to handcrafted designs. Experiments show that generated accelerators are dramatically more efficient than those created by current high-level synthesis tools. Meanwhile, accelerators generated for a set of irregular applications attain 0.5x∼1.9x performance compared to equivalent software implementations we selected on a server-grade 10-core processor, with the memory subsystem remaining as the bottleneck.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080228

CCS CONCEPTS

• **Hardware** → **Hardware accelerators**; • **Computing methodologies** → *Parallel programming languages*; • **Computer systems organization** → *Reconfigurable computing*;

KEYWORDS

FPGA, Hardware Accelerator, Parallel Programming

ACM Reference format:
Zhaoshi Li[1] Leibo Liu[1] Yangdong Deng[2] Shouyi Yin[1] Yao Wang[1] Shaojun Wei[1] [1]National Laboratory for Information Science and Technology, Tsinghua University [2]School of Software, Tsinghua University . 2017. Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 12 pages.
https://doi.org/10.1145/3079856.3080228

1 INTRODUCTION

CPU-FPGA heterogeneous architectures have shown great potential by providing specialized accelerators with post-silicon reconfigurability. Different from CPUs which are organized in a centralized fashion, FPGAs are built as a large number of reprogrammable logics and distributed memories to provide specializations. Specialized FPGA accelerators have potential for significant performance improvement and energy saving. Such advantages inspire the industry and academia to put efforts on developing CPU-FPGA heterogeneous architectures for emerging data driven applications. Microsoft [43], Baidu [25] and IBM [12] are integrating FPGA into their data center. The acquisition of Altera by Intel and announcement of the HARP (Intel-Altera Heterogeneous Architecture Research Platform) [2] suggests that FPGAs will become a basic building blocks for future systems. As a result, a large body of research has been dedicated to building FPGA based accelerators for applications from a wide range of domains ranging from graph analytics [16, 37, 52] to sparse linear algebra [20], and etc.

Although specialization and heterogeneity are wide-spreading, the lack of efficient high-level programming model remains to be the major stumbling block for software developers to implement applications on FPGAs [7]. Traditionally, FPGAs are programmed with hardware description languages (HDLs) that are based on a hardware-centered abstraction and thus in terms of expressiveness and flexibility. Recent developments of high-level synthesis (HLS) focus on leveraging software programming languages, such as C [50], Java [6], OpenCL [3] and Scala [29, 42], to raise the level of abstraction. A common problem of existing works is that the parallelism in an application is extracted at compile time. Thus

*Corresponding author: Leibo Liu (liulb@tsinghua.edu.cn)

current high-level FPGA programming models are only applicable to regular applications with relatively small amount of control flow and structured memory access pattern.

Irregular computing patterns, which are prevalent in emergent applications, such as graph analytics [46] and computer graphics [24], are based on data structures with poor locality or statically unpredictable control flows. Today an efficient mapping of such applications to FPGA can only be established by manually orchestrating the inherent parallelism and capturing the design in HDL. For example, dedicated preprocessing logic for sparse matrices are designed to analyze the input data and parallelize the respective computing process [20]; applications organized around pointer-based data structures such as graphs usually require ad hoc on-chip storage designs [16, 37]. They are considered irregular either because they are organized around irregular data structures with poor locality, or they have irregular control flow unpredictable at compile-time. The programmability challenge can be even more serious in warehouse-scale and distributed systems. For instance, an implementation of Bing's ranking engine on a fabric of 1632 servers equipped with FPGAs incurred extensive hand-coding and debugging in Verilog HDL [43]. It is thus necessary to develop a generalized high-level FPGA programming model, which at the same time enables efficient hardware implementation, for irregular applications.

The crux of this programmability problem is the mismatch between FPGA execution model and current high-level programming models. FPGAs are inherently parallel with a massive number of computing elements that are running simultaneously. Low-level HDL matches this execution model by providing an abstraction in which multiple processes operates in parallel [34]. Modern accelerator designs captured in such a low-level abstraction tend to consist of a huge number of concurrent processes, whose complexity makes it intractable to manage and debug. High-level languages such as C/C++, nevertheless, largely follow a sequential programming model that may over-restrain the parallelism and hence the compile-time analysis of HLS tools may fail to find efficient FPGA designs [50]. An alternative solution is to explicitly express the parallelism with concurrent languages [3, 29, 39] and/or annotations [15]. These techniques remain unable to effectively synthesize irregular applications whose parallelism can only be determined at run-time [32] and can only generate over-serialized FPGA implementations, which can be less efficient than their CPU equivalents [51].

An investigation into handcrafted accelerators of irregular application on FPGA, which will be detailed in section 2, reveals that they outperform synthesized accelerators by exploiting fine-grained pipeline parallelism to meet the parallel execution model of FPGAs. Datapaths in these designs are pipelines augmented with data forwarding and squashing to orchestrate the pipeline execution at a finer granularity, which is dramatically different from the pipelines generated by HLS. The fine-grained pipeline parallelism bears similarity to the aggressive parallelization paradigm [41], which is a family of software techniques to parallelize irregular applications by leveraging runtime information. With such an approach, the runtime system aggressively assigns tasks to workers for parallel execution and at the same time dynamically watch all workers to ensure the execution correctness. This work is inspired by the similarity and adopt the aggressive parallelization paradigm to generate high-quality datapath designs on FPGA.

This paper introduces a comprehensive software-hardware co-designed framework to effectively program and implement irregular applications on FPGAs. The proposed techniques are problem-independent and can be integrated in to high-level synthesis flow so that applications developers could build efficient FPGA implementations for irregular applications without the need for hardware knowledge.

- We propose an inherently parallel abstraction which enables efficient extraction of fine-grained parallelism in irregular applications. Although aggressive parallelization has been extensively exploited in the software community, it is typically built upon synchronization primitives that are hard to realize on FPGAs. Our framework instead abstracts aggressive parallelization as tasks and rules that can be synthesized into pipelines and on-chip schedulers.
- We develop a synthesis flow that generates accelerators on FPGA from aggressively parallelized applications. To integrate our flow with existing HLS tools, we introduce dataflow graphs as the intermediate representation (IR) between abstractions and FPGA implementations. Parameterized templates for each primitive operations in IR are devised to map and scale up the design on FPGAs.
- We evaluate the proposed techniques with representative irregular applications. Our evaluation shows that the proposed high-level abstraction enables programmers to design datapaths with their quality similar to handcrafted designs. We also evaluate the performance of generated accelerators on Intel HARP [4], an emerging CPU-FPGA heterogeneous platform.

The paper is structured as follows. Section 2 introduces a motivational example to explain how the aggressive parallelization paradigm can be used to find efficient FPGA implementations for irregular applications. Section 3 provides an overview of the proposed framework. Section 4 presents the formulation of the framework. Section 5 introduces details on FPGA implementation. The evaluations are shown in Figure 6. Then section 7 introduces related works and section 8 concludes the paper.

