HECTOR: A Multi-level Intermediate Representation for Hardware Synthesis Methodologies

Ruifan Xu[1], Youwei Xiao[1], Jin Luo[1], Yun Liang[1,2]
[1]Peking University
[2]Beijing Advanced Innovation Center for Integrated Circuits
{xuruifan,shallwe,luo-jin,ericlyun}@pku.edu.cn

ABSTRACT

Hardware synthesis requires a complicated process to generate synthesizable register transfer level (RTL) code. High-level synthesis tools can automatically transform a high-level description into hardware design, while hardware generators adopt domain specific languages and synthesis flows for specific applications. The implementation of these tools generally requires substantial engineering efforts due to RTL's weak expressivity and low level of abstraction. Furthermore, different synthesis tools adopt different levels of intermediate representations (IR) and transformations. A unified IR obviously is a good way to lower the engineering cost and get competitive hardware design rapidly by exploring different synthesis methodologies.

In this paper, we propose Hector, a two-level IR providing a unified intermediate representation for hardware synthesis methodologies. The high-level IR binds computation with a control graph annotated with timing information, while the low-level IR provides a concise way to describe hardware modules and elastic interconnections among them. Implemented based on the multi-level compiler infrastructure (MLIR), Hector's IRs can be converted to synthesizable RTL designs. To demonstrate the expressivity and versatility, we implement three synthesis approaches based on Hector: a high-level synthesis (HLS) tool, a systolic array generator, and a hardware accelerator. The hardware generated by Hector's HLS approach is comparable to that generated by the state-of-the-art HLS tools, and the other two cases outperform HLS implementations in performance and productivity.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation**.

KEYWORDS

Intermediate Representation, Hardware Synthesis

ACM Reference Format:
Ruifan Xu[1], Youwei Xiao[1], Jin Luo[1], Yun Liang[1,2]. 2022. HECTOR: A Multi-level Intermediate Representation for Hardware Synthesis Methodologies. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22), October 30-November 3, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3508352.3549370

1 INTRODUCTION

Power efficiency has become a critical design factor with the continual demands of emerging applications including machine learning and scientific computation. As alternatives to general purpose CPUs and GPUs , customized hardware accelerators [16, 29] like ASICs and FPGAs provide good energy efficiency and performance. However, hardware design is still a big challenge. Hardware description languages (HDLs) including Verilog [20] and VHDL [19] are frequently used in industry for hardware design. These HDLs adopt a low level of abstraction including wires, registers, and gates, also known as register transfer level (RTL). However, HDLs' productivity is seriously hampered by this low level of abstraction, which makes it difficult for hardware designers.

Both high-level synthesis (HLS) and hardware generators have been proposed to improve the productivity of hardware design. HLS is a general method for automatically generating hardware from a behavioral description written in high-level languages such as C, C++, or OpenCL [24, 38, 40]. HLS offers a promising way to design hardware in a high-level abstraction, which provides opportunities for software engineers without hardware experience to design hardware quickly. However, it can lead to bad performance and high resource utilization due to the lack of domain knowledge about the computation and architecture [28]. On the other hand, hardware generator makes use of domain knowledge to improve productivity. These generators often focus on a specific domain, such as streaming [14, 34] and spatial accelerators [22, 23]. With the domain knowledge of the target applications, these approaches use specific languages and synthesis flows to generate hardware with good performance. Although these approaches provide high level abstraction, they can only support a specific class of applications.

Furthermore, different synthesis tools often adopt different synthesis methodologies. HLS compilers convert the high-level description to hardware implementation in three steps: allocation, scheduling, and binding. The scheduling step, which determines the cycles for each operation, can be implemented using different algorithms including static, dynamic, and hybrid scheduling [4, 24]. Hardware generators perform sophisticated architectural transformations to improve performance and resource consumption. In addition to the domain specific optimizations, hardware generators are often guided with specific hardware templates to generate hardware implementation for different applications [14, 22].

Both synthesis flows require substantial engineering efforts due to RTL's weak expressivity and low level of abstraction. HLS tools take high-level languages as inputs and use the compiler infrastructure.

For example, Vitis HLS [42] adopts LLVM IR [26] as its internal representation, which is a widely used software compilation IR. After that, additional passes are needed to convert to hardware semantics. As for hardware generators, domain specific languages (DSL) and optimizations are often used for description and optimization. For example, Tensorlib [22] selects tensor algebra as its representation, and maps it into the hardware PE array using space time transformation. However, these different methodologies share some similarities in the intermediate representations, such as control logic and hardware generation from a high-level abstraction. Therefore, it is possible to have a unified IR for different synthesis approaches so that users can easily design new hardware synthesis techniques based on the same infrastructure and explore different methodologies.

In this paper, we propose Hector, a two-level IR providing a unified description for different hardware synthesis methodologies with expressivity and flexibility. The high-level IR (TOpological Representation) binds computations with a control graph annotated with timing information, while the low-level IR (Hierarchical Elastic Componenent) provides a concise way to describe various hardware components and elastic interconnections among them using customizable primitives. Both IRs provide a uniform representation of the control logic with various manners, but at different abstraction levels. The IRs in Hector are converted to synthesizable RTL programs through a series of transformations including time graph transformation, lowering pass, and RTL generation. The two-level IR and all the transformations are built on the multi-level compiler infrastructure (MLIR) [27].

The main contributions of this work are as follows:

- We propose Hector, a unified IR and synthesis framework supporting multiple hardware synthesis methodologies including high-level synthesis and hardware generator.
- We propose a two-level IR that is general enough for different hardware synthesis methdologies.
- We implement a compiler that lowers the IRs to synthesizable RTL through a series of analysis and optimizations.

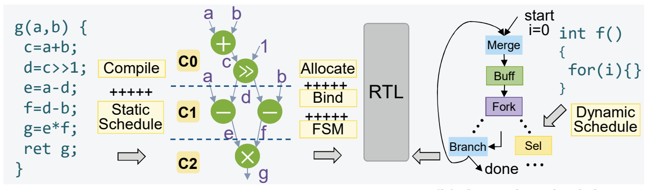
To demonstrate the expressivity and versatility, we implement three synthesis approaches based on Hector: a high-level synthesis (HLS) tool, a systolic array generator, and an accelerator component for sparse linear algebra. The experimental results show that the hardware generated by Hector's hybrid HLS scheduling improves the performance by 29% on average compared to that generated by the state-of-the-art HLS tool. The other two approaches outperform HLS implementations in performance and productivity. *Hector is open source at GitHub (https://github.com/pku-liang/Hector).*

2 BACKGROUND

2.1 High Level Synthesis

HLS offers a promising way to design hardware at a high-level abstraction such as C-like language, which can release users from designing at register transfer level. HLS compilers [2, 24, 38, 42] then automatically convert the high-level language to HDLs by three major processes: allocation, scheduling, and binding. Allocation places all compute units on the datapath, and binding determines where each operation executes.

