OPTIMIZED MEMORY ACCESS FOR DYNAMICALLY SCHEDULED HIGH LEVEL SYNTHESIS

Submitted on the 29th of June, 2018
in part fulfillment of the
requirements for the degree of
Master of Computer Science
at the
Laboratoire d'architecture des processeurs (LAP)
of the
School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL)
by

Atri Bhattacharyya

supervised by

Prof. Paolo Ienne, Director, LAP

Lausanne, EPFL, 2018

Acknowledgements

I would like to thank my supervisor, Prof. Paolo Ienne, for the opportunity to do my Master's project in the fascinating field of computer architecture and to be part of the intellectually stimulating and pleasant environment at the Laboratoire d'architecture de processeurs (LAP). His guidance during this project has been invaluable and his insistence on formally expressing concepts has been essential for me to get a clear grip on my own ideas.

My sincerest gratitude also goes out to Lana Josipovic whose doctoral research has been the basis for this project. She been around to help me at every stage of the way by suggesting paths to explore, acting a sounding board for my ideas and by critically reviewing all my work including the dreary exercise of proofreading this document. I would also like to acknowledge Andrea Guerrieri for helping me out with technical problems with the compiler infrastructure that threatened to derail the entire project at one point.

Most of all, I must thank my parents who have supported me through all of my endeavors irrespective of the odds. Without their love and unconditional support, none of this would have been possible. I am also grateful to my brother and cousins who have always been available to pump up drooping spirits. Together, they have been my pillars through thick and thin.

Finally, I would like to thank the staff at LAP and the School of Computer and Communication Sciences for helping me with all the associated formalities and for enabling me to work free of all hassles.

Lausanne, 29 June 2018

Abstract

Dynamically-scheduled elastic circuits generated by High-Level Synthesis (HLS) tools are inherently out-of-order, following the flow of data rather than the evolution of an instruction pointer. Components of the circuit which access memory need to be connected to a Load-Store Queue (LSQ) that dynamically checks for memory dependencies, performs store ordering and forwarding, and allows unordered access to Random-Access Memory (RAM) whenever possible. While connecting every memory access (load/store) component to an LSQ ensures correctness of program execution, the hardware and power cost makes this solution unattractive. Statically ruling out dependencies allows circuits to access memory via lightweight components that use an arbitrator to handle RAM port sharing. Reducing the number of components using the LSQ allows the compiler to generate smaller queues which results in superlinear savings in hardware and power for the memory subsystem.

This work describes additions to the Elastic Compiler (EC) that allow it to analyze algorithms expressed in LLVM-IR, an intermediate code representation, to rule out memory dependencies between load/store instructions and their underlying insights. These analyses leverage pointer analysis as well as array access patterns to narrow down the list of possibly dependent instructions. We also enhance the compiler to leverage our analyses and automatically generate relevant memory-access components for the circuit and to connect them to the relevant arbitrator or LSQ.

Contents

Introduction

A ubiquitous feature of modern computational environments is the functional accelerator. From FPGA filled datacenters [2] for faster web-search to specialized AI coprocessors on hand-held devices, accelerators move the execution of commonly used and compute-intensive applications away from general-purpose processors to specialized processors on dedicated (ASIC) or reconfigurable (FPGA/CGRA) hardware. These accelerators offer faster executions of specific functionality, often under strict power constraints. Other examples of accelerators include cryptographic accelerators which enable fast, efficient and secure communication between arbitrary endpoints and digital signal processors (DSPs) which handle the particularly circumscribed set of processing required for broadband telecommunication.

With the end of Dennard Scaling, power-density constraints have almost halted the traditionally drastic increases in operating frequency between processor generations. No longer can the ballooning computational requirements of modern workloads be met by pushing up core frequencies. Other work has projected that power scaling will also arrest the current trend of scaling up processors by increasing the number of cores [5]. Proposals [8, 16] have called for populating the abundant, power-constrained on-die area with heterogeneous, efficient, application-specific processors.

Developing application specific processors is currently an involved process that requires significant investment in highly-skilled personnel and resources. This leads to a gap between the demand for rapid development of a large variety of application specific hardware and the current capabilities for the same. High-Level-Synthesis (HLS) aims to accelerates this process by automating the generation of hardware from algorithms expressed in high level languages such as C or even functional languages such as Haskell.

Currently available HLS solutions almost universally depend on statically scheduled datapaths. Due to indeterminacy in timing introduced by control decisions based on data from memory accesses, long latency operations or by possible memory dependencies the static schedules generated by HLS tools are overly conservative and restrict parallelism in loops. While dynamic scheduling in elastic circuits removes the necessity to conservatively schedule operations, the inherently out-of-order nature of dataflow circuits warrants the use of bulky load-store-queue (LSQ) interfaces to memory. In accelerators, the ability to disambiguate addresses at compile time can allow the usage of lightweight ports which bypass the power-hungry LSQs and directly access memory.

This work focuses on the problem of statically disambiguating memory accesses in dynamically scheduled elastic circuits in order to exploit the lack of memory dependencies between components. We wish to connect only the minimal set of nodes to LSQs while allowing other provably data-dependency free components to directly access the memory. For this, we exploit known techniques such as alias analysis and develop a novel approach that leverages the determinate nature of Static Control Parts and the properties of data flow through an elastic circuit.

1 Background

1.1 High Level Synthesis

Data-driven architectures are a class of computer architectures where operations are triggered by the availability of data as operands, rather than a requirement for the results [17] as in a traditional von-Neumann architecture. Dataflow circuits are an implementation of a data-driven architecture. They consist of hardware components for various operations which are connected to reflect the natural flow of data in the algorithm. An example of a dataflow circuit is shown in Fig. 1.1.

High-level Synthesis (HLS) is the compilation of algorithms expressed in high-level languages such as C/C++/SystemC or functional languages to digital hardware expressed in a Register-Transfer Level (RTL) language such as Verilog or VHDL. The output, a netlist, can be used by a logic synthesis tool to synthesize a gate-level dataflow circuit that implements the algorithm. This gate-level description may be used by VLSI tools to fabricate ASICs or to program an FPGA.

1.2 Scheduling and Elastic Circuits

Scheduling in a dataflow circuit is the activation of the hardware elements of the circuit to perform their function. Most existing HLS frameworks use static scheduling where the schedule is set at compile time and a central scheduling unit may be responsible for these activations. When control and/or data dependencies are indeterminable at compile time, static schedulers follow conservative schedules based on worst-case assumptions. This is demonstrated in Fig. 1.2, where the static scheduler must assume that the summation operation from the previous iteration might execute, and must delay the current operation accordingly. On the other hand, a dynamic scheduler can execute the current summation earlier, if the previous value is not positive.

Elastic circuits [3] are dynamically-scheduled circuits where each hardware element corre-

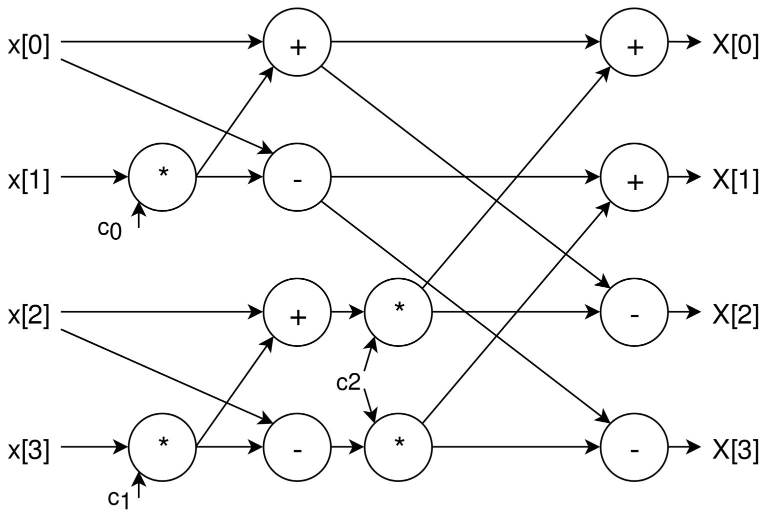


Figure 1.1 – Dataflow circuit for computing the Discrete Fourier Transform of a 4-wide array

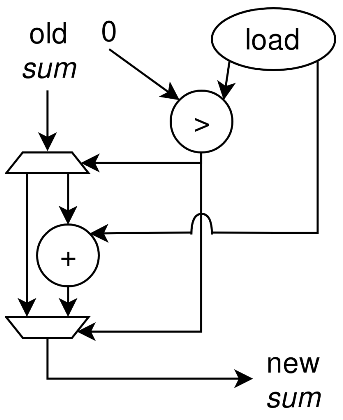
sponds to a node in a dataflow-graph. Nodes are activated dynamically when they are ready (to perform their operation) and their operands are available. Each component is augmented with *ready* and *valid* signals to its predecessor and successor nodes, which forms a handshake mechanism for regulating the flow of data through the circuit. An example elastic node with two predecessors and a single successor is shown in Fig. 1.3. When data passes from one element to the next, a token is said to have passed between them. Unlike statically-scheduled circuits, there is no central scheduling unit activating elements according to some pre-determined schedule. Josipovic et al. [11] propose an algorithm for generating elastic circuits from C code.