2 MOTIVATIONAL ANALYSIS

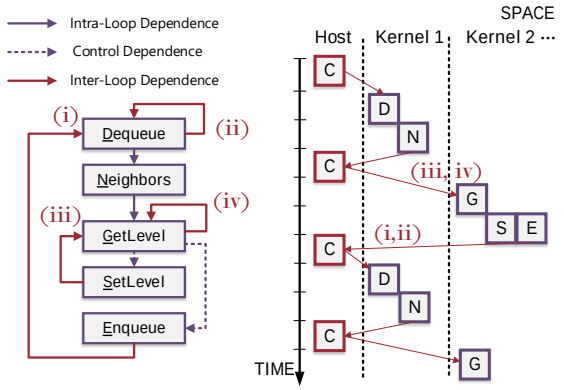
In this section, we use a typical irregular application, breadth-first search (BFS) to demonstrate the motivation for this research. Section 2.1 sketches the BFS algorithms and identifies the sources of irregularity. Section 2.2 shows the limitations of current high-level programming model by synthesizing BFS coded in OpenCL. Section 2.3 examines the gap between handcrafted accelerator and synthesized ones for irregular applications. Section 2.4 explains two aggressively parallelization schemes, which provide key clues for the design abstraction proposed in this work.

2.1 Irregularity of BFS

Figure 1 (a) shows the sequential version of BFS algorithm. BFS traverses a graph and labels each vertex `v` such that `v.level` is the number of edges on the shortest path from `root` to `v`. A task `Visit`

```
1. struct Visit {Vertex vertex; int level;};
2.
3. Queue<Visit> Q; // FIFO for new Visits
4. Level[root] = 0; // Initial value for root
5. Q.push(Visit(root, 1));
6. for (Visit t : Q){                          // Dequeue
7.   for (Vertex v : G.neighbors(t.vertex)){  // Neighbors
8.     if(Level[v] == INF){     // not visited.  GetLevel
9.       Level[v] = t.level;                  // SetLevel
10.      Q.push(Visit(v, t.level+1);          // Enqueue
11. }}}
```

(a) Source code of sequential BFS



(b) Program dependency graph of sequential BFS

(c) Schedule diagram inferred from AOCL-BFS

Figure 1: Analysis and schedule of BFS

abstracts the visit to a vertex. It is characterized by two components, vertex_id `vertex` and `level`. A workset of `Visit` is maintained in first-in-first-out (FIFO) order (line 3). This workset can be implemented as a queue in sequential program. Initially, all levels of nodes are set to `INF`, and `Visit(root, 1)` is pushed into workset (line 5). Each iteration (line 6) removes a `Visit t` and visit all neighbors of its vertex (line 7). Upon reaching a node that has not been visited before (line 8), its `level` is set to `t.level` (line 9) and new `Visit(v, t.level+1)` is added to workset (line 10). A compile-time parallelization tool would be able to find the concurrency in the inner loop (line 7) since all iterations on neighbors of a given vertex have no conflict in memory references.

A program dependence graph (PDG) can be generated (figure 1 (b)) for the nested loop (line 6-11). Each node represents the computation performed atomically on a task `Visit`. Each directed edge represents a dependence.

The irregularity of BFS is reflected in two aspects. First, tasks are created dynamically, making static scheduling infeasible. For BFS, tasks created by the outer loop are dependent on the connectivity of each node (Dependence i & ii in figure 1 (b)), and workloads induced by the inner loop are imbalanced depending on the degree of the nodes at each level (Dependence iv). Second, dynamic access patterns to shared memory locations introduces unpredictable dependences (Dependence iii). In case of BFS execution flow of the outer loop is serialized by synchronizations which ensure consistent view of memory among iterations. In fact, these two characters are prevalent in irregular applications across multiple domains. For example, vertex based programming frameworks [46] for graph analytics feature iteratively modifying active vertices stored in a sparse data structure until there is no more active vertex.

2.2 High-Level Synthesis of BFS

To evaluate current high-level language tools for FPGA, BFS coded in OpenCL [31] is synthesized by Altera OpenCL (AOCL) SDK [3]. Since generated hardware description is not readable, a schedule diagram shown in figure 1 (c) is constructed based on the user-guide [4] and profiles from execution. We will use "D" and "E" to denote choosing an `Visit` and committing an `Visit` respectively. In AOCL-BFS, irregularity is handled by a host CPU interacting with the FPGA through board-level interconnection. Two kernels are generated: kernel 1 checks whether neighbors of a vertex has been visited, and set them to be visited if not; kernel 2 check whether a vertex is to be updated and update it if true. Threads in each kernel streams through hardware pipelines synthesized on FPGAs. Then host invokes kernel 1 and kernel 2 iteratively until kernel 2 finds no more vertex is to be visited. By introducing a host, all inter-loop dependences in figure 1 (b) are resolved.

Apparently such schedule cannot fully take advantage of the computation resources. Execution flow is over-serialized. Barriers are inserted at the end of each kernel to ensure no collision occurs. As a result, newly created tasks have to be stored back in memory and retrieved in the next round.

2.3 A Comparison between Synthesized and Handcrafted Designs

A number of handcrafted accelerators with better performance than both multi-core counterparts and synthesized ones is reported in the literature [16, 37, 48]. A detailed examination of these designs suggest they share some constructs that cannot be expressed by current high-level language tools based on multi-threaded programming for FPGA.

The major difference between synthesized accelerator for irregular applications and handcrafted ones is how dynamically created tasks are handled. Synthesized accelerators usually leverage a centralized control unit to collect and dispatch tasks throughout the FPGA, which incurs huge overheads as shown in the case of OpenCL-BFS. On the contrary, handcrafted accelerators assign tasks following the spirit of dataflow execution [38], eliminating overheads brought by a centralized controller. This difference is resulted from the limitation of high-level abstractions. However, a robust implementation of a software thread pool requires intensive tuning of synchronization primitives, let alone customizing it for specific applications. On the other hand, a customized hardware implementation of a task pool can be simply realized with a multi-bank double-ended queue (or FIFO) and an arbitrator to distribute tasks among pipelines. In fact, similar structures are commonly seen in almost all handcrafted accelerators for irregular applications.

Another difference lies in how to handle irregular memory access patterns. In multi-threaded programs conflicts can be resolved by using conditional variables or semaphores, whose semantics require a large amount of hardware on FPGA. Thus current HLS has to use barriers or sequential semantics to serialize tasks with potential collisions. In contrast, handcrafted accelerators exploit fine-grained

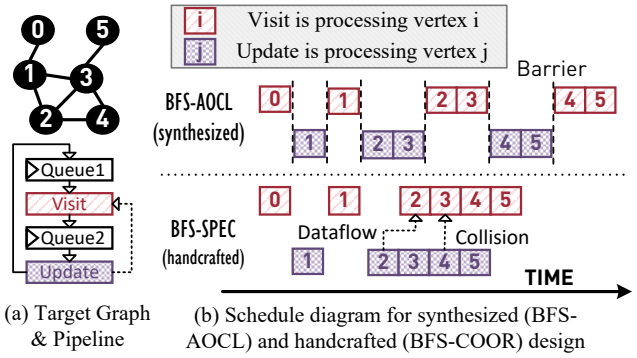


Figure 2: Applying synthesized and handcrafted pipeline to a simple graph

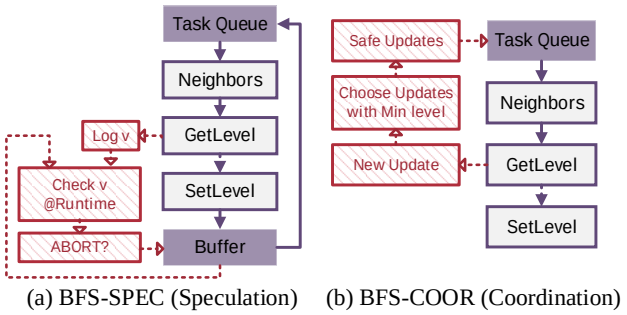


Figure 3: Program dependence graph (PDG) after parallelizing BFS with (a) speculation and (b) coordination.

pipeline parallelism to build problem-specific structures. For example, in [48] conflicts to the `level` array of BFS are detected by comparing input addresses in pipelines to ready-to-commit addresses stored in on-chip BRAMs.