Scheduling is the most important step in modern HLS tools. There are two different scheduling methodologies: static and dynamic, as



```
g(a,b) {
 c=a+b;
 d=c>>1;
 e=a-d;
 f=d-b;
 g=e*f;
 ret g;
}
```

```
int f()
{
 for(i){}
}
```

(a) static schedule (b) dynamic schedule

Figure 1: Static and dynamic scheduling in high level synthesis.

shown in Figure 1. The most common flow of static approach consists of 4 steps [9]: software compilation, static scheduling, allocating & binding, and finite state machine (FSM) construction, as shown in Figure 1(a). Compilation transforms the high-level program into a software intermediate representation. Then, static scheduling algorithm determines the execution time for operations considering dependencies and resource constraints. For example, the add and shift-right operations are placed in the first cycle (`C0`). Allocating & binding algorithms explore the opportunity for resource sharing, allocate necessary resources, and bind operations to them. Finally, a finite state machine is constructed for RTL generation.

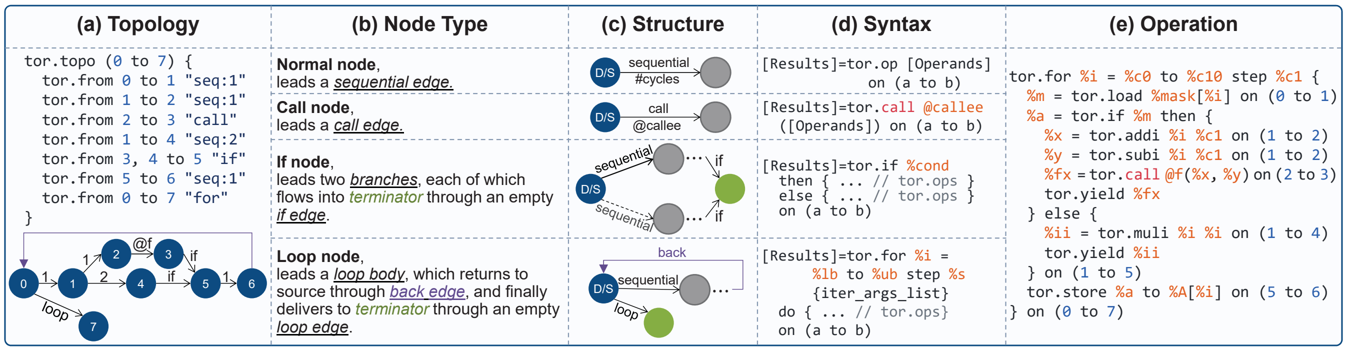
Static scheduling [3, 11, 17] is very effective for statically predictable programs such as perfect loops. Pipelining is a key optimization technique to exploit parallelism among multiple loop iterations. In sequential execution, one iteration can only begin after the previous iteration is finished. Pipelining allows iterations to overlap which improves throughput and gain resource sharing. The distance between two adjacent iterations is called initial interval (II) [18], which is the indicator of throughput.

Dynamic scheduling generates a dataflow circuit by leveraging elastic units [8], such as merge, branch, etc., as shown in Figure 1(b). All data signals in dataflow circuits are accompanied by handshake signals, which are *valid* and *ready*, in opposite directions, indicating the availability of the next data from the source unit and the readiness of the target unit to accept it, respectively. Static scheduling makes conservative assumptions for unresolvable dependencies at compile time. Dynamic scheduling overcomes this inefficiency by postponing the scheduling decisions until run time, However, dynamic scheduling suffers from high resource consumption. Hybrid scheduling combines static and dynamic schedulings [4, 5].

2.2 Intermediate Representation

Intermediate representation is an important abstraction to simplify the design of compilation and synthesis flows. Compiler infrastructures like LLVM [26] and GCC apply machine-independent optimizations and generate code for different target architectures based on IRs. For example, Static single assignment (SSA) form [10] and control dataflow graph (CDFG) representation are widely used in compiler optimizations and static analysis. Many HLS tools [2, 24, 42, 43] adopt LLVM IR as their internal representation. However, LLVM is a pure software IR that doesn't contain any hardware information. [21, 30, 33, 37, 38] present hardware IRs that provide a low-level abstraction to support hardware design.

MLIR [27] is a novel compiler infrastructure that provides powerful scalability and modularity. MLIR greatly facilitates the implementation of various IRs and transformations among them. All IRs in MLIR follow the SSA form and an explicit type system. *Dialect* in



(a) Topology

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "seq:1"
  tor.from 1 to 2 "seq:1"
  tor.from 2 to 3 "call"
  tor.from 1 to 4 "seq:2"
  tor.from 3, 4 to 5 "if"
  tor.from 5 to 6 "seq:1"
  tor.from 0 to 7 "for"
}
```

(b) Node Type

Normal node, leads a *sequential edge.*

Call node, leads a *call edge.*

If node, leads two *branches*, each of which flows into *terminator* through an empty *if edge*.

Loop node, leads a *loop body*, which returns to source through *back edge*, and finally delivers to *terminator* through an empty *loop edge*.

(d) Syntax

```
[Results]=tor.op [Operands]
          on (a to b)
```

```
[Results]=tor.call @callee
  ([Operands]) on (a to b)
```

```
[Results]=tor.if %cond
  then { ... // tor.ops }
  else { ... // tor.ops }
  on (a to b)
```

```
[Results]=tor.for %i =
     %lb to %ub step %s
     {iter_args_list}
  do { ... // tor.ops}
  on (a to b)
```

(e) Operation