1.3 LLVM

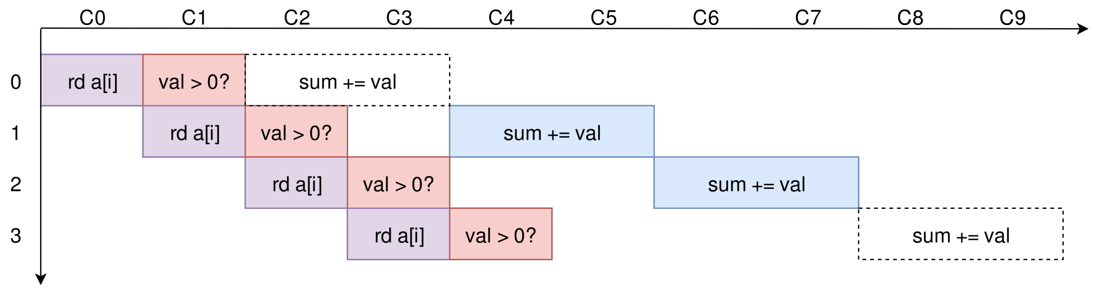
The LLVM Project is a research initiative that was started at the University of Illinois to design a comprehensive, modular compiler framework [12, 13]. It includes compiler frontends for a variety of programming languages including C/C++/Objective-C which compile code to LLVM-IR, an intermediate representation in Static Single Assignment (SSA) form with a simple, language-independent type-system that can cleanly represent code in high-level languages [14]. External projects have extended frontend support to other high-level languages such as Python, Java and Haskell. LLVM-IR is the basis of a number of analysis and transformation passes which respectively aim to generate insights about the code and to optimize it. Several commonly-used passes are included with the project, while most external projects implement passes of their own. LLVM also provides backends for generating static and just-in-time (JIT) code for popular existing architectures such as ARM, x86, and MIPS as well as research architectures such as RISC-V.

```
for(i = 0; i < n; i++){
    val = a[i];
    if(val > 0)
        sum += val;
}
```

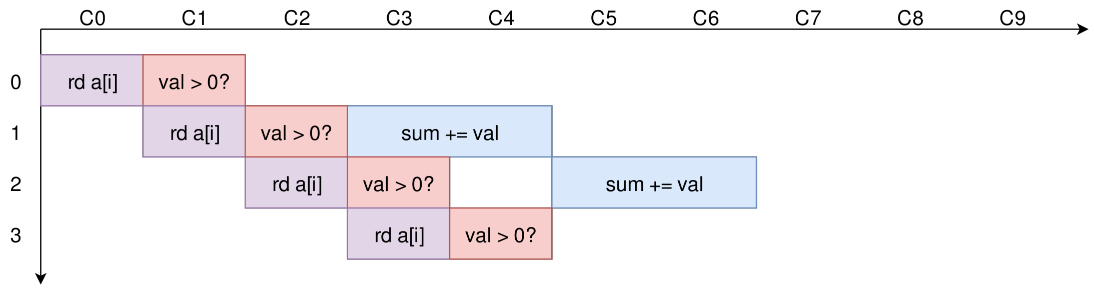
(a) C code



(b) Portion of the corresponding dataflow circuit



(c) Conservative scheduling by static scheduler



(d) Possible optimal schedule by dynamic scheduler

Figure 1.2 – Example code to compute the sum of all positive elements in an array. Fig. 1.2b shows a portion of the corresponding dataflow circuit. A static schedule may produce the schedule shown in Fig. 1.2c. In contrast, a dynamic scheduler may produce the optimal schedule shown in Fig. 1.2d.

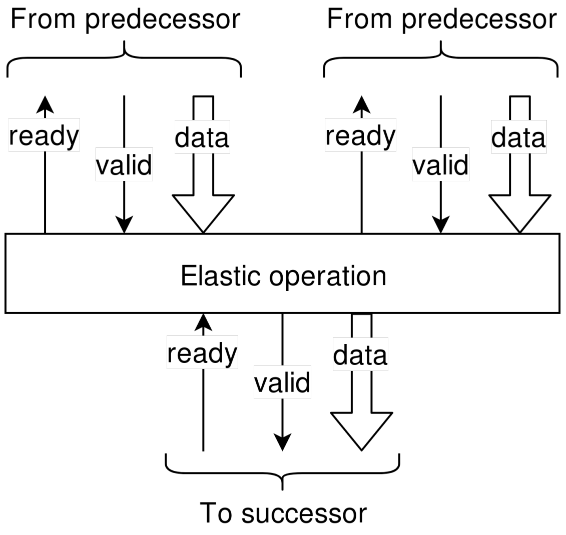


Figure 1.3 – A node in an elastic circuit with *ready* and *valid* signals which regulate the flow of tokens

1.3.1 Pass Infrastructure

LLVM includes a number of analysis and transformation passes which operate on LLVM-IR [15]. These are the basis for LLVM's multi-stage optimization paradigm. Analysis passes generate information about the code which the transformation passes may exploit to generate optimized code. Commonly used analysis passes include:

- Alias Analysis - which checks whether or not two pointers may alias i.e. reference the same location. It is implemented by various modules which provide information at various levels of analysis. For example, BasicAA distinguishes between distinct global, stack and heap allocated variables. It also disambiguates different members of a struct, indices into arrays with statically different subscripts. SteensAA, implements a variation of the Steengaard's Points-To analysis algorithm and is capable of providing inter-procedural alias analysis.

- Loop Information - which generates information about loops such their depth and contained basic blocks. Basic blocks within loops have an associated level (≥ 1) where the outermost loop is at level 1 and each subsequent nesting raises the level by 1.

- Dominator/Post-Dominator Tree - which computes the dominator/post dominator tree[1] by analyzing the control-flow graph (CFG).

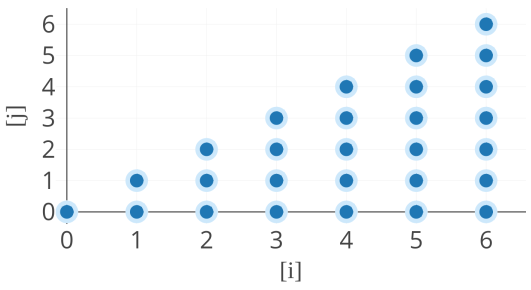
- Region Information - which generates a list of valid regions[2] in the CFG.

[1]https://en.wikipedia.org/wiki/Dominator_(graph_theory)

[2]In LLVM-IR, a region is defined by a single-entry header and a single-exit footer basic blocks. It is a set all basic blocks which are dominated by the header and post-dominated by the footer, excluding the footer.

```
for(i = 0; i < n; i++){
  for(j = 0; j <= i; j++){
    Instruction(i, j)
  }
}
```

(a) An example nested loop code in C



(b) The domain of Instruction(i, j)

Figure 1.4 – The domain for an instruction inside a nested-loop in C is shown to form a polygon in 2-dimensional integer-space

1.4 Integer Polyhedra

An integer polyhedron P is defined as a set of m dimensional vectors of the form $P = \{x \in \mathbb{Z}^m | Ax \leq b\}$ for some matrix $A \in \mathbb{Z}^{m \times n}$ and some vector $b \in \mathbb{Z}^n$. An integer polyhedron is often an apt descriptor for the domains of induction variables[3] for loops in a high-level language. As an example, the induction variables in the code from Fig. 1.4a are bound by the constraints $0 \leq j \leq i < n$. The execution domain for the instruction is described by the set shown below.

$$\left\{[i,j] : \begin{pmatrix} 1 & 0 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} n-1 \\ 0 \\ 0 \end{pmatrix} \right\}$$

1.4.1 Integer Set Library

The Integer Set Library `isl` is a C library for creating and manipulating sets and relations of integer tuples bounded by affine constraints [18]. Among others, `isl` defines data structures representing the following types of objects[4]:

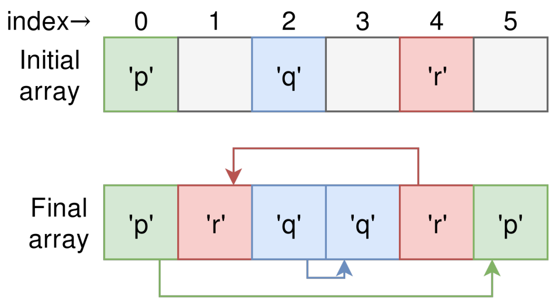
- Basic Integer Set
- Set
- Basic Map
- Map

[3]An induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop. Without loss of generality, this thesis assumes that induction variables start at 0 and increase by 1 every iteration. Ref. Wikipedia

[4]The definitions in this section have been taken from Grosser [7].

```
for(i = 0; i < n; i++){
    if(i % 2 == 0){
        from = i;
        to = n - 1 - i;
        a[to] = a[from];
    }
}
```

(a)



(b) The array, before and after running the code. Arrows show the copying of data.

Figure 1.5 – Code to generate a palindromic array from the elements at even position of an array

Definition 1. *An* `isl` *basic set is a function* $S : \mathbb{Z}^n \to 2^{\mathbb{Z}^d} : s \mapsto S(s)$*, where*

$$S(s) = \left\{x \in \mathbb{Z}^d | \exists z \in \mathbb{Z}^e : Ax + Bs + Dz + c \geq 0\right\}$$

with $A \in \mathbb{Z}^{m\times d}, B \in \mathbb{Z}^{m\times n}, D \in \mathbb{Z}^{m\times e}, c \in \mathbb{Z}^m$.

In the definition, m is the number of constraints on the set, d is the number of set dimensions, n is the number of parameters and e is the number of existentially qualified variables.

An example of a basic set is $[n] \to \{i | \exists e : 0 \leq i < n, i = 2e\}$. This basic set can be used to represent the execution domains for the load and store instructions from the loop in Fig. 1.5a. The induction variable i is used to parse through the even positions in an array of size n. The same set can be represented as:

$$[n] \to \left\{[i] : \begin{pmatrix}1\\-1\\1\\-1\end{pmatrix}\begin{pmatrix}i\end{pmatrix} + \begin{pmatrix}0\\1\\0\\0\end{pmatrix}\begin{pmatrix}n\end{pmatrix} + \begin{pmatrix}0\\0\\-2\\2\end{pmatrix}\begin{pmatrix}e\end{pmatrix} + \begin{pmatrix}0\\-1\\0\\0\end{pmatrix} \geq 0\right\}$$

where the constraints $0 \leq i < n, i = 2e$ are re-arranged to get:

- $i \geq 0$ and
- $n - 1 - i \geq 0$
- $i - 2e \geq 0$
- $-(i - 2e) \geq 0$

To express the same basic set as per definition 1,

- $d = 1$
- $m = 4$
- $n = 1$
- $e = 1$
- $A = \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$
- $B = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$
- $D = \begin{pmatrix} 0 \\ 0 \\ -2 \\ 2 \end{pmatrix}$
- $c = \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \end{pmatrix}$

An `isl` set is a union of a finite number of `isl` basic sets, each of which have the same number of set and parameter dimensions.

Definition 2. *An* `isl` *basic map is a function* $M : \mathbb{Z} \to 2^{\mathbb{Z}^{d1} \times \mathbb{Z}^{d2}} : s \mapsto M(s)$ *where*

$$M(s) = \left\{ x_1 \to x_2 \in \mathbb{Z}^{d1} \times \mathbb{Z}^{d2} | \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + Bs + Dz + c \geq 0 \right\}$$

with $A_1 \in \mathbb{Z}^{m \times d_1}, A_2 \in \mathbb{Z}^{m \times d_2}, B \in \mathbb{Z}^{m \times n}, D \in \mathbb{Z}^{m \times e}, c \in \mathbb{Z}^m$

In the definition, d_1 is the number of input dimensions, d_2 is the number of output dimensions, m is the number of constraints on the set, n is the number of parameters and e is the number of existentially qualified variables.