With the two aforementioned differences, handcrafted designs achieve considerably higher performance by exploiting fine-grained pipeline parallelism. To illustrate this notion, figure 2 compares the pipelines of synthesized and handcrafted designs. The 5-stage PDG from figure 1 (b) is reduced to a 2-stage pipeline, with the first stage visiting all neighbors of a vertex and the second stage updating them. Each stage gets tasks from one queue and put new tasks to the other queue. Different control mechanisms are applied in two designs. As a result, different runtime schedule diagram (figure 2 (b)) is derived when the target graph is traversed. In synthesized designs, these two stages execute alternatively with barriers inserted, whereas handcrafted ones features dataflow execution that a stage is invoked as soon as new tokens arrives in the input buffer. Memory collisions are resolved in handcrafted designs by forwarding pipeline contents: when visiting vertex 3, the Visit stage will try to visit its neighbor vertex 4; but vertex 4 (as neighbor of vertex 2) is being updated by the Update stage. In the handcrafted pipeline states of the stage Update is visible to the stage Visit, eliminating potential collisions caused by shared memory locations.

In addition, problem-specific on-chip memory hierarchies are designed in handcrafted accelerators. Although memory bandwidth and latency are decisive in accelerator performance, our framework focus on devising an abstraction to expose parallelism. We will use problem-independent memory optimizations in our FPGA implementations.

2.4 Aggressive Parallelization

Aggressively parallelizing irregular applications has been studied extensively in recent years. In contrast to conservatively parallelize regular applications at compile time, these parallelization approaches make aggressive assumptions about input data and execution flow, and use the runtime systems to ensure these assumptions to be hold during execution. For example, thread-level speculation (TLS) [18, 30] exploits parallelism in loops with statically unpredictable dependences by executing each iteration speculatively and detecting dependence violations by the runtime system. Software Transactional Memory (STM) [45, 47] parallelizes unpredictable shared data accesses for irregular data structures with runtime conflict detection and recovery. Analysis [41] of irregular algorithms reveals that such parallelism is ubiquitous. To exploit such parallelism, programmers or compilers specify the conditions under which tasks can be executed concurrently. Based on whether conflicts are allowed in concurrent tasks, runtime parallelization can be labeled as either speculative or coordinative. In speculative approaches, the runtime system schedules concurrent execution regardless of potential conflicts, and recovers from faults once these conditions are violated; whereas in coordinative parallelization, the runtime system ensures that only non-conflicting tasks are scheduled for simultaneous execution.

A speculative parallelization strategy of BFS (BFS-SPEC) is shown in Figure 3 (a). At compile time it is asserted that no memory conflict occurs, i.e. dependence **iii** in Figure 1 (b) doesn't exist for issued tasks. Then memory references of `getLevel` and the committing `setLevel` are compared at runtime. If a collision is detected, i.e. `level` loaded by this iteration has been modified by an earlier one, result of `setLevel` for this iteration is aborted by the runtime system.

For coordinative parallelization (BFS-COOR), it is observed that `Visits` with minimum `level` can be executed simultaneously, because when two `Visits` visits the same neighbor, that vertex always gets the same `level` regardless of the execution flow [35]. The runtime system schedules all `Visits` with minimum `level` to execute in a level-by-level fashion, as shown in Figure 3 (b).

These two approaches share similar features with the handcrafted accelerators. First, workloads are pushed in a queue and are executed as soon as resources (threads or processes) are available. Second, correctness is guaranteed by inspecting run-time states. In fact, if workloads are discarded as soon as collisions are detected, the schedule diagram of BFS-SPEC for the target graph in figure 2 (a) is the same with that of handcrafted design in figure 2 (b); whereas the schedule diagram of BFS-COOR is similar to that of synthesized design except that workloads are scheduled autonomously without barriers.

Our framework strives for devising an abstraction in reference to aggressive parallelization, which will expose fine-grained pipeline

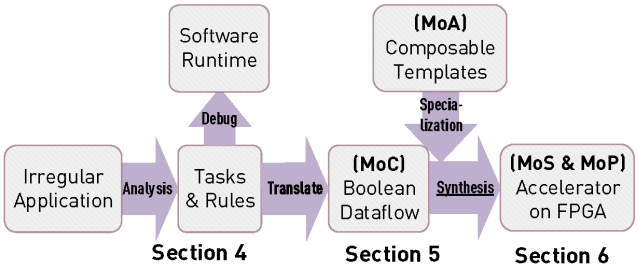


Figure 4: Design flow from specification to implementation

parallelism tailored for FPGA implementation. Run-time collaboration in aggressive parallelization will be expressed as rules that will be evaluated using states of the runtime system during execution. Using these rules data forwarding and squashing can be specified in this abstraction.

3 SYSTEM DESIGN METHODOLOGY

Figure 4 presents design flow of our software-hardware co-design framework. We follow the generic synthesis framework [19] that derives *implementations* (How to do) from *specifications* (What to do) so that our flow could be easily integrated into existing tools. The synthesis process can be denoted as **MoC** (Model of Computation) + **MoA** (Model of Architecture) = **MoS** (Model of Structure) + **MoP** (Model of Performance).

- **MoC** represents how to express applications. We provide an abstraction which specify irregular applications as a set of well-ordered tasks whose dependencies are described as rules. Tasks will be aggressively parallelized with a runtime scheduler ensuring correctness by evaluating rules. Programmers' specification in this abstraction could be automatically translated into Boolean Dataflow Graph (BDFG) [10], an MoC well-matched for FPGAs. Besides, a pure software runtime is provided to help programmers debug applications.
- **MoA** provides constraints of target platform. To capture architectural features of FPGA, we provide parameterized templates for each components in BDFG as the MoA. These templates will be used as building block to construct datapaths on FPGA. Currently, some parameters have to be tuned by programmer to achieve better performance.
- To evaluate the accelerator, we compare datapath from this framework to handcrafted ones (**MoS**). Also combined with a problem-independent memory system, generated datapath has been mapped on Intel HARP platform to evaluate its performance (**MoP**).

A featured aspect of our framework is the orthogonalization of concerns [28]. Mainstream high-level abstractions for FPGA blurred *specification* with *implementation*. They are built upon the imperative programming paradigm that a program executes as it is described. As a result, dynamically recurring workloads are expressed as either while-loops or thread blocks separated by barriers, precluding parallel executions in either case. And unpredictable dependences are described and fulfilled by synchronization primitives such as barriers and locks derived from multi-threaded programming, which consume a large amount of resources on FPGA. In our abstraction, dynamically recurring workloads and unpredictable dependences are described as what-it-is and leave the implementation details to the system. If targeted platform is an FPGA, implementation based on dataflow execution is available to exploit the fine-grained parallelism.

In summary, our framework enables programmers to express parallelism in irregular applications without worrying about implementation details. Though human intervention is required for high-level decisions such as parallelization strategies and parameter selections, the amount of design effort paid in our flow is similar to mainstream parallelization frameworks, e.g. OpenCL.