```
tor.for %i = %c0 to %c10 step %c1 {
  %m = tor.load %mask[%i] on (0 to 1)
  %a = tor.if %m then {
    %x = tor.addi %i %c1 on (1 to 2)
    %y = tor.subi %i %c1 on (1 to 2)
    %fx = tor.call @f(%x, %y) on (2 to 3)
    tor.yield %fx
  } else {
    %ii = tor.muli %i %i on (1 to 4)
    tor.yield %ii
  } on (1 to 5)
  tor.store %a to %A[%i] on (5 to 6)
} on (0 to 7)
```

Figure 2: The design of ToR IR. ToR IR consists of topology and functional operations. In (a), topology describes the time graph with supplementary information on edges, where all edges and nodes are set "static" as default. In (e), operations are bound on the graph according to the syntax in (d). The binding situation determines the types of nodes on the time graph, as shown in (b). (c) shows the restrictions the node types set on the graph structure.

MLIR is a hierarchical structure template, allowing basic optimizations in MLIR to be reused. This hierarchy supplies an expressive representation, which makes it easy to implement a flexible IR.

3 INTERMEDIATE REPRESENTATION

In this section, we present the details of the IRs including ToR (TOpological Representation) and HEC (Hierarchical Elastic Component).

3.1 Overview

Hector contains a two-level IR system, where ToR is the high-level IR and HEC IR is the low-level IR. ToR IR combines a software-like control flow with the schedule information of each operation. HEC IR proposes an allocate-assign mechanism to explicitly describe the relationship between computation and compute units. Both IRs provide a uniform representation of the control logic with various scheduling manners such as static and dynamic. The main difference between the two IRs is that ToR describes when the operation begins, while HEC describes where it takes place.

Thanks to the expressiveness of the two-level representation, Hector supports versatile hardware synthesis methodologies. ToR is capable of providing a high-level abstraction of the scheduling information. Both dynamic and static scheduling in HLS can be captured by ToR IR. The allocation of hardware resources and binding of operations can then be described in HEC IR. The entire HLS procedure can then be obtained from a series of lowering transformations based on Hector IR. Hector can also be used to describe hardware at the architectural level, which is necessary for hardware generators. The allocate-assign method in HEC makes it easier to describe the interconnection between different modules at the low level, while high-level scheduling transformations can be applied to ToR IR.

3.2 ToR IR

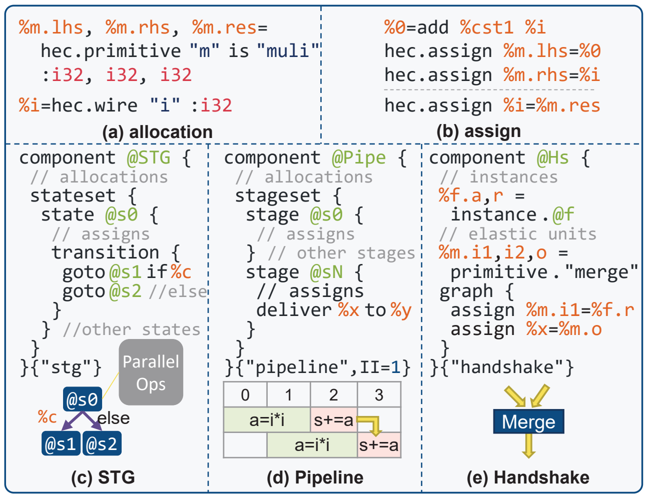
The software IR such as LLVM IR lacks hardware semantics. The idea of the high-level IR is to make it closer to hardware by providing a directed graph that carries control flow and timing information and binding software operations to elements of the graph. ToR is composed of two parts, topology and functional operations.

Topology describes a time graph, which is a directed graph describing control flow and timing information. Topology includes a `tor.topo (x to y)` operation, which indicates the source node `x` and sink node `y`, respectively. The `tor.from` operations inside `tor.topo` specify edges of the time graph. Attributes add supplementary information such as latency and scheduling manners to the time graph. There are four types of nodes in the time graph including normal, call, if and loop. Normal node leads a sequential edge, where `"seq:2"` indicates that the bounded operation takes two cycles. Call node represents a function call, where `"call"` indicates a state that stalls until the callee finishes. If a node leads two edges, and loop node leads a loop body and loop back edge. The type of each node on the time graph is determined by the operation binding. The combination of these four node types is capable of describing the schedule at high-level.

Three scheduling manners: **static**, **pipeline**, and **dynamic**, are supported in ToR. Pipelining is a key optimization technique to improve throughput. ToR supports pipelining by aligning branches of all `if` operations and adding `pipeline` and `II` attributes to modules. Topology also supports dynamic behavior that resolves conflicts at run-time. Stalling occurs only when the conflict occurs, avoiding the conservative assumption of static behaviors. This unified representation makes it easier to transform among different behaviors.

Functional operations present the algorithmic specification with high-level control flow semantics (e.g., if, for, and while). It binds each operation to some element of the time graph, either a node or an edge. To be specific, general operations (computation, memory access, function call) are bounded on edges, while if/loop operations are bounded on nodes. Functional operations specify functionality and binding according to the syntax in Figure 2(d). For example, the `tor.load` operation in Figure 2(e) is bounded on edge `(0 to 1)`, which loads the address `%i` of memory `%mask`. The `tor.for` operation that iterates from `%c0` to `%c1` is bounded on node `0`, and exits the loop at the edge `(0 to 7)`.

Figure 2 illustrates an example of ToR IR. The time graph in (a) contains a loop, which is composed of two branches $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 4$. There is also a function call on the edge $2 \rightarrow 3$, and the loop exits at the edge $0 \rightarrow 7$. Figure 2(b)(c) present the four types of nodes in the graph structure. Figure 2(e) shows the functional operations which are bounded on the time graph. For example, the `tor.for` operation is bounded on node `0`, and the `tor.muli` is bounded on



```
%m.lhs, %m.rhs, %m.res=
  hec.primitive "m" is "muli"
  :i32, i32, i32
%i=hec.wire "i" :i32
```

(a) allocation

```
%0=add %cst1 %i
hec.assign %m.lhs=%0
hec.assign %m.rhs=%i
hec.assign %i=%m.res
```

(b) assign

```
component @STG {
 // allocations
 stateset {
  state @s0 {
   // assigns
   transition {
    goto @s1 if %c
    goto @s2 //else
   }
  } //other states
 }
}{"stg"}
```

(c) STG

```
component @Pipe {
 // allocations
 stageset {
  stage @s0 {
   // assigns
  } // other stages
  stage @sN {
   // assigns
   deliver %x to %y
  }
 }
}{"pipeline",II=1}
```

0	1	2	3
a=i*i		s+=a	
	a=i*i		s+=a

(d) Pipeline