An example of a basic map is $[n] \to \{i \to o : o = n - 1 - i\}$. This basic map can be used to represent the access relation for the store instruction in the loop from Fig. 1.5a. The location at index $n - 1 - i$ is accessed in the i^{th} iteration. The same map can be represented as

$$[n] \to \left\{ [i] \to [o] : \begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} o \end{pmatrix} + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \begin{pmatrix} n \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \geq 0 \right\}$$

where the constraint $o = n - 1 - i$ is expressed as:

- $i + o - n + 1 \geq 0$ and
- $-(i + o - n + 1) \geq 0$

To express the same basic map as per definition 2,

- $d_1 = 1$
- $d_2 = 1$
- $m = 2$
- $n = 1$
- $e = 0$
- $A_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$
- $A_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$
- $B = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$
- $D = ()$
- $c = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$

An `isl` map is a finite union of basic maps, each of which have the same number of input, output and parameter dimensions.

For the purpose of readability, constraints on `isl` sets and maps will be written as a list for the rest of the document. Syntactic sugar (e.g. 'i mod 2 = 0') will be used where possible.

1.5 Polly

Polly is a framework based on LLVM which uses polyhedral analysis to optimize loops within programs for data locality and parallelism [7]. Implemented as a series of LLVM passes, Polly includes separate modules for preparing code for Polly (canonicalization), detecting SCoPs and creating polyhedral descriptions for memory accesses, optimizing them and re-generating the IR.

Polly performs the following loop canonicalization passes:

- Loop Simplification: This pass ensures that each loop has a pre-header from which there is a single entry-edge to the loop header. It also ensures that the loop has a single latch edge[5] like in Fig.1.6b.
- Induction variable canonicalization: This modifies each loop to have a single induction variable counting from zero with steps of one, like in Fig. 1.6d.
- Tail call elimination: This pass transforms tail-recursive functions into their iterative form. This enables Polly to analyze and optimize loops written in functional programming languages or with functional paradigms in mind like in Fig. 1.6f.

1.5.1 Static Control Parts

Static Control Parts are regions of a program in which all control flow decisions and memory accesses are known at compile time and hence, may be statically scheduled by a control unit. SCoPs must be side-effect free [6] and contained loops must have affine expressions in induction variables and parameters for:

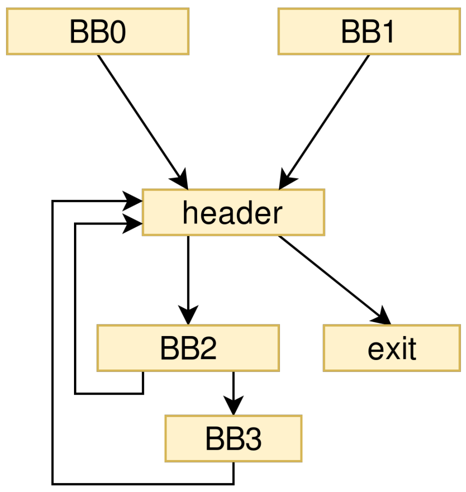
- Loop bounds.
- Flow control conditions.
- Memory access relations.

Examples of loops which are SCoPs can be found in Fig. 1.5a and Fig. 1.8 while loops which violate one or more of these conditions are shown in Fig. 1.7.

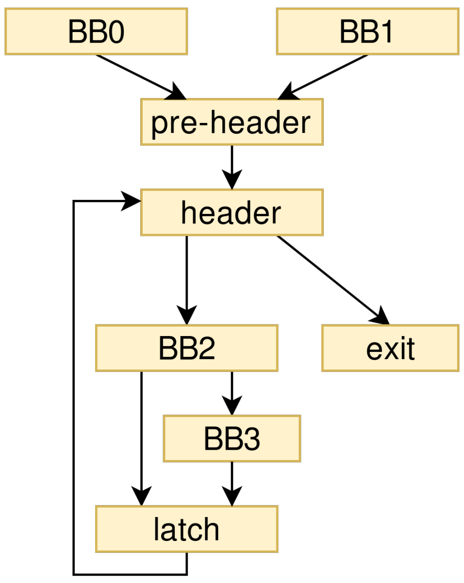
1.5.2 ScopInfo Pass

The pass ScopInfo implemented by Polly detects SCoP regions and creates polyhedral descriptions for memory accesses in them. Grosser [7] defines a SCoP in LLVM-IR and describes the

[5]A latch for a loop is a CFG edge where control moves back to the header.



(a)



(b) Canonicalized CFG for loop in Fig.1.6a.

```
for(i = a; i < b; i += c){
    ...
}
```

(c)

```
for(j = 0; j < (b - a)/c; j++){
    i = a + c * j
    ...
}
```

(d) Canonicalized loop equivalent to Fig.1.6c

```
int factorial(int n){
    if(n < 2)
        return 1;
    else
        return n * factorial(n-1);
}
```

(e)

```
int factorial(int n){
    int product = 1, i;
    for(i = 2; i <= n; i++)
        product *= n;
    return product;
}
```

(f) Iterative equivalent of the recursive function from Fig.1.6e

Figure 1.6 – Examples of loop canonicalization passes used by Polly

```
for(i = 0; i*i < n; i++){
    lock(i);
    a[i] = i;
    unlock();
}
```

(a) Loop has a non-affine upper-bound and function calls with side-effects (modifies a lock variable)

```
while(low < high){
    mid = (low + high)/2;

    if(a[mid] == n)
        ...
    else if(a[mid] < n)
        low = mid + 1;
    else if(a[mid] > n)
        high = mid;
}
```

(b) Loop has data-dependent control decisions and loop bound

Figure 1.7 – Two non-SCoP loop regions. The algorithm in Fig. 1.7a acquires a lock before writing to the array at index i. Fig. 1.7b shows an implementation of the binary search algorithm.

SCoP-detection algorithm. The ScopInfo pass analyzes the LLVM-IR representation allowing detection of regions that are semantically a SCoP, but are not expressed as such. Examples may be seen in Fig. 1.8.

For every valid region, ScopInfo creates a Scop object containing ScopStmt objects corresponding to each contained basic block. A description is generated for each memory access (load/store) in a MemoryAccess object comprising a domain, a schedule, a base and an access relation map.

- The base for a MemoryAccess is a pointer to the base of the array accessed by the instruction.
- For an instruction I nested inside m loops, the domain D for a MemoryAccess is the set of vectors $v_{ind} = \{v_1, \dots v_m\}$ of the values of induction variables for which I will be executed. Each v_i refers to the value of the induction variable for the loop at level i. Therefore, v_1 is the value of the induction variable for the outermost loop and v_m is the value of the induction variable for the innermost loop. Polly describes the domain for an instruction in an `isl` set object. Values from outside the outermost loop which are used in the SCoP are specified as parameters.
- The access relationship for a MemoryAccess is a function that maps a vector of m induction variables to a vector of n indices into the n-dimensional base array. Polly represents the access relationship as an `isl` map object. Values from outside the outermost loop which are used in the SCoP are specified as parameters.

```
i = 0;

do {
    int a = 3 * i;
    int b = n/2 + i + 5 * a;

    arr[b] = i;
    i += c;
} while (i < n);
```

(a) Using a complicated index expression

```
for(i = 0; i == 0 || i < n; i += c){
    arr[n/2 + 16 * i] = i;
}
```

(b) Equivalent code in canonical form

```
int *iter = arr;
int *end = &arr[n];
int count = 0;

while(iter != end) {
    *iter++ = count++;
}
```

(c) Using pointer arithmetic to parse an array

```
for(i = 0; i < n; i++)
    arr[i] = i;
```

(d) Equivalent code in canonical form

Figure 1.8 – Two valid SCoP regions and their canonicalized counterparts

- The schedule is a vector which allows for partial ordering on the set of memory instructions within a SCoP. This information is used by Polly for analyzing memory dependencies and parallelizing loops.

1.6 Elastic Compiler

The Elastic Compiler (EC) under development at EPFL implements the HLS strategy as proposed in Josipovic et al. [11] and is based on the LLVM compiler infrastructure. Starting from the intermediate representation generated by an LLVM frontend (e.g. Clang) and its corresponding CFG, the Elastic Compiler generates a VHDL netlist for a dynamically scheduled circuit. The circuit comprises elements implementing specific basic operations (e.g. arithmetic, branch, select) similar to those in traditional dataflow circuits, but augmented with elastic control signals connecting each element to its predecessors and successors in order to achieve latency-insensitive scheduling. It also includes other elastic components such as elastic buffers, elastic FIFOs, eager and lazy forks and joins. These are described in detail in previous work [4, 9]. Connections to arrays of random-access-memory (RAM) are made through read and write ports that connect to an arbitrator or a load-store-queue. The connection to memory is the focus of this work and is described in more detail in chapter 2.

Elastic sub-circuits are first generated for each basic block by literally translating the dataflow

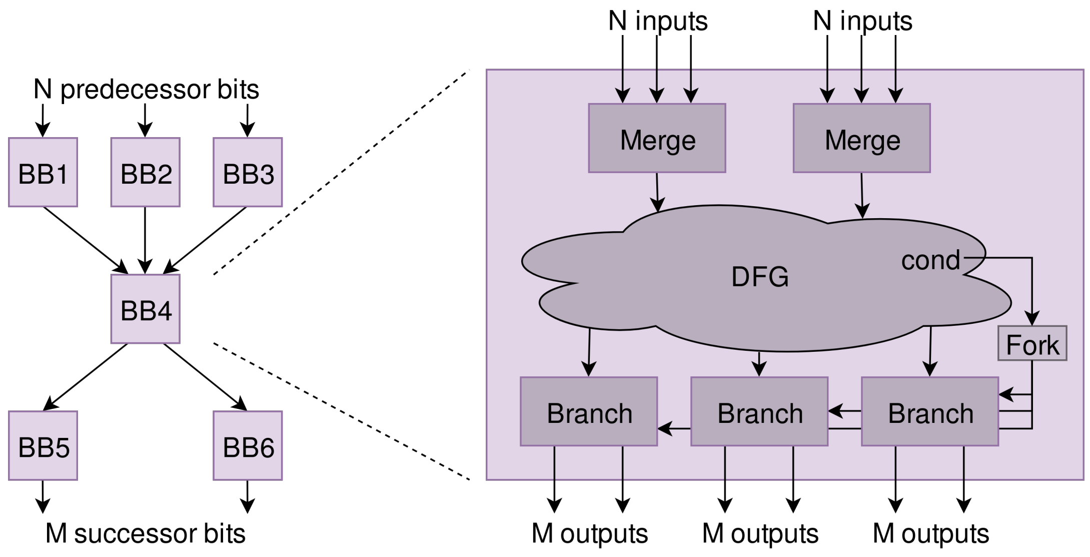


Figure 1.9 – The template for generating elastic basic blocks

graph by replacing operators with their corresponding functional units, connecting components to their predecessors and successors and introducing forks wherever the output of a component has more than one successor (at least one of which is within the same basic block). Control nodes resembling a data-less variable are added for each block. These sub-circuits are then augmented with branch nodes for each live value leaving a basic block and merge nodes for each value entering it. Finally, the sub-circuits are stitched together. For each basic block edge in the CFG, the branch node from the predecessor for each live value is connected to the corresponding merge node in the successor. This is illustrated in Fig. 1.9 reproduced from Jospovic et al. [11]. Other added components include FIFOs to decouple fast paths from slower ones, and buffers to break combinatorial loops or critical edges.