4 ABSTRACTION FOR SPECIFICATION OF IRREGULAR APPLICATIONS

In this section, we formalize the process of aggressive parallelization and derive a respective high-level abstraction to exploit runtime information.

4.1 A Formalization of the Sequential Execution Model of Irregular Applications

In this subsection, we abstract irregular applications as sets of tasks and present the norm for correct parallelization following standard software approaches.

Typically an irregular application is built around loop constructs. The sequential execution process iterates the loops and continuously inserts newly created tasks into queues. Following the taxonomy of compiler, we classify loop constructs into **for-all** and **for-each** loops (corresponding to `do-all` and `do-across`, respectively, in [27]) according to how the ordering among tasks are managed.

- **for-all** loop: all loop iterations can be executed in parallel. In sequential semantics an arbitrary iteration is chosen to be executed one by one.
- **for-each** loop: synchronizations are needed to ensure later iterations get the correct value from earlier ones. In sequential semantics synchronizations are achieved by executing iterations in serial.

Each iteration is abstracted as a task t, and new tasks can be created during the execution of a task. Given an application, let Σ be the domain of program states (i.e. memory locations and their values), and let T be the domain of tasks.

Definition 4.1. A **task** t is a partial function from states to states and tasks, $t : \Sigma \to \Sigma \times T$.

For the sequential BFS in figure 1, there are two sets of tasks, the `Visit` from the outer for-each loop (line 6), and the `Update` from the inner for-all loop (line 7). Here the loop construct of `Update` is nested in that of `Visit`, which will be discussed later. Tasks with the same function (i.e. loop body) are gathered into a **task set**, which can be categorized as either a **for-all task set** or a **for-each task set** based on which loop construct is used to iterate it. A task domain for an applications is the conjunction of all task sets.

Definition 4.2. A task is **active** if it is ready to be executed. All active tasks with the same function form a **active task set**.

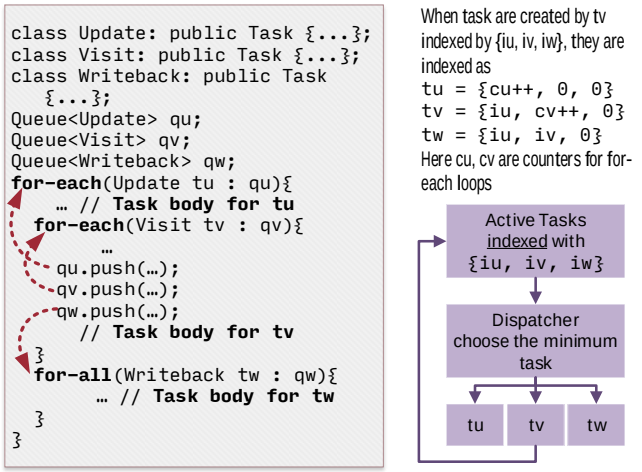


Figure 5: Indexing tasks from a nested loop to maintain well-order

Given an application and an execution flow, the active task set is always a subset of the task set.

A well-order is defined on the task domain based on the type of the respective loop construct and the sequence of activation. For a single loop, each task is indexed with a non-negative integar to denote its order in the activating sequence. In for-each task set, $t_i, i \in \mathbb{N}$ denotes the i^{th} activated task; whereas in for-all task sets, since the sequence of task activation doesn't influence execution order, all tasks are labels with 0 so that they have the same order.

Definition 4.3. **Sequential execution** of an application is that given an initial active task t_0, choosing the minimum task t_{min} in all active tasks and applying t_{min} to Σ iteratively until there is no more active task.

Given juxtaposed or nested M loops, a well-order on all task sets in the task domain is maintained by indexing each task with a M-tuple $\{\mathbb{N} \times \mathbb{N} \times \cdots\}$, where each $\mathbb{N}$ corresponds the index of one loop. Loops are arranged from left to right in the order they appear in the program, with left ones having higher weight than right ones in the well-order. Indexes from preceding loops are inherited by rear loops, otherwise indexes are discarded. In this way, the sequential execution of juxtaposed or nested loops conform to that of a single loop.

For example, in figure 5, a for-each `Visit` loop (task tv) and a for-all `Writeback` loop (task tw) are nested in the outer for-each `Update` loop (task tu). Task body of `Visit` actives (i.e. push in queues) three types of tasks. Tasks are indexed by a 3-tuple $\{iu, iv, iw\}$, with $iu, iv, iw \in \mathbb{N}$. With the indexing scheme for 3 types of activated tasks in figure 5, the sequential execution conforms to Definition 3.

As we established, an irregular application is abstracted as a collection of well-ordered task sets and a sequential program is the iterative execution of the minimum active tasks. A parallelized execution is **correct** if its result is equivalent to those of the sequential executions.

4.2 Specification of Aggressive Parallelization

In this sub-section, we show a novel abstraction for aggressive parallelization and its formalization. This abstraction enable programmers to answer two questions regarding irregular applications: (1) where is the unpredictable dependence? (2) How to detect and resolve this dependence by leveraging runtime information?

As discussed before, irregular applications feature dependences that cannot be resolved at compile-time. As a result, if multiple tasks in the active task set are to be executed, conflicts among them may introduce errors in the program state. To avoid errors, conservative implementations with only compile-time analysis tend to be over-serialized. With aggressive parallelization techniques, the parallel executions of tasks are scheduled at runtime, during which dependences among tasks are resolved given the input data and a particular execution flow.

Depending on whether conflicted tasks are allowed to be executed simultaneously, the aggressive parallelization techniques can be grouped into two categories as follows.

- **Speculative parallelization**: Multiple tasks are executed regardless of their conflicts and each task inspects collisions with other tasks at runtime. For example in speculative BFS, a task `Visit` should be *aborted if* an earlier task `Visi` modifies the same vertex *when* a write is committing;
- **Coordinative parallelization**: A runtime system ensures that only non-conflict tasks are activated. For example in coordinative BFS, a newly-created task `Update` by a task `Visit` should be *pushed* into queue *when* task `Visit` has the minimum level across all the waiting tasks.

4.2.1 Formalized Aggressive Parallelization. Given a collection of well-ordered task sets, an aggressive parallelization is the process of scheduling a subset of tasks to be executed concurrently, with each task inspecting all running tasks at runtime to ensure that no dependences are violated. Each inspection of dependence can be abstracted as a **rule**.

Definition 4.4. A **rule** is a *promise* to return a value to its creator in the *future*, when its creator reaches a planned rendezvous. The returned value is a function of all states of the runtime system spanning from the creation of the rule to the return point.

A task can create a rule and define a respective rendezvous its task body. When the task reaches the rendezvous, it will stall and wait for the return value of the corresponding task.

In the speculative BFS example, the task `Update` creates a rule that checks all writes to `Level[v]` by tasks **earlier** than itself, with a rendezvous before this task committing its result. If any conflict is detected by the rule, this task aborts its result; **otherwise** it commits.

Two issues are critical in the speculative BFS example. (1) By "**earlier**" the well-ordering of tasks is exposed by the rules. In practice, all implementations feature rules created by parent tasks with their ordering indexed as a parameter in rule constructor. (2) By "**otherwise**" exit paths are provided in rules.