```
component @Hs {
 // instances
 %f.a,r =
  instance.@f
 // elastic units
 %m.i1,i2,o =
  primitive."merge"
 graph {
  assign %m.i1=%f.r
  assign %x=%m.o
 }
}{"handshake"}
```

(e) Handshake

Figure 3: Allocate-assign mechanism and three styles of components in HEC. (a) shows an example for unit allocation, where hec.primitive allocates a "muli" unit named "m" with three ports, and hec.wire declares a wire, which is commonly used in HDLs. (b) shows the usage of combinational operations and hec.assign for signal delivery among ports and wires. (c)-(e) presents three component styles.

edge `(1 to 2)`. As shown in Figure 4(a), there can be multiple manners in the same time graph, which are attached to the time graph as attributes like `"seq:2"` and `"dynamic"`.

3.3 HEC IR

Here, we propose HEC IR, which describes hardware with different manners in a unified allocate-assign mechanism depicted in Figure 3(a)(b). Allocation explicitly defines all function units and sub-modules on the datapath, and the signals of these units are determined through assignments. The allocate-assign mechanism omits the insertion of the multiplexer, simplifying the assignment of signals.

Compared with ToR, HEC works at a level much closer to hardware. It explicitly describes the resource usage (including registers, memory, and compute units). Corresponding to the different behaviors in ToR: **static**, **pipeline** and **dynamic**, a HEC design is composed of three types of components matching their manners as follows.

STG-style component. HEC describes a static module in a state transition graph (STG) style. The `hec.stateset` operation defines a set of states, as shown in Figure 3(c). Inside each state `@sx`, `hec.assign` operations specify the signal delivery among the allocated resources. Such representation naturally supports fine-grained parallelism. There is also a `tor.transition` operation in every state, specifying which state the control is transferred to, either unconditionally or based on guard signals. Based on the allocate-assign mechanism, it is convenient to describe resource sharing in an STG-style component by simply feeding signals into the shared resources (either registers or compute units) inside different states.

Pipeline-style component. The component for a pipelined module is described in a multi-stage style. HEC explicitly presents all pipeline stages by `hec.stage` operations inside the `hec.stageset` operation, as shown in Figure 3(d). For each value, HEC allocates a register to carry it for each stage between its definition and the latest use. This is because there may be time overlap between the lifetimes of the same value in consecutive executions. Signal delivery description is similar to that of STG-style components, while resource sharing additionally needs to consider the initial interval (II) for conflict avoidance [12, 44]. We also define operation `hec.deliver` to specify inter-iteration data delivery.

Handshake-style component. HEC describes the dynamic submodule in handshake style [8]. In order to simplify the description, extra signals in the handshake protocol, such as valid and ready, are hidden in this kind of component. With elastic units, such as `branch` and `fork`, predefined as primitives in HEC, it is easy to describe the interconnections among components with the allocate-assign mechanism inside the `hec.graph` operation, as shown in Figure 3(e).

3.4 IR Implementation

Hector is built on a novel compiler infrastructure, MLIR [27], that provides powerful scalability and modularity. Both IRs are implemented as *dialects* in MLIR infrastructure. MLIR provides a generic form of operations, which can be customized for new languages. Besides basic operands and results, operations may have *attributes* and *regions*. Figure 2(d) shows the functional syntax of operations in ToR. There are two *regions* in `tor.if` representing two branches separately. The nested region is naturally supported in MLIR, which enables the definition of `stateset` and `stageset` in Figure 3(c)(d). The different behaviors of schedule, the binding of operations and the instantiation of sub-module are all implemented as *attributes* including literal and symbolic references.

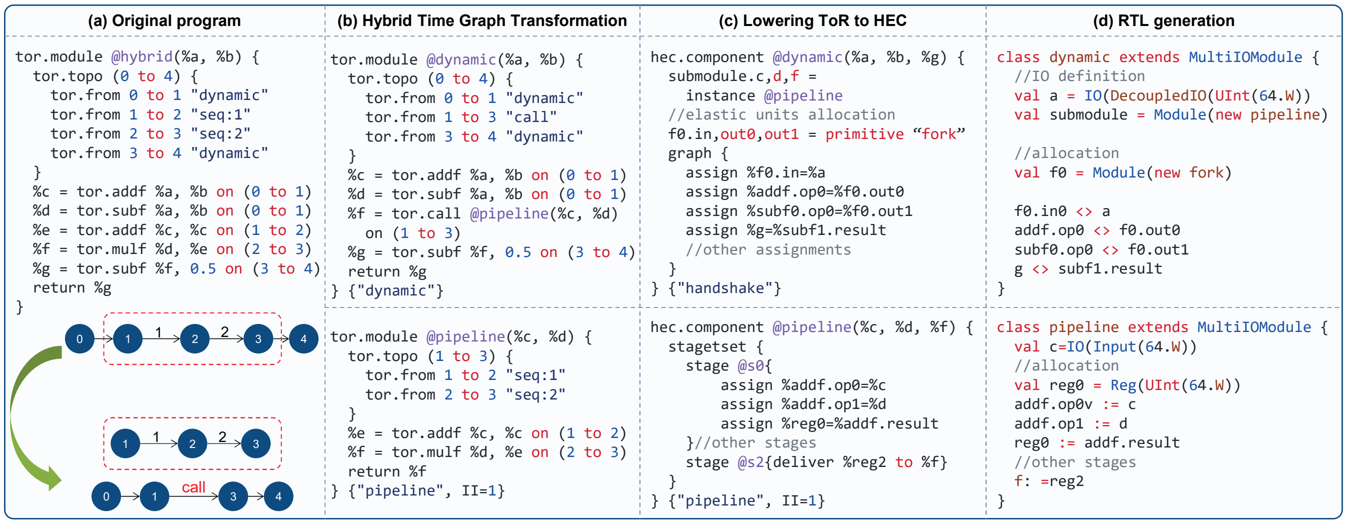
The two-level representation makes it easy to implement different synthesis methods. ToR IR provides a high-level abstraction of schedule information, and different scheduling approaches can be easily implemented by transformation on the time graph. The explicit representation of allocation in HEC IR brings the opportunity for resource sharing, which significantly reduces resource consumption. The allocation of sub-modules forms a hierarchical representation of hardware, which is capable of describing the architectural design. The allocate-assign mechanism also simplifies the definition of interconnection between different modules. Therefore, both HLS and hardware generators can be easily implemented in Hector IR.

4 COMPILING IRS TO HARDWARE

To compile IR to hardware, Hector involves the following passes, 1) Time Graph Transformation (4.1), which converts the time graph in ToR into subgraphs by outlining graph connection, 2) Lowering from ToR to HEC (4.2), which produces HEC programs with the same functionality of ToR through pipeline partition, control logic generation, and resource sharing, 3) RTL Generation (4.3), which generates Chisel programs by implementing hardware controller and inserting necessary signals. Figure 4 illustrates the main steps from HEC to RTL implementation written in Chisel.

4.1 Time Graph Transformation

ToR's semantics allows for the existence of multiple scheduling manners in the same time graph. However, because different types



(a) Original program

```
tor.module @hybrid(%a, %b) {
  tor.topo (0 to 4) {
    tor.from 0 to 1 "dynamic"
    tor.from 1 to 2 "seq:1"
    tor.from 2 to 3 "seq:2"
    tor.from 3 to 4 "dynamic"
  }
  %c = tor.addf %a, %b on (0 to 1)
  %d = tor.subf %a, %b on (0 to 1)
  %e = tor.addf %c, %c on (1 to 2)
  %f = tor.mulf %d, %e on (2 to 3)
  %g = tor.subf %f, 0.5 on (3 to 4)
  return %g
}
```

(b) Hybrid Time Graph Transformation