2 Memory Dependencies in Elastic Circuits

Nodes in an elastic circuit are dynamically scheduled when their operands are available. As a result, the temporal ordering between the activations of any arbitrary pair of nodes in the circuit is unclear at compile time and memory accesses are inherently out-of-order. The flow of tokens through the circuit provides us with the only mechanism for temporally ordering the activations of pairs of nodes where one depends on the other for receiving a token, and hence being activated. Nodes corresponding to memory instructions (load/store) may have read-after-write (RAW) and write-after-write (WAW) dependencies. Being out-of-order, these nodes might need a load-store-queue (LSQ) to dynamically resolve memory dependencies and correctly order dependent accesses.

Load-store queues are structures that allow memory operations to be issued to memory out-of-order by checking addresses with all previous operations. For example, a load may be issued to the RAM as soon as all previous store addresses are available to the LSQ and it verifies that none of them write to the same address as the load. For this purpose, it uses Content-Addressable Memory (CAM) which incorporates comparison circuitry with every address storage cell. Since every address must be compared with all previous addresses, the hardware and energy costs of an LSQ increase super-linearly with its size, i.e. the number of outstanding memory operations it can handle.

An LSQ must be sized to handle the latency between a load entering the queue and addresses for all previous stores becoming available. To store all addresses that arrive in this period, the size of its queues must increase with the number of read and write ports. As a result, it is desirable to only connect memory components to the relevant LSQ if there might be RAW/WAW dependencies at runtime.

2.1 Elastic compiler: Memory access

Memory components in the circuits generated by the Elastic Compiler (EC) are connected to arrays of dual-ported random-access memory (RAM). Each RAM corresponds to an array in

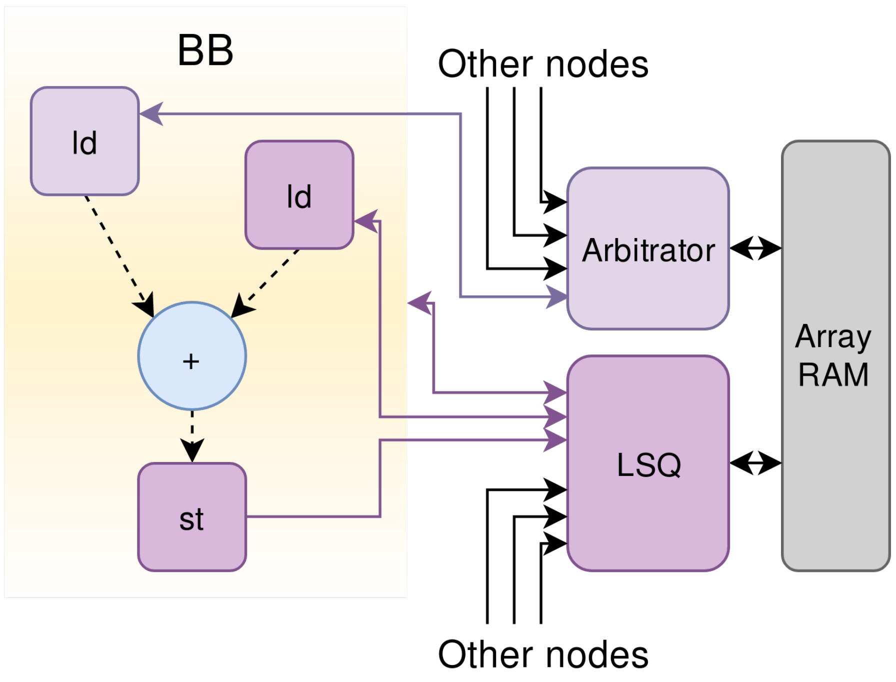


Figure 2.1 – Elastic memory components may be connected to the memory via an arbitrator or an LSQ. The LSQ also has a control connection to the basic blocks from which it has read/write connections

the C-representation of the algorithm. The hardware node for a memory element might be a simple port which connects to an arbitrator that uses one of the RAM ports or a connection to a LSQ that uses the other RAM port.

EC implements LSQs as described in Josipovic et al. [10]. Elastic components corresponding to memory instructions are connected to the relevant LSQ via elastic interfaces which include separate elastic signals for the data and address. An example connection is shown in Fig 2.2. When control reaches a basic block, slots corresponding to the loads and stores in the block are allocated in the LSQ prior to any of those components being activated. In the existing EC compiler, memory nodes need to be manually connected to the memory subsystem. Further, they are all conservatively attached to the LSQ.

In this work, we extend EC by automating and optimizing the assignment of memory nodes to LSQs. We describe the insights which allow the compiler to make the decision to connect to memory via a simple port to the arbitrator instead of the LSQ. Further, it describes the implementation of memory disambiguation passes based on these insights in EC which allows it to automate the process of connecting the elastic circuit to memory. An example of a portion of a possible circuit generated by the final compiler is shown in Fig. 2.1. In this example, the load node connected to the arbitrator must have been proved to be independent of all other nodes accessing the same RAM.

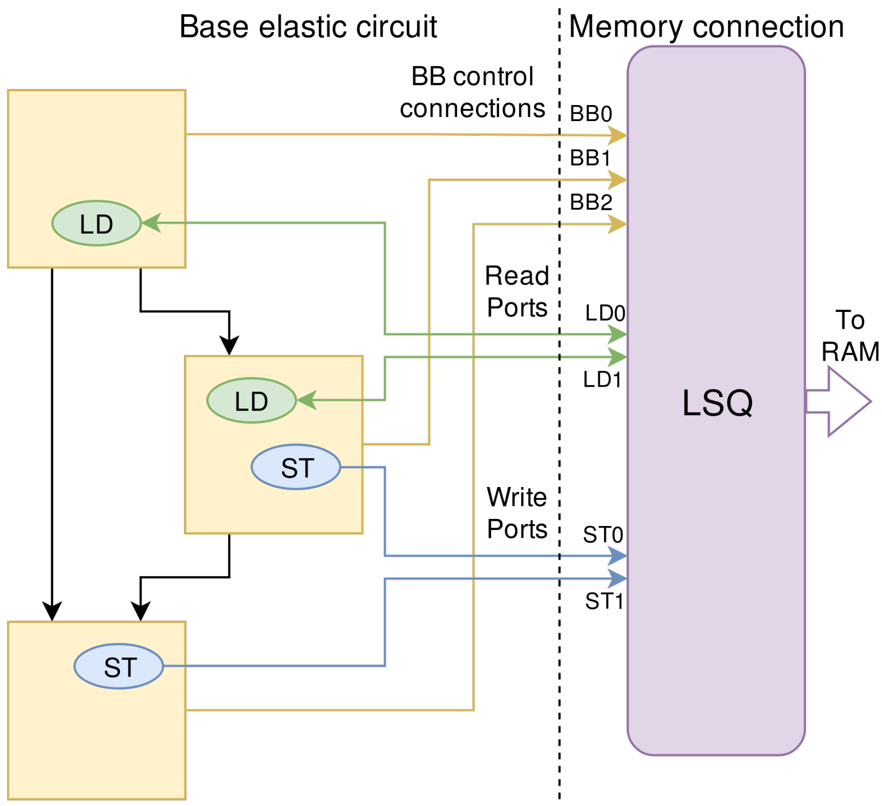


Figure 2.2 – Generated LSQs have connections to elastic read/write ports and to their basic blocks. Yellow blocks represent basic blocks.

In the augmented EC infrastructure, the MemElemPass analysis pass generates a MemElemInfo object containing information used while generating the hardware components for accessing memory. For each node corresponding to a memory instruction, we can query whether it requires an LSQ connection or a basic memory port, and which RAM it needs to access, in order to connect it correctly. Finally, MemElemInfo tracks the necessary control connections between basic blocks and LSQs for slot allocation [10].

2.2 Architecture

The architecture of the memory dependence analysis infrastructure is shown in Fig. 2.3. Nodes indicate analyses while arrows from Node A to Node B indicates that analysis A depends on results produced by analysis B.

Token dependence is defined in Chapter 3 and relates the flow of tokens between pairs of circuit components within a loop. The TokenDependenceInfo class presents the query interface for this analysis.

IndexAnalysis checks for memory dependencies within SCoPs and is described in Chapter 4. It

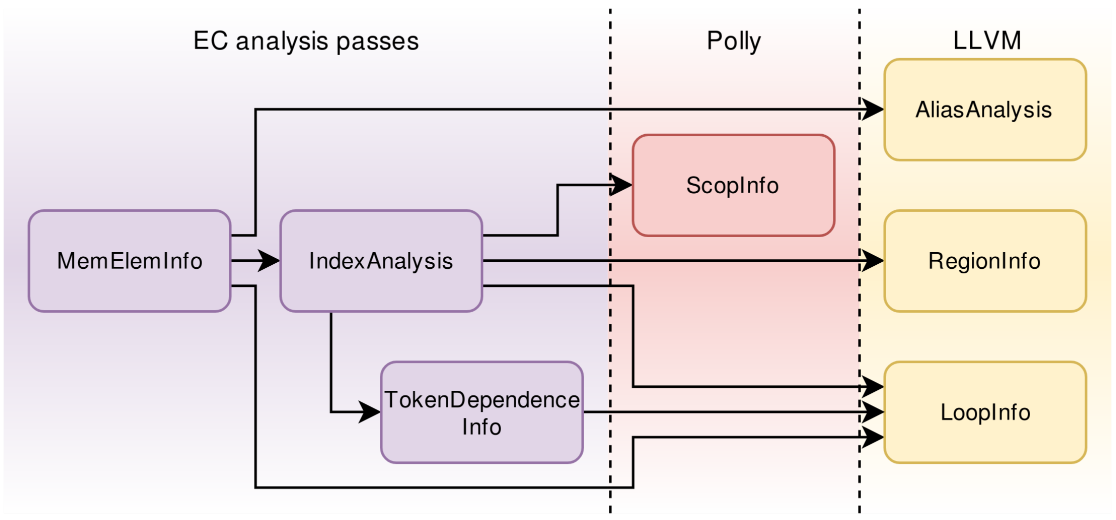


Figure 2.3 – Analysis passes for analyzing memory dependencies between instructions. Arrows indicate dependences between passes.

uses polyhedral information from the ScopInfo wrapper pass to create sets of indices accessed by each instruction and then checks for overlapping sets to detect possible dependencies. It also uses token dependence information from the TokenDependence analysis to temporally relate the activations of memory operations to further rule out dependencies.