In fact these exit paths are the key to ensure *liveliness*. If a rule does not return a value upon an unfinished inspection, a task will always wait at rendezvous. When limited resources are provided in implementation, e.g. the number of rules under inspection, a deadlock will occur. Thus, each rule must contain an exit path that

is triggered automatically when its parent task is the minimum one among all waiting tasks at the planned rendezvous. In practice, we found this design extremely powerful in both ensuring liveliness and specifying applications.

4.2.2 Event-Condition-Action Rule Grammar. Potential dependences among tasks have been exposed by specifying rules in tasks. Then programmers could design ad-hoc strategies for detecting and resolving unpredictable dependences at runtime in a Event-Condition-Action (ECA) rule grammar.

Inspecting these rules for the BFS example, it is straightforward to find a pattern that can be generalized as the following form: a task should **do** an *action* **if** a particular *condition* is met **when** an *event* happens. Such a generalization is designated as ECA grammar [23].

An ECA rule grammar in our framework follows traditional ECA clause: **ON** event **IF** conditions **DO** actions. Rules are created by tasks at runtime, with each rule corresponding to a parent task. The semantics of rules are simple: a rule is triggered by an event; then it examines the conditions and fires an action if the condition is met. However, syntax of each lexicon is tailored to fit in aggressive parallelization.

- Events are limited to activation of tasks, or tasks reaching specific operations in task bodies, or combinations of these two types of events. By signal an event, index and data fields of the triggering task are broadcast to all rules.
- Condition are boolean expressions composed of index and data fields in triggering events, and parameters forwarded by parent tasks when creating this rule.
- Actions are limited to return a boolean value to steer task tokens in the task body of the parent task.

Besides, **otherwise** clause is obligatory for every rule to ensure liveliness. Thus, a rule is composed of an constructor for creation in tasks, any number of ECA clauses, and an otherwise clause.

Even with limitations, the expressiveness of ECA rule in our framework is not compromised. In general, an aggressively parallelized task should react to states of concurrent tasks. In our abstraction, all running tasks with changed states could be captured by events. Then a task can choose its reaction by inspecting states of all running tasks. By steering task tokens, conditional branches are constructed in task body.

For speculative BFS, before loading `Level[v]`, task `Visit` tv_i creates a rule with `(index, &Level[v])` as parameter for its constructor, and plans a rendezvous before committing results. The grammar of this rule is: **ON** (any other rule tv_j trying to write `&Level[v]`), **IF** ($tv_i.index > tv_j.index$), **DO**(`return false`); **otherwise**, **DO**(`return true`). At rendezvous, the task aborts its result if the rule returns false, otherwise it commits.

For coordinative BFS, before trying to activate new `Update`, a task `Visit` creates a rule and plans a rendezvous on the spot. A rule will return true when the task is one of the minimum tasks at rendezvous. Then the task `Visit` will activate the new `Update` when its rule returns true.

4.3 Fine-grained Pipeline Parallelism

The fine-grained pipeline parallelism can be easily captured with our formulation.

- By collecting newly created tasks into active task sets, dependences induced by recurring tasks are eliminated. A pipeline could be derived at ease from task bodies with tasks streaming from active task sets.
- By transforming unpredictable dependencies into rules, tasks in execution could inspect any state in the pipeline under the command of programmers. In this way, architecture-level optimizations for pipelines, such as squashing, forwarding, etc. can be specified in our abstraction at a much higher level, as in the case of speculative BFS which squashes tasks with collisions.

4.4 Putting It All Together

Irregular applications have been abstracted as well-ordered tasks and a number of rules for aggressive parallelization. But we haven't touched the implementation of this abstraction. Possible implementations are listed here.

- A C++ implementation based on `std::thread`, `std::async` and `std::future` is provided for debugging. Programmers could debug their specification is a pure software environment with the help of multi-threading tools to detect memory collisions and race conditions.
- A thread pool and conditional variables can be used to implement in Pthread.
- As the definition of rule suggested, any language supporting asynchronous programming paradigms with futures and promises might be used.
- A FPGA implementation, which is our target, is presented in section 5.

An intriguing point to note is all software implementations rely on fine-grained synchronization to exploit fine-grained thread-level parallelism, which is not supported in current high-level language tools for FPGAs.

5 SYNTHESIZING IMPLEMENTATIONS FROM SPECIFICATIONS

In this section, we introduce the work flow from specification to FPGA implementation. The overall flow is shown in figure 4. Section 5.1 presents design considerations in choosing an MoC suitable for FPGA. The architectural features of FPGA is captured by **MoA** in the form of parameterized templates, whose details will be provided in section 5.2. On such a basis, the synthesis of irregular applications is reduced to specializations of **MoA** templates according to dataflow **MoC**.

5.1 From Specification to Dataflow MoC

As demonstrated in Figure 6, the bridge from software specification to hardware implementation is the dataflow model of computation. A code snippet taken from the inner loop body of sequential BFS (Figure 1 (b)) is listed in (Figure 6 (a)), meanwhile the respective PDG is depicted in Figure 6 (b) with the control dependence represented as dash lines. The dataflow graph encodes the control dependences of a program into data dependences among actors to eliminate the need of centralized control units. Each actor fires an execution when its input operands are ready. In our framework we adopt BDFG

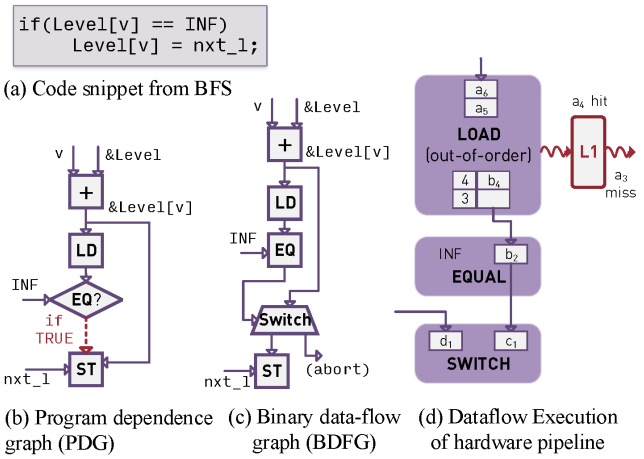


Figure 6: A dataflow approach to synthesis software specification into hardware implementation

MoC that provides switch actors (Figure 6 (c)) as IR. On spatial architectures such as FPGAs, dataflow graph can be implemented as a hardware pipeline (Figure 6 (d)) by properly embedding the graph into underlying fabric such that a computational unit corresponds to a node (actor) in the graph. Then parallel tasks represented as data tokens can flow through the pipeline as long as there is sufficient computing resource.

Currently specifications of rules are translated to BDFG model manually in a systematic manner as a modern compiler would do. Task bodies and condition-actions of rules are transformed into BDFG, with task queues (inferred from `for-each`/`for-all` constructs), rule constructors and rule rendezvous inserted as primitive operations in the graph. This flow is compatible with standard HLS flow.

5.2 MoA Based Templates for FPGA Implementations

Our framework leverages parameterized templates to ease the generation of accelerators. Four types of templates are provided: modules of primitive operations in pipelines, multi-bank task queues, rule engines, and a generic memory subsystem.

Figure 7 shows a generalized architecture for synthesized accelerators. The host processor initializes task queues and waits for the FPGA to finish. Task queues will pop tasks to pipelines in FIFO order and collect activated tasks in the pipelines. All pipelines share a rule engine to forward or squash task tokens in order to guarantee correctness.