```
tor.module @dynamic(%a, %b) {
  tor.topo (0 to 4) {
    tor.from 0 to 1 "dynamic"
    tor.from 1 to 3 "call"
    tor.from 3 to 4 "dynamic"
  }
  %c = tor.addf %a, %b on (0 to 1)
  %d = tor.subf %a, %b on (0 to 1)
  %f = tor.call @pipeline(%c, %d)
    on (1 to 3)
  %g = tor.subf %f, 0.5 on (3 to 4)
  return %g
} {"dynamic"}
```

```
tor.module @pipeline(%c, %d) {
  tor.topo (1 to 3) {
    tor.from 1 to 2 "seq:1"
    tor.from 2 to 3 "seq:2"
  }
  %e = tor.addf %c, %c on (1 to 2)
  %f = tor.mulf %d, %e on (2 to 3)
  return %f
} {"pipeline", II=1}
```

(c) Lowering ToR to HEC

```
hec.component @dynamic(%a, %b, %g) {
  submodule.c,d,f =
    instance @pipeline
  //elastic units allocation
  f0.in,out0,out1 = primitive “fork”
  graph {
    assign %f0.in=%a
    assign %addf.op0=%f0.out0
    assign %subf0.op0=%f0.out1
    assign %g=%subf1.result
    //other assignments
  }
} {"handshake"}
```

```
hec.component @pipeline(%c, %d, %f) {
  stagetset {
    stage @s0{
        assign %addf.op0=%c
        assign %addf.op1=%d
        assign %reg0=%addf.result
    }//other stages
    stage @s2{deliver %reg2 to %f}
  }
} {"pipeline", II=1}
```

(d) RTL generation

```
class dynamic extends MultiIOModule {
  //IO definition
  val a = IO(DecoupledIO(UInt(64.W))
  val submodule = Module(new pipeline)

  //allocation
  val f0 = Module(new fork)

  f0.in0 <> a
  addf.op0 <> f0.out0
  subf0.op0 <> f0.out1
  g <> subf1.result
}
```