MemElemInfo provides the high-level abstraction that the compiler can use to assign hardware components to memory operations. It uses information from IndexAnalysis and AliasAnalysis passes to detect pairs of instructions with memory dependencies and thereby create sets of instructions to be assigned to every LSQ. This pass is described in Chapter 5.

3 TokenDependenceInfo

The TokenDependenceInfo class tries to determine if there is a token dependence or reverse dependence, as defined below, between a pair of instructions by examining the LLVM-IR generated by the compiler frontend.

We define two types of token dependence to describe the flow of tokens between a pair of nodes:

- Token Dependence
- Reverse Token Dependence

Definition 3. *An instruction I_B is dependent on I_A (written as $I_A \xrightarrow{D} I_B$) w.r.t. a set S_{ind} of induction variables if every token arriving at the node corresponding to I_B has passed through the node for I_A without passing though basic block edges that would increment any of the induction variables in S_{ind}.*

Definition 4. *An instruction I_A is reverse dependent on I_B (written as $I_A \xrightarrow{RD} I_B$) w.r.t. a set S_{ind} of induction variables if every token arriving at the node corresponding to I_A will flow to the node for I_B without passing through basic block edges that would increment any of the induction variables in S_{ind}.*

In Fig. 3.1, we see a snippet of C code, its intermediate representation in LLVM and a portion of the elastic circuit generated as a result. From the LLVM code, we can find the following dependences, among others:

- $I_0 \xrightarrow{\mathrm{D}} I_1$
- $I_1 \xrightarrow{\mathrm{D}} I_2$
- $I_2 \xrightarrow{\mathrm{D}} I_3$
- $I_3 \xrightarrow{\mathrm{D}} I_4$
- $I_3 \xrightarrow{\mathrm{D}} I_5$
- $I_3 \xrightarrow{\mathrm{D}} I_8$
- $I_3 \xrightarrow{\mathrm{D}} I_9$
- $I_6 \xrightarrow{\mathrm{D}} I_8$
- $I_7 \xrightarrow{\mathrm{D}} I_8$
- $I_8 \xrightarrow{\mathrm{D}} I_9$

However, we can also determine that the instruction I_6 does not have a token dependence on I_3 because there is a path in which it gets a token from the node with the constant value -1.

Also, there is no token dependence between I_4 and I_5 because there is no path between them without incrementing the induction variable i.

The elastic circuit also has the following reverse dependencies, among others:

- $I_3 \xrightarrow{\text{RD}} I_8$
- $I_3 \xrightarrow{\text{RD}} I_9$
- $I_4 \xrightarrow{\text{RD}} I_6$
- $I_4 \xrightarrow{\text{RD}} I_8$
- $I_4 \xrightarrow{\text{RD}} I_9$
- $I_5 \xrightarrow{\text{RD}} I_7$
- $I_5 \xrightarrow{\text{RD}} I_8$
- $I_5 \xrightarrow{\text{RD}} I_9$
- $I_6 \xrightarrow{\text{RD}} I_8$
- $I_6 \xrightarrow{\text{RD}} I_9$
- $I_7 \xrightarrow{\text{RD}} I_8$
- $I_7 \xrightarrow{\text{RD}} I_9$
- $I_8 \xrightarrow{\text{RD}} I_9$

In contrast, the instruction I_4 does not have a reverse dependence on I_5 since there is no path from the basic-block *if.then* to *if.else* without incrementing the induction variable i. I_4 also does not have a reverse dependence on I_7 since there is a path in which I_7 gets a token from the node with the constant value 1.

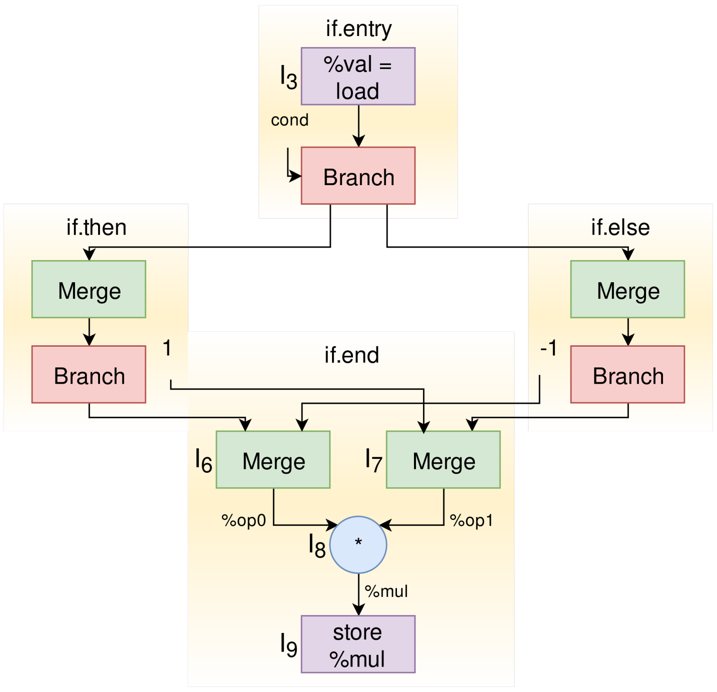
3.1 Properties

1. If $I_A \xrightarrow{\text{D}} I_B$, BB_A dominates[1] BB_B.
2. If $I_A \xrightarrow{\text{RD}} I_B$, BB_B post-dominates[2] BB_A.
3. Both dependence and reverse dependence are transitive, non-symmetric and non-reflexive.
4. If $I_A \xrightarrow{\text{D}} I_B$ w.r.t the set of common induction variables S_{ind}, I_A cannot be within a deeper loop body.
5. If $I_A \xrightarrow{\text{RD}} I_B$ w.r.t the set of common induction variables S_{ind}, I_B cannot be within a deeper loop body.
6. If $I_A \xrightarrow{\text{D}} I_B$ or $I_A \xrightarrow{\text{RD}} I_B$ w.r.t the set of common induction variables S_{ind}, every execution of I_A for iteration vector [3] v will finish before any execution of I_B for the same iteration vector.
7. If $I_A \xrightarrow{\text{D}} I_B$ or $I_A \xrightarrow{\text{RD}} I_B$ w.r.t the set of common induction variables S_{ind}, every execution of I_A for iteration vector $v_A \leq v_0$ will finish before any execution of I_B starts for iteration vector $v_B \geq v_0$.

[1]In control-flow graphs, a basic block BB_A dominates another BB_B if every path from the entry node to BB_B must pass through BB_A.
[2]In control-flow graphs, a basic block BB_B post-dominates another basic block BB_A if every path from BB_A to the exit node must pass through BB_B.
[3]The vector of the values of induction variables in use at a certain time forms the iteration vector. Comparisons of iteration vectors are done lexicographically.

```
for(i = 0; i < n; i++) {
  int val = a[i];
  if(cond){
    op0 = val;
    op1 = 1;
  } else {
    op0 = -1;
    op1 = val;
  }
  a[i] = op0 * op1;
}
```

(a) C code



(b) Portion of the elastic circuit

```
for.head:
  %i = phi i32 [0, %entry], [%i.inc, %if.end]                   ...(I0)
  %cmp = icmp slt i32 %i, %n
  br i1 %cmp, label %if.entry, label %final
if.entry:
  %idx = sext i32 %i to i64                                     ...(I1)
  %ptr = getelementptr inbounds i32, i32* %vla, i64 %idx        ...(I2)
  %val = load i32, i32* %ptr, align 4                           ...(I3)
  br i1 %cond, label %if.then, label %if.else
if.then:
  %0 = %val                                                     ...(I4)
if.else:
  %1 = %val                                                     ...(I5)
if.end:
  %op0 = phi i32 [%0, %if.then], [-1, %if.else]                 ...(I6)
  %op1 = phi i32 [1, %if.then], [%1, %if.else]                  ...(I7)
  %mul = mul nsw i32 %op0, %op1                                 ...(I8)
  store i32 %mul, i32* %ptr, align 4                            ...(I9)
  %i.inc = add nsw i32 %i, 1
  br label %for.head
```

(c) LLVM code

Figure 3.1 – Example code and circuit

3.2 Proofs of properties

1. If $I_A \xrightarrow{\text{D}} I_B$, but BB_A does not dominate BB_B, the token may flow to I_B along a path that does not pass through BB_A. I_B will execute with a token that has not passed through I_A. This violates the definition of token dependence.

2. If $I_A \xrightarrow{\text{RD}} I_B$, but BB_B does not post-dominate BB_A, the token may flow from I_A along a path that does not pass through BB_B. I_A will execute on a token that shall not pass through I_B. This violates the definition of token reverse dependence.

3. The proof is trivial and omitted.

4. If I_A is in a deeper inner-loop, BB_A does not dominate BB_B since the inner-loop might never be entered. This result follows from property 1.

5. If I_B is in a deeper inner-loop, BB_B does not post-dominate BB_A since the inner-loop might never be entered. This result follows from property 2.

6. Given $I_A \xrightarrow{\text{D}} I_B$, every token reaching I_A must have the same iteration vector v for common induction variables. I_B might be within a deeper loop and execute multiple times, but I_A might not by property 4. By the properties of elastic circuits, the deeper loop will be entered after the token flows through I_A. Similarly, given $I_A \xrightarrow{\text{RD}} I_B$, I_A might be in a deeper loop, but I_B might not by property 5. In an elastic circuit, the deeper loop will complete before the token flows to I_B. If neither are in deeper loops, the token flow is obvious. In all cases, the property holds.

7. From property 6, the execution of I_B for iteration vector v_0 is strictly after the execution of I_A for the same iteration vector. Elastic circuits have the property that each node processes all of its tokens in-order. Thus, the node for I_A will process tokens for $v_A < v_0$ before that for v_0 followed by I_B processing the token for v_0 and, finally, tokens for $v_B > v_0$.

3.3 Finding dependence relations

Let us consider instructions I_A (in basic block BB_A) and I_B (in basic block BB_B). Let S_{ind} be the set of induction variables corresponding to loops common to both instructions.