Pipelines are generated by arranging modules of primitive operations according to the topology of BDFG. Multiple pipeline instances can be incrementally generated until the resource limit of targeted FPGA is reached. It must be noted that operations with unpredictable latency may severely degrade the utilization of hardware pipelines. On the current HARP platform, a direct read hit to the 64KB on-FPGA cache results in 70ns (or 14 FPGA cycles) latency [14]. And a cache miss could stall the memory operation for over 200ns. We adopt the dynamic dataflow approaches [5, 49] to reorder tasks at

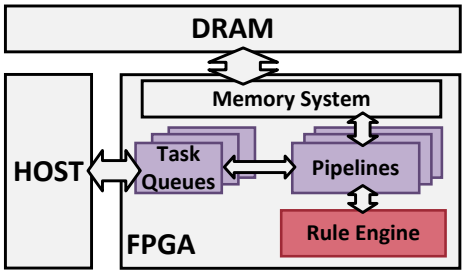


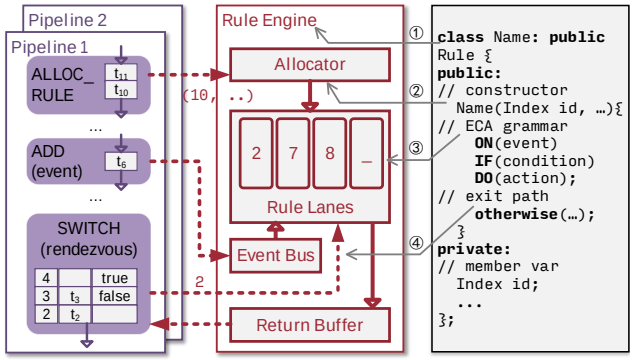
Figure 7: Generalized Architecture of Synthesized Accelerators

operations according to the availability of operands. As a result, blocked tasks can be bypassed. As showed in figure 6 (d) load operation issued new request a_4 regardless of the stalled a_3 due to cache miss. However, out-of-order operations incur resource overheads on FPGAs since they requires large matching logics. We limit out-of-order executions to load/store units and rendezvous in pipelines and keep other operations in-order so that their interfaces can be implemented frugally as dual-port FIFOs.

Separate task queues are designated for each active task set. Each task corresponds to one entry whose width can be inferred from the specification. A task is considered active when it is pushed into a task queue. Then tasks will be popped into pipelines in first-in-first-out order. An index indicating the well-order is assigned to each task when it is pushed using the scheme shown in figure 5. To interact with multiple pipelines, a multi-bank queue with customizable number of input/output ports is provided. A wavefront allocator [8] is used between input ports and pipelines to ensure load balance among banks. This design is equivalent to a thread pool for recurring tasks in software. Compared to previous software implementations which rely on either thread pools [44] or priority queues [33], our implementation is much more approachable on FPGAs.

Each type of rule is specified in the ECA grammar, and will be implemented as a rule engine on FPGA, as showed in figure 8 (c) ①. An AllocRule operation is inserted in task pipelines where rules will be constructed with the task indexes and other variables as parameters (figure 8 (c) ②). Then an allocator in the corresponding rule engine will allocate a lane for that index. If no lane is available, the parental task will be stalled. Lanes are constructed from ECA grammar as pipelines (figure 8 (c) ③). Events are captured by broadcasting tasks reaching a specific operation by an event bus. If a rule decides to return, it puts a return value in the return buffer and releases the lane. Rendezvous is planned in the task pipeline as a switch actor, which reorders the out-of-order return values and steers task tokens based on these values. Moreover, the minimum task index at this rendezvous across all pipelines is broadcast to rule lanes to trigger the **otherwise** clause in the rule (figure 8 (c) ④). The rule engines are equivalent to the runtime scheduler in software implementations of aggressive parallelizations.

As for the memory system, we have noticed that in handcrafted designs problem-specific optimizations based on underlying data structures are crucial for their performance. Since this paper focuses on programing the datapath of accelerators from high-level abstractions, we use on a generic cache design provided by HARP. In this way, the memory subsystem is kept problem-independent.



(a) Integration of task pipelines and rules (b) Architecture of the rule engines (c) Programmer's declaration of a rule

Figure 8: A rule engine and its connections with task pipelines

6 EVALUATION

In this section, we evaluate the quality of accelerators generated by our framework, by comparing them with handcrafted designs and aggressively parallelized applications from published works running on a state-of-the-art server processor in terms of structure and performance respectively. The target platform of our framework is Intel HARP [2], which consists of a processor and an Altera FPGA working as a cache-coherent peer. We further show the performance of the generated accelerators is bandwidth-bounded.

6.1 Benchmarks

We use the following benchmarks. We limit our referential designs to works related to aggressive parallelization. Performance reported in these works might be less than best-effort CPU performance(e.g. Graph500) since they may not fully exploit memory systems. It is a fair comparison as the focus of our framework is leveraging aggressive parallelization to generate datapaths.

The speculative BFS (SPEC-BFS) comes from Steffan et al. [33], while the coordinative BFS (COOR-BFS) comes from Leiserson et al. [35]. We evaluate the performance of BFS using a USA Road graph [1].

(**SPEC-SSSP**) Speculative single-source shorted path [21]. The idea of aggressively-parallelized SSSP is based on Bellman-Ford algorithm [13] that calculates the shortest distance to a root vertex for all vertices in the graph. Initial vertices in task queues are the neighbors of the root. Then each task updates a vertex with the minimum of current distance and distances induced from its neighbors. If a vertex has been updated, all its neighbors are pushed into task queue. A rule is defined so that the distance of committing vertices are broadcast to all running tasks to avoid data hazard.

(**SPEC-MST**) Speculative Kruskal's minimal spanning tree [9]. The idea of Kruskal's MST is to sort edges by their weights and add them one-by-one to a disjoint set. To parallelize this process, smallest k edges (tasks) are fired, under the rule that if the end point of a larger edge overlap with a smaller one, the larger one will be aborted (squashed).

(**SPEC-DMR**) Speculative Delaunay mesh refinement [33]. DMR is a process of iteratively re-triangulating bad triangle that overlap with other ones until no more bad triangle. Initially all bad triangles stores in task queues, or are pushed in queues incrementally from a host processor. A rule can be defined that if a bad triangle doesn't overlap with others anymore, its corresponding task is squashed.

(**COOR-LU**) Coordinative sparse LU factorization [22]. The underlying dense version is from Barcelona OpenMP Task Suit [17]. Initial tasks for updating different blocks of a sparse matrix are pushed incrementally into task queues from host. The order of updating each block is guaranteed by a set of rules that detect block collisions on the input sparse matrix at runtime.

If not explicitly stated, the input data for evaluation are the same as the original papers.

6.2 Structure of Accelerators

Handcrafted accelerators on FPGA for irregular applications emerged in the past a few years. Not much published works can be found, especially for large-scale applications where data must reside in off-chip storages. Comparing datapaths of synthesized accelerators whose structures are sketched in figure 7 to datapaths of problem-specific designs for BFS [16, 37, 48] and SSSP [52], we conclude that our framework could build datapaths similar to handcrafted accelerators.

The major difference between accelerators generated by us (figure 7) and handcrafted ones is the memory system. Handcrafted accelerators handle data in an aggressive manner by either preprocessing input data to increase locality [37] or dynamically prefetching data to on-FPGA memory [16]; whereas our accelerators use a problem-independent memory system.