```
class pipeline extends MultiIOModule {
  val c=IO(Input(64.W))
  //allocation
  val reg0 = Reg(UInt(64.W))
  addf.op0v := c
  addf.op1 := d
  reg0 := addf.result
  //other stages
  f: =reg2
}
```

Figure 4: An overall example for compiling Hector to hardware. The original module with hybrid manners is split into two modules with dynamic and pipeline manners separately, and the connection between them is represented as a function call. Through resource and register sharing, pure ToR modules are transformed to HEC components containing compute units and registers. RTL implementations are generated by mapping ports and constructing controllers with the same functionality.

of hardware have different interfaces and cannot communicate directly with one another, this time graph is unsuitable for subsequent transformations. This pass converts this time graph into multiple sub-graphs where all edges and nodes have the same manner. The connected sub-graph and related computation are defined as external modules, and communication between the outer and inner modules is accomplished through function calls.

In order to get inputs and outputs of the new function, we calculate all live-in and live-out variables of the connected sub-graph. Live-in variables are the arguments of the new function, and that function returns the values of all live-out variables. As shown in Figure 4(a), the connected sub-graph of time graph is marked by the red rectangle. That time graph is split into two sub-graphs: a dynamic module and a pipeline module described in Figure 4(b). The inner pipeline module takes `%c` and `%d` as inputs, and returns the value of `%f`. This pass ensures that only one type of hardware is considered in the following transformations.

4.2 Lowering ToR to HEC

The time graph in static modules is converted to a low-level FSM with the same functionality. For pipeline ToR modules, this pass traverses the time graph and calculates the "distance" of each node from the source node. The "distance" is indicated by “#cycle” attribute of every time edge composing the path from the source node to the current node. The time nodes with the same "distance" will be placed in one stage. Each stage corresponds to a clock cycle, on which the allocated time node starts the execution of the operations whose start time is set as the current node.

In contrast to ToR, HEC specifies where the computation takes place and where the intermediate value is stored. This opens up the possibility of resource and register sharing, which allows disjoint computations and values to share the same unit. ToR fits well in such optimizations because of the high-level control flow and timing information of computation contained in the time graph.

Resource and register sharing. Resource sharing can only happen between two operations that never occur at the same time, so two disjoint operations mustn't overlap in the time graph. We build a conflict graph that shows all potential conflicts and uses a coloring strategy to assign compute units to each node. As for register sharing, we use a live range analysis on the time graph to calculate the live range of each variable. The value of a variable in its live range must be stored in a register unless it is used immediately. The main difference between register sharing and resource sharing is that each edge in the conflict graph represents an overlapping in live range.

As for hardware with dynamic behavior, each pair of operations may happen at the same time, restricting resource sharing. As a result, the lower pass just considers the generation of control logic. The handshake protocol, which describes data transfer with valid and ready signals, is used to implement the dynamic approach. This protocol is hidden in the HEC IR, which further simplifies the description. Elastic components [8] are adopted to solve the synchronization of tokens in the hardware with dynamic behavior. For example, `branch` transfers the data to one of the outputs based on a condition, and `merge` accepts one of the inputs which is similar to a ϕ function [10] in software IR. We calculate the live-in and live-out variables of each regions, and generate data transfer of variables among different region. After that, `fork` is inserted to synchronize multiple uses of the same value, which is shown in Figure 4(c). Both `addf` and `subf` need the value of `%a`, so a `fork` component is inserted for that variable. The two outputs of `fork` are fed into `addf` and `subf` separately.

4.3 RTL Generation

HEC contains sufficient information to generate synthesizable RTL. Each component in HEC has explicit I/O port definitions, assignments of ports, a controller, and allocations of register and resource (including memory, compute units, and other sub-component). The RTL generation pass creates a Chisel [1] program by mapping each component to a Chisel module with the same ports and assigning the corresponding value to these ports. Chisel will automatically insert the multiplexer, allowing for multiple assignments under different conditions. All of the built-in resources, such as memory and compute units are implemented as a pre-defined Chisel Module.

Controller Generation. The STG-style component is controlled by a state transition graph that contains a set of states and the transition between them, whereas the pipeline-style component is controlled by a collection of stages and matching II. By inserting some extra registers and wires, this pass builds a Chisel-style FSM for these two types of components with the same functionality in `ToR`. Due to the need to resolve conflicts at run-time, handshake-style components do not have a centralized controller. For each port, the handshake protocol is implemented as a Chisel class `DecoupledIO` with ready, valid, and data signals. The Chisel program for a pipeline component and a dynamic component is shown in Figure 4. The controller of the pipeline component is eliminated because the II equals one.

Wrapper construction. Due to the distinction between the handshake-style component and the other two types of components, this pass additionally generates a wrapper that attaches the handshake protocol to these components. Whether an STG-style (or pipeline-style) component is valid and ready is determined by the component's completion (or the validity of the initial stage and final stage).

5 CASE STUDIES

We use Hector to build three synthesis approaches: an HLS tool, a systolic array generator, and an accelerator component for sparse linear algebra. These case studies are chosen to show how Hector can be used to build various synthesis tools.

5.1 High-level Synthesis

Recently, [4, 5] demonstrates the effectiveness of hybrid scheduling. Their design is based on LLMV IR. However, LLVM IR is mainly a software compilation IR and lacks hardware semantics. Therefore, one central question is *what are the intermediate abstractions suitable for both static and dynamic HLS scheduling*?

In this case study, we build an HLS tool based on Hector to demonstrate the generality of the proposed IR. MLIR provides several built-in Dialects, such as Affine, SCF, and Standard [27]. SCF Dialect describes static control flow in a higher-level abstraction which is selected as the input of HLS. To build a complete HLS flow, we additionally implement an SDC-based module scheduling [44] that converts SCF Dialect to `ToR`. The scheduling result can be easily represented by `ToR` because of the concise structure of the time graph. Hybrid scheduling is also supported as a hybrid time graph that contains multiple manners in the same graph. We design a simple partition strategy on `ToR` that converts the manners of certain edges from static to dynamic. This demonstrates that Hector can flexibly support different HLS scheduling strategies.