Definition 5. *A valid path P w.r.t. a set of induction variables S_{ind} is a vector of basic blocks $[BB_1, \ldots BB_n]$ such that*

- $\forall i \in \{1, \ldots n-1\}$ BB_i *to* BB_{i+1} *is an edge in the CFG.*
- BB_n *is the latch block*[A] *for the innermost loop whose induction variable is in* S_{ind} *or*
- BB_1 *is the header block for the innermost loop whose induction variable is in* S_{ind}

The definition above assumes a canonicalized CFG where each loop has unique header and latch blocks.

For determining token dependence/reverse-dependence relations, we need to track the flow of tokens along every path that the program may take through the CFG. If we want to check if $I_A \xrightarrow{D} I_B$ holds, we need to find out the various valid paths to the basic block containing I_B and confirm that along each of these paths, I_B directly or indirectly uses the value produced by I_A. Conversely, to explore whether $I_A \xrightarrow{RD} I_B$, we need to find the various valid paths from the basic block containing I_A and confirm that along each of these paths, the value produced by I_A is used by I_B.

The $K-sets$ as defined below track those instructions that produce values on which an instruction I_B directly or indirectly depend. Each sequence of $K-sets$ (K_0, K_1, ..., K_n) is defined for a path $P = [BB_1, \ldots BB_n]$. K_n is defined to contain only I_B. If BB_{i+1} to BB_n are the final $n-i$ blocks in P, K_i for $i < n$ contains those instructions in these basic-blocks on which I_B depends. As we work backwards along a path, moving from BB_{i+1} to BB_i, we add those instructions in BB_i to the set K_{i-1} if it produces a value used in K_i or itself. If BB_A appears at a position $i = i_A$ in the path P, I_B is directly or indirectly dependent on the value produced by I_A iff K_{i_A-1} contains I_A. Therefore, to check for token dependence, this condition needs to be checked for each valid path.

Definition 6. *For each path $P = [BB_1, \ldots BB_n], \forall i < n$ we define the set K_i corresponding to the basic block BB_{i+1} in the path as*

$$\begin{aligned} K_i = & K_{i+1} \\ & \cup \{v | \exists w : w \in BB_i, w \in K_{i+1}, v \in operands(w)\} \\ & \cup \{v | \exists w : w \in BB_i, w \in K_i, v \in operands(w)\} \end{aligned}$$

For phi instructions, only the operand corresponding to the previous basic block BB_{i-1} is considered. The set K_n is specially defined.

Similarly, the $M-sets$ as defined below track those instructions that use the value produced by I_A directly or indirectly. Each sequence of $M-sets$ (M_0, M_1, ..., M_n) is defined for a path $P = [BB_1, \ldots BB_n]$. M_0 is defined to contain only I_A. If BB_0 to BB_i are the first i blocks in P, M_i for $i > 0$ contains those instructions in these basic-blocks which depend on I_A. As we work our way along a path, moving from BB_i to BB_{i+1}, we add those instructions in BB_{i+1} to the set M_{i+1} which use values produced by instructions in M_i or itself. If BB_B occurs at position $i = i_B$ in the path P, I_B is directly or indirectly dependent on the value produced by I_A iff M_{i_B} contains I_B. Therefore, to check for reverse token dependence, this condition needs to be checked for each valid path.

Definition 7. *For each path $P = [BB_1, \ldots BB_n], \forall i > 0$ we define the set M_i corresponding to the*

[4]The latch block for a latch is the block containing the branch to the header.

basic block BB_i in the path as

$$\begin{aligned}M_i = &M_{i-1}\\ &\cup \{v \mid \exists w : v \in BB_i, w \in M_{i-1}, w \in operands(v)\}\\ &\cup \{v \mid \exists w : v \in BB_i, w \in M_i, w \in operands(v)\}\end{aligned}$$

For phi instructions, only the operand corresponding to the previous basic block BB_{i-1} is considered. The set M_0 is specially defined.

3.3.1 Dependence

For $K_n = \{I_B\}$, $I_A \xrightarrow{D} I_B$ if $\forall$ valid paths P ending in BB_B,

- P contains BB_A at least once, and
- If BB_A first occurs at index i in P, K_{i-1} contains I_A.

Examples

Consider the example code in Fig. 3.1. Let us see if $I_3 \xrightarrow{D} I_9$. The two valid paths ending in the basic block for I_9, *if.end*, are P_0 = [*for.head*, *if.entry*, *if.then*, *if.end*] and P_1 = [*for.head*, *if.entry*, *if.else*, *if.end*]. The basic block containing I_3 is *if.entry*.

- For P_0, the K-sets are shown in Fig. 3.2a. For K_3 the previous basic block is *if.then*, and only operand %0 is considered for I_6 and the constant 1 for I_7. It can be seen that the basic block *if.entry* occurs at index 2 and $I_3 \in K_1$. For this path, I_3 produces a value indirectly used by I_9.
- For P_1, the K-sets are shown in Fig. 3.2b. For K_3 the previous basic block is *if.else*, and only operand %1 is considered for I_7 and the constant -1 for I_6. It can be seen that the basic block *if.entry* occurs at index 2, and $I_3 \in K_1$. For this path as well, I_3 produces a value indirectly used by I_9.

It can be seen that the conditions hold for both paths, implying $I_3 \xrightarrow{D} I_9$.

Now, let us see if $I_3 \xrightarrow{D} I_6$. The two valid paths are as in the previous example.

- For P_0, the K-sets are shown in Fig. 3.3a. For K_3, the previous basic block is *if.then* and only operand %0 is considered for I_6. The conditions hold for this path. For this path, I_3 produces a value indirectly used by I_6.

- $K_4 = \{I_9\}$
- $K_3 = \{I_9, I_8, I_7, I_6, I_4, I_2\}$
- $K_2 = \{I_9, I_8, I_7, I_6, I_4, I_3, I_2\}$
- $K_1 = \{I_9, I_8, I_7, I_6, I_4, I_3, I_2, I_1, I_0\}$

(a) K-sets for path P_0

- $K_4 = \{I_9\}$
- $K_3 = \{I_9, I_8, I_7, I_6, I_5, I_2\}$
- $K_2 = \{I_9, I_8, I_7, I_6, I_5, I_3, I_2\}$
- $K_1 = \{I_9, I_8, I_7, I_6, I_5, I_3, I_2, I_1, I_0\}$

(b) K-sets for path P_1

Figure 3.2 – K-sets for determining if $I_3 \xrightarrow{D} I_9$ in Fig. 3.1c

- $K_4 = \{I_6\}$
- $K_3 = \{I_6, I_4\}$
- $K_2 = \{I_6, I_4, I_3\}$
- $K_1 = \{I_6, I_4, I_3, I_2, I_1, I_0\}$

(a) K-sets for path P_0

- $K_4 = \{I_6\}$
- $K_3 = \{I_6\}$
- $K_2 = \{I_6\}$
- $K_1 = \{I_6\}$

(b) K-sets for path P_1

Figure 3.3 – K-sets for determining if $I_3 \xrightarrow{D} I_6$ in Fig. 3.1c

- For P_1, the K-sets are shown in Fig. 3.3b. For K_3, the previous basic block is *if.else* and only the constant -1 is considered for I_6. The basic block *if.entry* occurs once at index 2 and $I_3 \notin K_1$, so the conditions do not hold for this path. For this path, I_3 does not produce a value used by I_9.

The conditions do not hold for path P_1. Therefore, there is no dependence.

3.3.2 Reverse Dependence

For $M_0 = \{I_A\}$, $I_A \xrightarrow{RD} I_B$ if $\forall$ valid paths P starting in BB_A,

- P contains BB_B at-least once
- If BB_B last occurs at index i in P, M_i contains I_B.

Examples

Consider the example code in Fig. 3.1. Let us see if $I_3 \xrightarrow{RD} I_9$. The two valid paths starting in the basic block for I_3, *if.entry*, are P_0 = [if.entry, if.then, if.end] and P_1 = [if.entry, if.else, if.end]. The basic block containing I_9 is *if.end.*

- For P_0, the M-sets are shown in Fig. 3.4a. It can be seen that the basic block *if.end* appears at index 3, and $I_9 \in M_3$. For this path, I_3 produces a value indirectly used by I_9.

- $M_0 = \{I_3\}$
- $M_1 = \{I_3\}$
- $M_2 = \{I_3, I_4\}$
- $M_3 = \{I_3, I_4, I_6, I_8, I_9\}$

(a) M-sets for path P_0

- $M_0 = \{I_3\}$
- $M_1 = \{I_3\}$
- $M_2 = \{I_3, I_5\}$
- $M_3 = \{I_3, I_5, I_7, I_8, I_9\}$

(b) M-sets for path P_1

Figure 3.4 – M-sets for determining if $I_3 \xrightarrow{\text{RD}} I_9$/ $I_3 \xrightarrow{\text{RD}} I_6$ in Fig. 3.1c

- For P_1, the M-sets are shown in Fig. 3.4b. It can be seen that the basic block *if.end* appears at index 3, and $I_9 \in M_3$. For this path as well, I_3 produces a value indirectly used by I_9.

Since the conditions hold for all paths, $I_3 \xrightarrow{\text{RD}} I_9$.

Now, let us see if $I_3 \xrightarrow{\text{RD}} I_6$. The two valid paths and their M-sets are as in the previous example. We can see that the conditions hold on path P_0 but not on path $P_1 (I_6 \notin M_3)$, implying that there is no reverse dependency.

4 IndexAnalysisInfo

IndexAnalysisInfo checks for memory dependences between memory accesses in a SCoP. Implemented as a LLVM FunctionPass (IndexAnalysisPass), this EC component analyzes the indices within an array accessed by load/store instructions to decide if the same location may be accessed by different instructions.

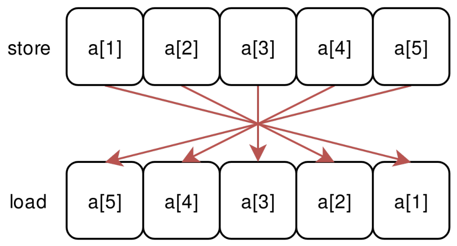
In the example from Fig. 4.1, the load instruction accesses the first half of the string, while the store accesses the second half of the string. In other words, the set of accessed indices by the load is $\{0, \ldots \lfloor n/2 \rfloor - 1\}$ while that for the store is $\{\lceil n/2 \rceil, \ldots n-1\}$. It can be seen that the above sets are disjoint, ruling out the possibility of a RAW dependency.

When a pair of instructions access memory locations in the same array, an analysis of accessed indices may reveal that they cannot access the same address. This is the insight behind IndexAnalysisInfo as a memory disambiguation pass.