Another difference is the rule engine for detecting and resolving dependences. Resource overheads of the rule engines are acceptable compared to task pipelines. Depending on applications rule engine takes 4.8∼10% of total registers in our design, most of which are consumed by the allocator and event bus. BRAMs and combinational logics are negligible when compared to task pipelines. Investing more resources gains little return as memory subsystem is the bottleneck.

6.3 Performance Evaluation

Performance evaluations are conducted on the Intel HARP platform. The accelerators are synthesized to an Altera Stratix V 5SGXEA7N1FC45 FPGA. Given an application, a number of parameters of architectural templates, e.g. the number of pipelines and the number of lanes in the rule engine, have to be customized. Currently we rely on a heuristic approach to ensure the resultant design to occupy the FPGA resource as much as possible to deliver the best performance. All accelerators are running at 200MHz. The software counterparts are evaluated on an Intel Xeon E5-2680 v2 with 10 cores running at 2.8GHz. Source codes are complied in GCC 4.9.2 with -O3 optimization switch. Both software and hardware implementations pre-load all input data to DRAM.

FIG 9 shows the speedup of synthesized accelerators over their software counterparts on Xeon. Our HARP based accelerators attain 2.3∼5.9x speedup over sequential implementations on one core, but cannot beat parallelized designs on 10 cores yet (with 0.5∼1.9x speedup). We consider the memory subsystem as the bottleneck, since the QPI-based shared memory provides ∼7.0GB/s bandwidth

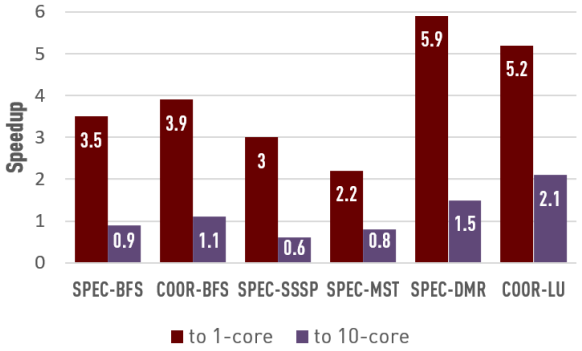


Figure 9: Speedup of synthesized accelerators on FPGA over their sequential (1-core) and parallel (10-core, 20-thread) counterparts on Xeon

for HARP FPGA [14], which is an order-of-magnitude lower than Xeon processor.

To predict the potential performance gains with improved memory system, we collected execution profiles from the HARP and developed a software emulator. Bandwidth of the QPI can be scaled in this emulator. Then speedup over baseline implementation on real HARP and pipeline utilization rates are plotted in figure 10. Here the pipeline utilization rate is calculated as the average number of active (neither stall nor idle) primitive operations throughout the execution over total number of primitive operations for all pipelines instantiated on FPGA.

As shown in figure 10, in most cases speedup and utilization rate are positively correlated to the available bandwidth. In the cases of SPEC-DMR and COOR-LU in which tasks are sent from host, a linear correlation can be observed. With the increasing bandwidth expected in commercial Xeon-FPGA platforms, our framework will enjoy a performance improvement.

Another point to notice is in applications with recurring tasks. If tasks are activated speculatively (as in the case of SPEC-BFS, SPEC-SSSP and SPEC-MST), pipelines may be flooded with tasks that will be squashed later. This is especially obvious in the case of SPEC-BFS, where pipeline utilization scales linearly while speedup degrades with increasing bandwidth. Thus rules should be chosen judiciously to avoid flooding invalid tasks into pipelines in speculative parallelization.

Finally, abundant fine-grained pipeline parallelism has been exploited. In all benchmarks the pipeline utilization rates scale linearly to memory bandwidth. This indicates that as long as data is available from the memory system, tens or hundreds of operations are executed simultaneously.

As a reference, we measured execution time of an OpenCL version of BFS from OpenDwarf [31], which is optimized for FPGA implementation with hand annotations, on an Altera Stratix IV GX EP4SGX530 (when this paper is submitted, Intel has not delivered support for OpenCL on HARP). The execution times of three accelerators on FPGA are listed in table 1. Accelerators generated from our framework are considerably better than that from current HLS tools. The performance advantage is mainly attribute to the elimination of both centralized control from the host processor and barriers for synchronizations in our framework.

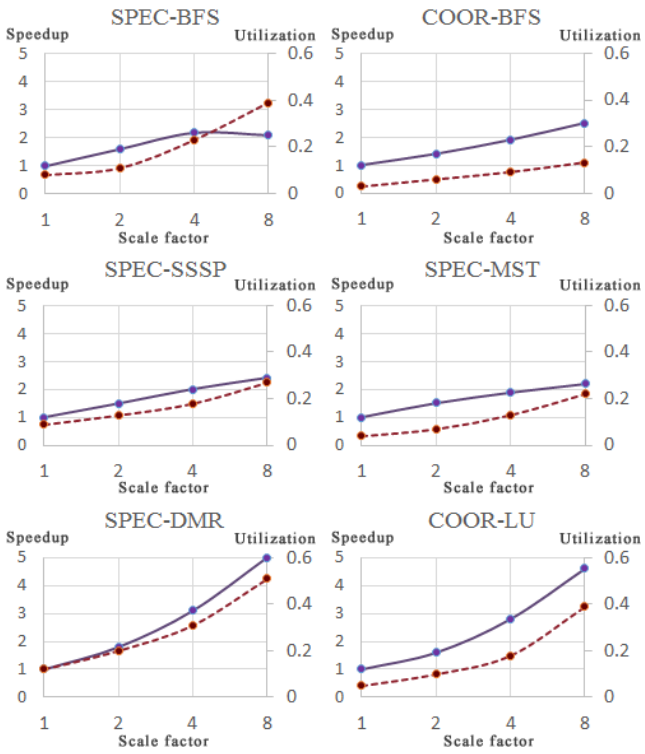


Figure 10: Speedup over baseline (solid, left y-axis) and utilization rate of pipelines (dash, right y-axis) when QPI bandwidth scales up

Table 1: Comparison of BFS in OpenCL to SPEC-BFS and COOR-BFS (in seconds)

Accelerator	OpenCL	SPEC-BFS	COOR-BFS
Best time	124.1	0.47	0.64

7 RELATED WORK

Our framework in a novel confluence of disparate categories.

Abstractions for FPGA. The necessity of abstraction for FPGA has long been recognized and carried out in HLS introduced in section 1. Recent works try to exploit parallel patterns in functional languages [29, 42] as high-level abstraction. A number of domain-specific architectural templates is emerging as lower-level abstractions [16, 20, 37]. Our framework adopts a layered abstraction in which task sets and rules abstract high-level constructs for applications, and templates abstract low-level architectural features.

Aggressive parallelization. There is rich literature in general software approaches to speculative parallelization for irregular applications, including thread-level speculation [18, 30] and software transactional memory [45, 47]. Run-time overhead in these approaches could be huge due to fine-grained synchronizations [11].

In recent works overhead could been alleviated by raising granularity [26, 33, 44], or use the runtime system for coordination [22]. A comprehensive software approach has been formulated as Tao analysis [41]. These approaches could be referred when designing accelerators for a broader range of applications in our framework.