Benchmarks. We evaluate using both regular and irregular applications. The regular kernels are selected from Machsuite[35] including GEMM, Stencil2D, Stencil3D, and Spmv(CSR). These four kernels do not have any branch, and thus are well suited for static scheduling. The irregular kernels are AELoss Pull and AELoss Push from the loss function of [32]. For these two kernels, a loop-carried dependence resides inside the innermost loop. A considerable amount of computation is guarded by an unpredictable condition, which is seldom true.

Methodology. Polygeist[31] is used as Hector's front-end to convert C programs into SCF dialect in MLIR, then Hector generates an RTL file in Verilog targeting FPGA part xc7z020clg484. We obtain cycle numbers using RTL level simulation. Timing and resource results are obtained from post synthesis report from Vivado 2021.1.

In the following, we first compare Hector with the state-of-the-art HLS frameworks. Specifically, we compare static scheduling in Hector with Vitis HLS [42] and dynamic scheduling with Dynamatic [24] [1]. For Vitis HLS, we apply pipeline directives on the inner-most loop. For comparison with Dynamatic, we convert the floating-point operations for benchmarks AELoss Pull and AELoss Push to their integer equivalence because the open-source Dynamatic with floating-point computation fails to run. Then, we evaluate the effect of our hybrid scheduling in Hector by comparing it with static and dynamic scheduling. In Hector, we can switch between static, dynamic, and hybrid scheduling by easily configuring the scheduling modes provided as an attribute of SCF functions.

Comparison with state-of-the-art HLS. Table 1 demonstrates that Hector can achieve comparable performance with Vitis HLS and Dynamatic for the tested kernels. The number of DSP in Vitis HLS is different because we use a different IP configuration. For Stencil2D and Stencil3D benchmarks, we attribute the large consumption of resources to our resource binding algorithm compared with Vitis HLS. Hector generates smaller hardware than Dynamatic because of Chisel's internal optimizations, such as width analysis. For the AELossPush benchmark, the resource usage of Dynamatic is substantially higher because LLVM IR, the input language of Dynamatic, has too many basic blocks, which have a negative impact on Dynamatic's basic blocks based scheduling algorithm.

Hybrid scheduling. For the AELossPull benchmark, the major bottleneck is caused by the loop-carried dependence guarded by a condition. The loop-carried dependence causes a conservative II in static scheduling, while dynamic scheduling yields better throughput. Our partition algorithm extracts out the operations except the ones associated with the loop-carried dependence into a static submodule. The submodule contains four floating-point multiplications and four floating-point additions, which can be implemented by only two multipliers and two adders in static scheduling. However, these operations can not share any resources in dynamic scheduling, leading to massive DSP consumption. The hybrid scheduling can retain the performance improvement of dynamic scheduling while reducing resource usage by enabling resource sharing inside static submodules. Static (dynamic) scheduling is effective for pure regular (irregular) applications, while hybrid scheduling is a promising solution for complex applications with a mixture of regular and irregular behaviors. Overall, the hybrid scheduling improves the performance by 29% compared to static on average.

AELoss Push contains a two-level loop nest with conditions. The comparison results between the three scheduling modes are similar to AELoss Pull except for the higher FF usage in static and hybrid scheduling. This is mainly because of our register allocation strategy. The complicated loop nest and conditions exacerbate the buffer

[1]Dynamatic only provides one buffer insertion strategy based on a Mix ILP solver, which is time-consuming and unscalable. Buffer insertion is disabled in both tools for a fair comparison.

Table 1: Timing and Resource Comparison between Hector and Vitis HLS (Vitis), Dynamatic (DYN)

Benchmark	Slices			LUTs			FFs			DSPs			Cycles			Period (ns)			Time (ms)		
	Vitis	ours	ratio	Vitis	ours	ratio	Vitis	ours	ratio	Vitis	ours	ratio	Vitis	ours	ratio	Vitis	ours	ratio	Vitis	ours	ratio
GEMM	486	441	0.846	852	890	1.045	1958	1600	0.817	14	13	0.929	3932.1k	3752.1k	0.954	5.073	4.14	0.816	19.948	15.533	0.779
Stencil2D	47	138	2.936	94	192	2.043	188	370	1.968	3	4	1.333	320.5k	312.9k	0.976	4.545	3.904	0.859	1.457	1.221	0.838
Stencil3D	204	245	1.201	454	372	0.817	668	890	1.332	6	8	1.333	102.5k	103.7k	1.011	5.692	4.672	0.821	0.583	0.484	0.30
SPMV (CSR)	502	438	0.873	881	932	1.058	1934	1625	0.840	14	13	0.929	37.1k	34.2k	0.920	5.299	4.848	0.915	0.197	0.165	0.842
Normalized mean			**1.388**			**1.189**			**1.177**			**0.964**			**0.966**			**0.876**			**0.845**
	DYN	ours	ratio	DYN	ours	ratio	DYN	ours	ratio	DYN	ours	ratio	DYN	ours	ratio	DYN	ours	ratio	DYN	ours	ratio
AEloss Pull	115	99	0.860	331	280	0.846	265	212	0.800	-	-	-	12.5k	14.7k	1.175	6.1	5.569	0.913	0.076	0.082	1.073
AEloss Push	365	87	0.238	1118	250	0.224	900	199	0.221	-	-	-	326.1k	293.7k	0.901	6.2	5.5	0.887	2.022	1.616	0.799
Stencil2D	534	408	0.764	1626	1227	0.755	1379	891	0.646	-	-	-	429.8k	398.6k	0.928	7.277	6.590	0.906	3.128	2.627	0.840
Normalized mean			**0.621**			**0.680**			**0.556**			**-**			**1.001**			**0.901**			**0.903**

Table 2: Timing and Resource Comparison between Hector's static scheduling (S), dynamic scheduling (D) and hybrid scheduling (H)

Benchmark	Slices			LUTs			FFs			DSPs			Cycles			Period (ns)			Time (ms)		
	S	D	H	S	D	H	S	D	H	S	D	H	S	D	H	S	D	H	S	D	H
AELoss Pull	833	2001	1607	1844	5216	3447	3198	7193	7737	16	55	29	15.4k	9.7k	9.3k	5.378	5.683	5.765	0.083	0.055	0.054
AELoss Push	2539	2429	3058	4471	7593	7339	11123	8445	11543	6	15	12	1506.2k	970.9k	723.6k	6.004	6.058	6.14	9.043	5.882	4.443
Stencil2D	138	442	138	192	1219	192	370	924	370	4	4	4	312.9k	304.9k	312.9k	3.904	6.4	3.904	1.221	1.951	1.221
Normalized mean	**1**	**2.139**	**1.378**	**1**	**3.625**	**1.503**	**1**	**1.835**	**1.486**	**1**	**2.313**	**1.6044**	**1**	**0.748**	**0.695**	**1**	**1.235**	**1.032**	**1**	**0.969**	**0.713**

insertion algorithm's effect, while the regular behavior mitigates this situation in static scheduling. As a result, hybrid scheduling preserves the irregular behavior and achieves a better throughput with the help of static submodules. As for the Stencil2D benchmark, the partitioning algorithm chooses the whole program as a static module because the computation has no branches and is very regular. In that case, dynamic scheduling can only get comparable performance while consuming a large number of resources.

The comparison result against existing HLS tools demonstrates that Hector can fit well in HLS no matter with static, dynamic, or hybrid manners. [5] implements a hybrid scheduling strategy by partitioning LLVM IR into multiple functions and synthesizing them with separate HLS tools. In contrast, Hector allows users to customize HLS algorithms (such as the partitioning algorithm) implemented as MLIR passes, which facilitates easy extension and allows more optimization opportunities.

5.2 Systolic Array Generator

Systolic array is a class of accelerators where inputs or results pass through the processor elements (PEs) [7, 22, 23]. In this case study, we use Hector to build a systolic array generator. This generator can generate PE arrays of arbitrary size given a PE's configuration, which can be written in either ToR or HEC. In addition, SCF Dialect is also supported by reusing the previous HLS process (5.1).

Implementation. We use an elastic implementation that allows each PE to stall if the data does not arrive or the subsequent PE is not ready to receive new data. Supporting dynamic behavior is critical, particularly when bandwidth is limited. When sufficient bandwidth is ensured, an elastic systolic array performs similarly to the static systolic array with the exception of a few more signals. This dynamic behavior is implemented as an elastic interconnection between the PE array. Figure 5 shows a 2×2 systolic array represented in HEC. The outer module is a dynamic style component that enables the implementation to be elastic. All PEs are instantiated