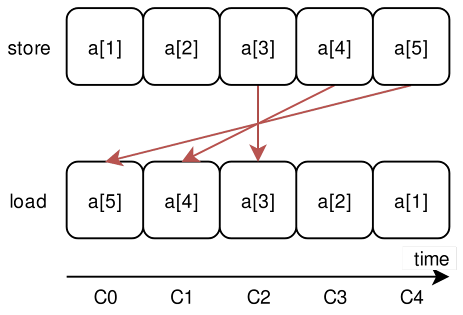
It uses TokenDependenceInfo as described in chapter 3 to determine if pairs of instructions have token dependences or reverse token dependences. Either dependence can be used to temporally relate activations of memory nodes and thereby rule out certain memory dependencies. Consider an example where stores to an array at indices 1 to 5 precede loads to the same. The accesses are demonstrated in the example shown in Fig. 4.2. All the stores should complete before the loads. Without timing information, we cannot be sure that any of the stores actually happens before the load to the same address. Therefore, all the accesses might violate memory dependencies as shown in Fig. 4.2. With cycle information as shown in

```c
int n = strlen(str);
for(i = 0; i < n/2; i++){
    str[n-1-i] = str[i];
}
```

Figure 4.1 – Example C code to create a palindrome from the first half of the character array *str*



(a) Without timing information, we cannot know if the write actually precedes the read to the same index. In this example, every read may potentially violate RAW dependencies.



(b) With timing information, we can see that the last two reads definitely respect RAW dependencies.

Figure 4.2 – Violations of read-after-write (RAW) dependencies, indicated by arrows, between instructions accessing the array a where the store programmatically precedes the load.

Fig. 4.2b, we can see that the accesses for indices 1 and 2 are properly ordered while those for indices 3, 4 and 5 might not be.

4.1 Information from Polly

The IndexAnalysisInfo pass requires precise information about the index within an array accessed by each memory instruction. For SCoPs, Polly is able to provide this information from its ScopInfoWrapperPass. This pass generates *Scop* objects containing domain sets(Dom), bases($Base$) and access relations(AR) for each memory access instruction in the SCoP.

For the example in Fig. 4.1, Polly gives us the following information:

- For the load:
  - Domain: $[n] \rightarrow \{[i] : 0 \leq i < floor(n/2)\}$
  - Base: Array *str*
  - Access relation: $[n] \rightarrow \{[i] \rightarrow [o] : o = i\}$
- For the store:
  - Domain: $[n] \rightarrow \{[i] : 0 \leq i < floor(n/2)\}$
  - Base: Array *str*
  - Access relation: $[n] \rightarrow \{[i] \rightarrow [o] : o = n - 1 - i\}$

4.2 Base algorithm

IndexAnalysisInfo analyzes pairs (I_A, I_B) of instructions within a SCoP. For RAW/WAW dependences to exist between the statements:

- at least one of them must write to memory,
- both instructions must access the same array which is stored in a unique RAM structure in the circuit, and
- there must exist vectors of induction variables v_A, v_B and a vector of array indices idx such that $v_A \in Dom_A$, $v_B \in Dom_B$, $(v_A \rightarrow idx) \in AR_A$ and $(v_B \rightarrow idx) \in AR_B$.

For pairs of instructions that satisfy the first two conditions, IndexAnalysisInfo creates the sets $\{o | \exists i : i \in Dom, i \rightarrow o \in AR\}$ for both. If these sets are disjoint, there are no memory dependencies between them.

4.3 Exploiting Token Dependence

In section 4.2, we discussed the algorithm allowing us to statically analyze memory accesses for memory dependencies within a SCoP. However, in certain cases, the properties of token flow in elastic circuits as discussed in Chapter 3 allow us to put restrictions between the iteration vectors being processed by a pair memory nodes. These additional restrictions might allow us to rule out certain dependencies as illustrated in Fig. 4.2. When we can statically determine that the read and store to an address must happen in the same order in an elastic circuit as in the LLVM code, we do not need an LSQ to order them.

If I_A is a load instruction, I_B is a store instruction currently executing iteration vector v_B and S_{ind} is the set of induction variables for loops common to both instructions, property 7 from section 3.1 assures us that every execution of I_A for iteration vectors $v_A < v_B$ will have completed if $I_A \xrightarrow{\text{D}} I_B$ or $I_A \xrightarrow{\text{RD}} I_B$ w.r.t S_{ind}. We only need to check if the store for iteration vector v_B and the load for lexicographically larger iteration vectors can access the same array index. Since we cannot determine the temporal ordering between them, RAW dependency violations are possible and we need to connect them to a LSQ.

Consider the example in Fig. 4.3 in which the elements of an array are shifted forward by one position. Based on SCoP analysis, we know that the set of indices accessed by the load is {1, 2, ..., $n-1$} while the set of indices accessed by the store is {0, 1, ..., $n-2$}. Since the same indices are accessed by both instructions, RAW dependencies between the load and the store seem possible. However, token dependence implies that, in an elastic circuit, the store to any index must happen after the load to the same index, as explained hereon. When the store is storing to the index i_B in iteration $i = i_B$, the load for all iterations $i \le i_B$ must have finished. Possible read indices in the future are to index $i_A + 1$ for iterations $i_A > i_B$. The store to i_B cannot be read in the future which proves that RAW dependences are practically impossible in this case.

```
for(i = 0; i < n - 1; i ++)
    arr[i] = arr[i + 1];
```

Figure 4.3 – Example of a loop with intersecting read and write sets. Temporally ordering the operations by token dependence shows the lack of RAW dependencies.

4.3.1 Algorithm for dependent instructions

Suppose I_A is a load instruction and I_B is a store instruction such that $I_A \xrightarrow{\text{D}} I_B$ or $I_A \xrightarrow{\text{RD}} I_B$ and S_{ind} is the set of induction variables for loops common to both instructions. For an iteration vector v, let $common(v)$ be the vector of values from v corresponding to members of S_{ind}.

Definition 8. *For a load instruction, the* Future load set *is defined as:*

$$\left\{ i \to o \middle| \exists i_{ld}, i'_{ld} : \begin{array}{c} i_{ld} \in Dom_A, i_{ld} \to o \in AR_A, \\ common(i'_{ld}) < common(i_{ld}), \\ i = common(i'_{ld}) \end{array} \right\}$$

Definition 9. *For a store instruction, the* Store set *is defined as:*

$$\left\{ i \to o \middle| \exists i_{st} : \begin{array}{c} i_{st} \in Dom_B, i_{st} \to o \in AR_B, \\ i = common(i_{st}) \end{array} \right\}$$

Suppose $(i_0 \to o_0)$ is an element of both sets. By membership in the store set, there exists a iteration i_{st} s.t. $common(i_{st}) = i_0$ that accesses o_0. By membership in the future load set, there exists a later iteration i_{ld} for which the load instruction accesses the array index o_0 and $common(i_{st}) < common(i_{ld})$. A RAW dependence exists and since token flow cannot order these accesses, incorrect execution is possible in the absence of an LSQ.

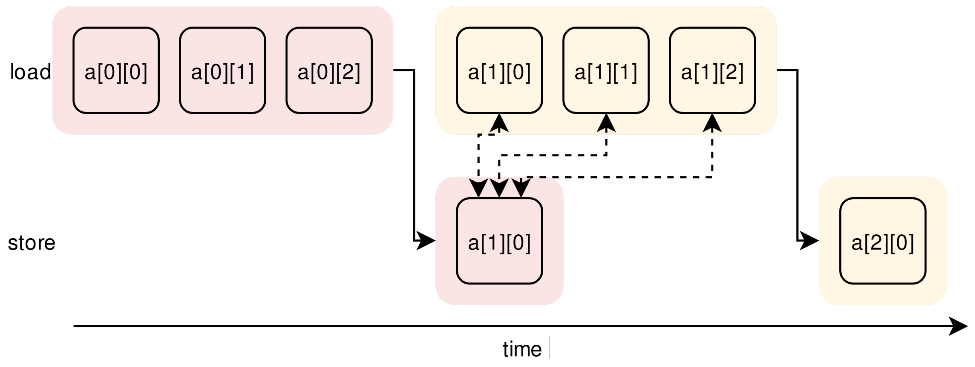
Congruently, suppose there is a RAW dependency between the iteration i_{ld} of the load and the iteration i_{st} of the store, both of which access the same index o_0. It must be that $i_{ld} \in Dom_A$, $(i_{ld} \to o_0) \in AR_A$, $i_{st} \in Dom_B$, $(i_{st} \to o_0) \in AR_B$. Token dependence implies that $common(i_{st}) < common(i_{ld})$. Therefore, $(i_{st} \to o_0)$ will be a member of both sets and the intersection of the sets cannot be empty.

Let us follow along with the example code from Fig. 4.4 where a is a 3×3 array of integers. There is a reverse dependency from the load to the store.

- For the load
  - Domain: $[] \to \{[0,0],[0,1],[0,2],[1,0],[1,1],[1,2]\}$
  - Access Relation: $[] \to \{[i,j] \to [i,j]\}$

```
for(i = 0; i < 2; i++) {
  sum = 0;
  for(j = 0; j < 3; j++)
    sum += a[i][j];
  a[i + 1][0] = sum;
}
```

(a) Example code for adding rows of an array, storing the sum in the first space of the next row.



(b) Temporal ordering between memory accesses. Solid links indicate known temporal ordering due to token flow. Dotted links indicate unknown temporal ordering

Figure 4.4 – Illustrative example of code with token reverse dependence and showing temporal ordering relations

  - Future Load Set: $\{[0] \to [1,0], [0] \to [1,1], [0] \to [1,2]\}$

- For the store

  - Domain: $[] \to \{[0],[1]\}$
  - Access Relation: $[] \to \{[i] \to [i+1,0]\}$
  - Store Set: $\{[0] \to [1,0], [1] \to [2,0]\}$

The timing relations due to token flow can be seen in Fig. 4.4b. The timing between the load for iteration $[1,0]$ and the store for iteration $[0]$, both of which access the index $[1,0]$ in the array, cannot be determined. Hence, these instructions have a RAW dependency which may be violated if they use simple memory ports. As expected, the store and future load sets are non-disjoint.

5 MemElemInfo

MemElemInfo is a LLVM FunctionPass used by EC to make decisions while generating hardware components for nodes which access memory.

For loops, it uses IndexAnalysisInfo and AliasAnalysis to decide whether pairs of instructions may have memory dependencies. It is designed to be extensible i.e. to be able to consider more sub-analyses that can rule out dependencies for other special cases. Currently, it uses the dependency decision for IndexAnalysisInfo for a pair of instructions from the same SCoP. For all other pairs, it uses aliasing information.