Dataflow. Dating back to 1980s, dataflow was studied as a model of both computation [38] and architecture [5]. Recently dataflow is resurged motivated by its energy-efficiency [36], and has been stimulating broader interests due to its efficiency for special scenarios, such as control-intensive codes [40], memory access phase [23], and massive thread-level parallelism [49]. Our work relies an ECA rule grammar borrowed from [23] to express dataflow computation for rules, and exploits fine-grained pipeline parallelism by dataflow architectural designs.

8 CONCLUSION

In this work, we propose a software-hardware co-design framework to synthesize accelerators for irregular applications on reconfigurable platform from high-level specifications. Software designers could specify irregular applications as tasks whose parallel executions are orchestrated aggressively by rules. Then implementations on FPGA will be generated based on templates for each constructs in the specification. Evaluations show our framework exposes fine-grained pipeline parallelism, which is the gap between synthesized datapaths of current high-level language tools and handcrafted ones.

As our experiments suggested, the memory bandwidth remains the major obstacle in synthesized designs. Handcrafted accelerators handle data transfer aggressively by prefetching or preprocessing in problem-specific ways, which cannot be captured in current high-level abstractions. Another question for future work is how to automatically choose parameters for templated components when generating structures on FPGA. With proper abstractions and automatic design space explorations, developing hardware accelerator for irregular applications will be open to software developers.

9 ACKNOWLEDGMENT

We thank the anonymous reviewers and the shepherd for their thoughtful comments. This research is supported in part by the National High Technology Research and Development Program of China (Grant No. 2012AA012701) and National Natural Science Foundation of China (Grant No. 61672317).

REFERENCES

[1] 2006. 9th DIMACS Implementation Challenge: Shortest Paths. (2006). http://www.dis.uniroma1.it/challenge9/

[2] 2015. Xeon+fpga platform for the data center. (2015). https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[3] Altera Corporation. 2013. Implementing FPGA Design with the OpenCL Standard (Altera). (Nov. 2013).

[4] Altera Corporation. 2016. Altera SDK for OpenCL - Best Practices Guide. (2016).

[5] K. Arvind and Rishiyur S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.* 39 (March 1990), 300–318. https://doi.org/10.1109/12.48862

[6] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[7] David Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Queue* 11 (Feb. 2013), 40:40–40:52. https://doi.org/10.1145/2436696.2443836

[8] Daniel U. Becker and William J. Dally. 2009. Allocator implementations for network-on-chip routers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/1654059.1654112

[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/2145816.2145840

[10] Joseph Tobin Buck. 1993. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.D. Dissertation. University of California, Berkeley. AAI9431898.

[11] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6 (Sept. 2008), 40:46–40:58. https://doi.org/10.1145/1454456.1454466

[12] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. 2014. Enabling FPGAs in the Cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14)*. ACM, New York, NY, USA, 3:1–3:10. https://doi.org/10.1145/2597917.2597929

[13] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. 1996. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming* 73 (May 1996), 129–174. https://doi.org/10.1007/BF02592101

[14] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, 109:1–109:6. https://doi.org/10.1145/2897937.2897972

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30 (April 2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[16] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[17] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*. IEEE Computer Society, Washington, DC, USA, 124–131. https://doi.org/10.1109/ICPP.2009.64

[18] Rahulkumar Gayatri, Rosa. M. Badia, and Eduard Aygaude. 2013. Loop level speculation in a task based programming model. In *20th Annual International Conference on High Performance Computing*. 39–48. https://doi.org/10.1109/HiPC.2013.6799132

[19] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor P. Stefanov, Daniel D. Gajski, and JÃijrgen Teich. 2009. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (Oct. 2009), 1517–1530. https://doi.org/10.1109/TCAD.2009.2026356

[20] Paul Grigoras, Pavel Burovskiy, and Wayne Luk. 2016. CASK: Open-Source Custom Architectures for Sparse Kernels. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 179–184. https://doi.org/10.1145/2847263.2847338

[21] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. Unordered: A Comparison of Parallelism and Work-efficiency in Irregular Algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1941553.1941557

[22] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. 2015. Kinetic Dependence Graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 457–471. https://doi.org/10.1145/2694344.2694363

[23] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient Execution of Memory Access Phases Using Dataflow Specialization. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 118–130. https://doi.org/10.1145/2749469.2750390

[24] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. 2006. Streaming Computation of Delaunay Triangulations. In *ACM SIGGRAPH 2006 Papers (SIGGRAPH '06)*. ACM, New York, NY, USA, 1049–1056. https://doi.

org/10.1145/1179352.1141992

[25] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. 2014. SDA: Software-defined accelerator for large-scale DNN systems. IEEE Press, 1–23. https://doi.org/10.1109/HOTCHIPS.2014.7478821

[26] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. 2012. Speculative Separation for Privatization and Reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 359–370. https://doi.org/10.1145/2254064.2254107

[27] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] Kurt Keutzer, Sharad Malik, A.Richard Newton, Jan M. Rabaey, and Alberto Sangiovanni-Vincentelli. 2000. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19 (Dec. 2000), 1523–1543. https://doi.org/10.1109/43.898830

[29] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 115–127. https://doi.org/10.1109/ISCA.2016.20

[30] Venkata Krishnan and Josep Torrellas. 1999. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. Comput.* 48 (Sept. 1999), 866–880. https://doi.org/10.1109/12.795218

[31] Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. 2015. OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures. *Journal of Signal Processing Systems* 85 (Oct. 2015), 373–392. https://doi.org/10.1007/s11265-015-1051-z

[32] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin CasÃğaval. 2009. How Much Parallelism is There in Irregular Applications?. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1504176.1504181

[33] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[34] Edward A. Lee and Alberto Sangiovanni-Vincentelli. 1998. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17 (Dec. 1998), 1217–1229. https://doi.org/10.1109/43.736561

[35] Charles E. Leiserson and Tao B. Schardl. 2010. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/1810479.1810534

[36] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 298–310. https://doi.org/10.1145/2749469.2750380

[37] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 111–117. https://doi.org/10.1145/2847263.2847337

[38] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[39] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2013. Efficient Compilation of CUDA Kernels for High-performance Computing on FPGAs. *ACM Trans. Embed. Comput. Syst.* 13 (Sept. 2013), 25:1–25:26. https://doi.org/10.1145/2514641.2514652

[40] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[41] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario MÃľndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[42] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 651–665. https://doi.org/10.1145/2872362.2872415

[43] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[44] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/1736020.1736030

[45] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. ACM, New York, NY, USA, 187–197. https://doi.org/10.1145/1122971.1123001

[46] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[47] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

[48] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. 2015. Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In *Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE Computer Society, London, UK, 1–8. https://doi.org/10.1109/FPL.2015.7293939

[49] Dani Voitsechov and Yoav Etsion. 2014. Single-graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 205–216. http://dl.acm.org/citation.cfm?id=2665671.2665703

[50] Skyler Windh, Xiaoyin Ma, R.J. Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A. Najjar. 2015. High-Level Language Tools for Reconfigurable Computing. *Proc. IEEE* 103 (March 2015), 390–408. https://doi.org/10.1109/JPROC.2015.2399275

[51] Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2013. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In *2013 International Conference on Field-Programmable Technology (FPT)*. 362–365. https://doi.org/10.1109/FPT.2013.6718388

[52] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Accelerating Large-Scale Single-Source Shortest Path on FPGA. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*. IEEE Computer Society, Washington, DC, USA, 129–136. https://doi.org/10.1109/IPDPSW.2015.130