```
component @PE(%Ain, %Bin, %Aout, %Bout) {}
component @Systolic_Array(...) {
  //Instantiate PE Array
  %pe_00.Ain, Bin, Aout, Bout = instance @PE
  %pe_01.Ain, Bin, Aout, Bout = instance @PE
  %pe_10.Ain, Bin, Aout, Bout = instance @PE
  %pe_11.Ain, Bin, Aout, Bout = instance @PE
  graph {
    assign %pe_01.Ain = %pe_00.Aout
    assign %pe_10.Bin = %pe_00.Bout
    assign %pe_11.Ain = %pe_10.Aout
    assign %pe_11.Bin = %pe_01.Bout
  }
}
```

Figure 5: A 2×n by n×2 matrix multiplication systolic array described in HEC. The interconnection between PEs is described as hec.assign.

from the corresponding component, and data movement between PEs is explicitly stated as hec.assign.

A single PE can be configured at multiple levels. For the PE with simple behavior, SCF Dialect can be automatically lowered using the approach described in subsection 5.1. HEC can be used in applications with no resource limitations, and ToR is capable of handling complex designs. This once again demonstrates the advantages of multi-level IR. Pipelined PE can be easily expressed at these levels due to the explicit pipeline in both levels.

Comparison against Calyx. Calyx [33] also provides a static systolic array generator for demonstration purposes, but it lacks the expressivity required for hardware generation. More than 3000 lines of code are required to describe an 8×8 systolic array. The fundamental reason is that the control logic is described in a flattened representation that explicitly represents all data moves at each cycle. Another issue in Calyx is that it lacks a pipeline primitive, prohibiting all PEs in generated hardware from being pipelined. In real-world applications, interleaving multiple calculations is commonly used to

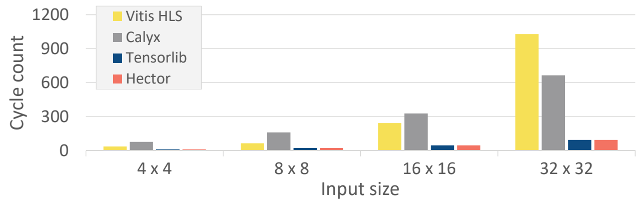


Figure 6: Cycle count comparison between different implementations of matrix multiplication.

Table 3: The comparison between Vitis HLS and Hector.

	Line of Codes	Initial Interval
Hector (RTL)	35	1
Vitis HLS	111	1

improve resource utilization ratio, but the lack of pipeline primitives makes this optimization impossible.

Evaluation. We evaluate matrix-multiplication kernels ranging from 4×4 to 32×32. The PE configuration of the systolic array is written in SCF, which is automatically lowered to HEC. Aside from the systolic arrays generated by Hector and Calyx, we implement a basic program in Vitis HLS that uses pipeline pragma in the inner-most loop. We optimize the HLS implementation by unrolling the outer two loops and partitioning all the matrices in the proper dimension to make a fair comparison. We also compare it with Tensorlib [22], a state-of-the-art systolic array generator. The experiment targets Zynq UltraScale+ XCZU3EG FPGA at a 7ns clock period, which is the same as Calyx's setup. We compare the cycle counts of the designs (Figure 6). Hector supports pipeline PE, which further optimizes the performance by interleaving multiple computations. On average, the systolic array generated by Hector improves the performance by 5.6× improvement compared to optimized HLS implementation. In comparison to the HLS implementation, Calyx implementation with sequential PEs can only achieve 79% performance, demonstrating the importance of pipeline semantics. Hector can achieve the same performance with Tensorlib thanks to the structural description and pipeline support.

The experiment results demonstrate Hector's versatility in applying domain-specific approaches, which is difficult to express in HLS tools. With architectural information, Hector can outperform the general-purpose HLS tool. The multi-level paradigm also provides the capability to design hardware at various levels.

5.3 Shuffle Unit

Shuffle unit is the key hardware component that solves bank conflicts between multiple PEs, which has been used in sparse matrix-vector multiplication (SPMV) design [13]. Each pair of streams may access an arbitrary bank determined by the column index, resulting in traffic conflict. However, it is challenging to implement the shuffle unit with optimized throughput in HLS tools [13]. In static scheduling, the conservative assumption of traffic pattern results in a big II, and dynamic scheduling is not suited to generate efficient control logic. Therefore, it's better to implement the shuffle unit at the RTL. In Hector, the combination of RTL and other synthesis approaches is simple. RTL design can be easily integrated with other synthesis approaches using the Chisel programs and dummy representations.

We use Hector to construct an RTL design and simplify a pipeline kernel [13] written in Vitis HLS. The RTL implementation takes only 35 lines to implement the control logic, while the optimal HLS implementation uses 111 lines. As previously stated, the conservative assumption in HLS results in a large II. Due to explicit control logic in HLS implementation, it gets the optimal II which matches the RTL implementation. The experiment demonstrates that HLS is not suitable for control logic. Therefore, combining HLS and RTL design is a superior option for achieving good performance while maintaining enough productivity.

6 RELATED WORKS

Domain-specific languages. DSLs provide a higher level of abstraction that is natural to the domain, assisting in the generation of hardware with expert knowledge. Raghu Prabhakar et al. [34] adopt parallel patterns like *map* and *fold* to express high-level computation. Aetherling [14] aims at generating streaming accelerators by applying transformations to the proposed data-parallel IR. Nithin George et al. [15] generate hardware systems from applications written in a machine-learning DSL. HeteroCL [25] provides an abstraction that decouples algorithm description and architecture specification, and develops a compilation flow to heterogeneous computing platforms. Apart from domain-specific optimizations, extra compilation to the RTL program is required, which makes it difficult for researchers to generate hardware quickly.

IR for hardware. FIRRTL [21] that obeys AST format is the intermediate representation in Chisel. LLHD [37] is a three-level IR that aims at different applications including simulation, verification, and logic synthesis. μIR [38] uses dynamic scheduling based on task-level parallelism and briefly presents a task-level representation. Wu et al. and synASM [39, 41] propose a CDFG representation for hardware and software. AHIR [36] proposes a low-level abstraction that decouples the datapath and control path. Calyx[33] provides software-like control flow primitives such as seq, par, and while to describe hardware. The goal of CIRCT [6] is to construct a reusable and modular infrastructure for the entire hardware generation including high-level synthesis and logic synthesis. This project is still in progress and absorbs existing IR designs like Calyx and LLHD.

7 CONCLUSION

In this paper, we propose Hector, a two-level IR providing a unified description for different synthesis methodologies. Through a series of transformations and optimizations based on the MLIR infrastructure, Hector's IRs are finally converted to synthesizable RTL programs. We demonstrate the expressivity and effectiveness of our design by implementing three synthesis approaches. The experiment results show that Hector can generate comparable hardware designs with existing HLS tools in terms of performance and resource utilization. With the combination of different methodologies, it's simple for Hector to outperform HLS tools. Moreover, the open-source framework provides flexibility to customize synthesis approaches and allows users to explore advanced techniques.

8 ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation of China (NSFC) under grant No.62090021.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*.

[2] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*.

[3] Hongzheng Chen and Minghua Shen. 2019. A Deep-Reinforcement-Learning-Based Scheduler for FPGA HLS. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[4] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*.

[5] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Finding and Finessing Static Islands in Dynamically Scheduled Circuits. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*.

[6] CIRCT Community. 2021. *CIRCT: Circuit IR Compilers and Tools*. Retrieved March 7, 2021 from https://github.com/llvm/circt

[7] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann. 2006. Synthesis of synchronous elastic architectures. In *2006 43rd ACM/IEEE Design Automation Conference*.

[9] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. 2009. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers* 26, 4 (2009).

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991).

[11] Steve Dai and Zhiru Zhang. 2019. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*.

[12] Steve Dai and Zhiru Zhang. 2019. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*.

[13] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. 2022. High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*.

[14] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*.

[15] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*.

[16] Kartik Hegde and et al. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*.

[17] Hsuan Hsiao and Jason Anderson. 2019. Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*.

[18] Yu-Chin Hsu and Yuang-Long Jeang. 1993. Pipeline scheduling techniques in high-level synthesis. In *Sixth Annual IEEE International ASIC Conference and Exhibit*.

[19] IEEE. 1076-2008. *VHDL Language Reference Manual.*

[20] IEEE. 1364-2005. *Standard for Verilog Hardware Description Language*.

[21] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design* (Irvine, California) *(ICCAD '17)*.

[22] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2021. TensorLib: A Spatial Accelerator Generation Framework for Tensor Algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*.

[23] Liancheng Jia, Yuyue Wang, Jingwen Leng, and Yun Liang. 2022. EMS: Efficient Memory Subsystem Synthesis for Spatial Accelerators. In *2022 59th ACM/IEEE Design Automation Conference (DAC)*.

[24] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*.

[25] Yi-Hsiang Lai and et al. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*.

[26] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*

[27] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[28] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N. Do, and Deming Chen. 2012. High-Level Synthesis: Productivity, Performance, and Software Constraints. *JECE* 2012, Article 1 (jan 2012), 1 pages. https://doi.org/10.1155/2012/649057

[29] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. 2016. High Level Synthesis of Complex Applications: An H.264 Video Decoder. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '16)*.

[30] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-Centric Notation. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press. https://doi.org/10.1109/ISCA52012.2021.00062

[31] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[32] Alejandro Newell, Zhiao Huang, and Jia Deng. 2017. Associative Embedding: End-to-End Learning for Joint Detection and Grouping. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*.

[33] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*.

[34] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. *SIGPLAN Not.* 51, 4 (mar 2016).

[35] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*.

[36] Sameer D. Sahasrabuddhe, Hakim Raja, Kavi Arya, and Madhav P. Desai. 2007. AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*.

[37] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*.

[38] Amirali Sharifian and et al. 2019. μIR -An Intermediate Representation for Transforming and Optimizing the Microarchitecture of Application Accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*.

[39] Rohit Sinha and Hiren D. Patel. 2012. synASM: A High-Level Synthesis Framework With Support for Parallel and Timed Constructs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 10 (2012).

[40] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[41] Qiang Wu and et al. 2002. A hierarchical CDFG as intermediate representation for hardware/software codesign. In *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions*, Vol. 2.

[42] Xilinx. 2021. *Vitis High-Level Synthesis*. Retrieved March 7, 2021 from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[43] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[44] Zhiru Zhang and Bin Liu. 2013. SDC-Based modulo Scheduling for Pipeline Synthesis. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '13)*.