After it generates a comprehensive list of such pairs, it creates sets of instructions accessing the same array base which require a LSQ.

The known shortcomings of the MemElemInfo analysis are:

- Since non-loop instructions do not practically affect the depth of the load-store-queue, all memory accesses for these instructions are assigned to a LSQ. However, we could use ports if token flow serializes these accesses.
- To limit the scope of other instructions with which an instruction may alias, MemElemInfo assumes serialization of top-level loops. This may lead to fewer components being connected to LSQs at the cost of increased execution time if the loops could significantly overlap at runtime. This presents a trade-off in the design space between performance and hardware/power costs.

6 Results

In this section, we show examples of code for which we have used the Elastic Compiler to generate VHDL circuits. We focus on three example kernels that demonstrate the utility of our insights and uses the passes discussed earlier to optimize the connections to LSQs required. For all examples, we use Clang as a frontend to compile the C-code to LLVM-IR without any optimization (*-o0* option). Next, they are run though some standard LLVM optimization passes to propagate constants (*constprop*), use registers instead of stack memory to pass values through the circuit (*mem2reg*), eliminate dead instructions (*die*) and simplify the CFG to remove trivial branches (*simplifycfg*). Finally, we use the EC to generate netlists in VHDL. We use this netlist to find the number of load/store ports used and their connections to LSQs, and to evaluate the cost of connecting to memory.

6.1 Methodology

We shall demonstrate the utility of each of our insights towards optimizing the generated circuit's interface to memory, specifically the size of the LSQs, by running the Elastic Compiler on three code kernels. Each of these kernels demonstrates the utility of a separate part of the analysis architecture.

As discussed earlier, the size of the queues of an LSQ are directly related to the number of read and write ports connected to it. We currently have an implementation that automates the process of compilation upto generating the required VHDL netlist. However, we would also need to place and route our designs onto an FPGA using a tool such as Vivado to be able to accurately measure the power and resource requirements for the LSQ. As an equivalent, but approximate metric, we shall estimate the cost of an LSQ as the square of the sum of number of connected read and write ports.

For each code kernel, we shall compare the following cases:

- All memory components are connected to a single LSQ. We designate this as the base case.
- We use AliasAnalysis to distinguish accesses to different arrays, and use different queues for each. We refer to this case as AA.
- We use basic IndexAnalysis without TokenDependenceInfo to analyse accesses within an array. We shall refer to this case as IA.
- We use IndexAnalysis with ordering information from TokenDependenceInfo. We refer to this case as IA+TD.

Each case incorporates and improves upon the results of the previous case. Our implementation in the Elastic Compiler corresponds to the final case listed above (i.e. *IA+TD*).

6.2 Test Cases

Here, we describe the test cases and the number of ports to LSQs at each step of the analysis. The final costs are summarized in Table 6.1. We can see that in these examples, our analysis results in LSQs that cost between 75% and 93% lesser than the base case.

6.2.1 Histogram kernel

The first kernel we analyze is shown in Fig. 6.1. The algorithm calculates an histogram with associated weights for every element. It is assumed that the elements of the feature vector lie in the $[0, n)$ range. As can be seen, there are three load operations and a single store. AliasAnalysis allows us to determine that the loads to the *feature* and *weight* arrays do not need connections to LSQs. Only the *hist* array requires an LSQ with one read and one write port. As the loop is not a Static Control Part, IndexAnalysis analysis is not possible.

Kernel	Number of Load Ports	Number of Store Ports	Base Case	AA	IA	IA+TD
Histogram	1	3	$(1+3)^2$	$(1+1)^2$	4	4
Pivot	1	3	$(1+3)^2$	$(1+2)^2$	$(1+1)^2$	4
Image Processing	18	1	$(18+1)^2$	$(9+1)^2$	100	$(4+1)^2$

Table 6.1 – Cost of LSQs connecting the generated circuit to memory, shown for four degrees of optimization.

```
void histogram(int *feature, float *weight, float *hist, int n)
{
  for (int i = 0; i < n; ++i) {
    int m = feature[i];
    float wt = weight[i];
    float x = hist[m];
    hist[m] = x + wt;
  }
}
```

Figure 6.1 – Code for calculating a weighted-histogram expressed in C

```
void pivot (int x[], int a[], int n, int k) {
  for(int i = k + 1; i <= n; ++i)
    x[k] = x[k] - a[i] * x[i];
}
```

Figure 6.2 – Code for pivoting a vector at position k expressed in C

6.2.2 Pivot kernel

In the pivot kernel, an n-dimensional vector is pivoted at position k. The code for this kernel is shown in Fig. 6.2 The generated circuit uses three read ports and one store port. In the base case, all of them are connected to the same LSQ. AliasAnalysis allows us to determine that the load to the array *a* does not need to be compared to the accesses to array *x*. The LSQ only connects to two read ports and one write ports in this case. Further, IndexAnalysis allows us to determine that the load and store to $x[k]$ cannot access the same location as $x[i]$ as i iterates in the range $[k+1, n)$. Thus, the LSQ finally connects to one write port and one read port. In this case, IndexAnalysis with TokenDependenceInfo does not lead to any further benefits.

6.2.3 Image processing kernels

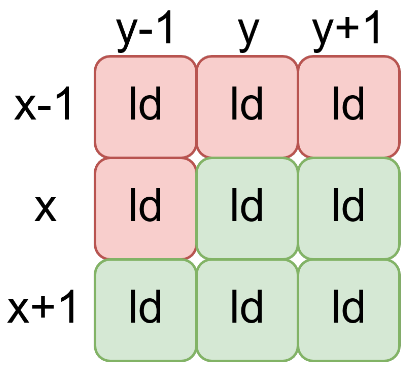
The kernel shown in Fig. 6.3 is used for implementing a number of image processing algorithms including *blur*, *emboss* and *sharpen*. Essentially, the value of each pixel is updated to be the weighted-mean of its surrounding pixels. These weights are stored in the *weight* array. The *multiplier* variable allows us to express the weights as integers and avoid floating point calculations. For example, the *weight* matrix for the *blur* kernel is $\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$ and the *multiplier* variable is 16. We also use an approximate algorithm that performs updates in-situ, trading off accuracy for memory space.

This algorithm requires 18 read elements and one write element. In the base case, they are all connected to a single large LSQ. AliasAnalysis allows us to determine that the accesses to the array *weight* do not conflict with those to *pic*. Further, as they are all loads, the accesses to the

```c
void process(int **pic, int **weight, int n)
{
  for(int x = 1; x <= n; ++x){
    for(int y = 1; y <= n; ++y){
      int sum = 0;
      sum += pic[x-1][y-1] * weight[0][0];
      sum += pic[x-1][y]   * weight[0][1];
      sum += pic[x-1][y+1] * weight[0][2];
      sum += pic[x][y-1]   * weight[1][0];
      sum += pic[x][y]     * weight[1][1];
      sum += pic[x][y+1]   * weight[1][2];
      sum += pic[x+1][y-1] * weight[2][0];
      sum += pic[x+1][y]   * weight[2][1];
      sum += pic[x+1][y+1] * weight[2][2];

      sum /= multiplier;
      pic[x][y] = sum;
    }
  }
}
```

(a) The code for image processing kernels expressed in C.



(b) Loads that need connections to the LSQ with the store shown in red. The other loads are shown in green.

Figure 6.3 – Test case: image processing

array *weight* does not require an LSQ. At this point, there is one LSQ for *pic* with nine read ports and one store port. The basic IndexAnalysis does not improve this result. However, the store has a token dependence on each of the loads. Incorporating this, we see that only four of the loads (shown in Fig. 6.3b) to *pic* need to share the LSQ with the store. The final LSQ design has four read ports and one write port for a cost that is 93% lower than the base case.

7 Conclusion

As accelerators find their way into diverse computing environments, HLS tools endeavor to streamline their development cycle and create a future where hardware development is as mainstream and simple as that for software. As these tools move away from the statically scheduled paradigm, characterized by the necessity to make all decisions at compile time leading to suboptimal performance, it faces the same requirement that Out-of-Order processors have with regards to memory access. Namely, it requires a power-intensive Load-Store Queue (LSQ) to dynamically order dependent instructions. Required to use the same memory units for all memory operations, OoO processors are unable to benefit from insights gleaned by the compiler regarding independence of certain accesses. They can only benefit by reordering instructions to exploit memory-level parallelism or to mask latency from cache misses. All accesses continue to use the LSQ which unnecessarily consumes power. Dynamically scheduled HLS, however, can generate circuits that exploit these insights directly by connecting certain memory operations to memory bypassing the LSQ.

In this work, we have shown a methodology for generating optimized memory access components for elastic circuits. Generated circuits have separate LSQs per array, removing address comparisons between accesses to different arrays. We also remove comparisons between instructions which access the same array, but never access the same indices in the array. Finally, we can detect when the characteristics of data flow in an elastic circuit restrict memory operations to occur in program order, which trivially removes the necessity for connections to the relevant LSQ. We have implemented the aforementioned analyses and added them to the Elastic Compiler infrastructure. We have also automated the process of generating memory access elements using the results from these analysis passes. Finally, we have used the improved Elastic Compiler to generate VHDL descriptions of circuits corresponding to three test cases with LSQs that cost 75% to 93% less than the trivial approach.

In the future, we plan to further optimize the compiler design and improve our disambiguation analyses. We would like to improve the MemElemInfo analysis to allow top-level loops to run in parallel, as described in Chapter 5. Resolving the trade-off involved may require heuristics, including determination of the degree of possible overlap between loops, and whether this overlap can significantly benefit performance. We would also like to be able to use the polyhedral model to analyze loops which are not Static Control Parts (SCoPs). This has been proposed in Benabderrahmane et al. [1] and would allow us to broaden the scope of analyzed loops to include those with data-dependent control flow. Although opportunity remains for other analyses which would allow us to further optimize our memory subsystem, we believe that the work presented in this thesis is a step in the right direction towards making HLS an efficient, performant option.

A TokenDependenceInfo API

- `tokenDepends`
  - Arguments: Instructions I_1 and I_0
  - Returns: bool
  - Description: To query whether $I_0 \xrightarrow{\text{D}} I_1$
- `tokenRevdeps`
  - Arguments: Instructions I_0 and I_1
  - Returns: bool
  - Description: To query whether $I_0 \xrightarrow{\text{RD}} I_1$

B IndexAnalysisInfo API

- `getRAWlist`
  - Returns: A set of instruction pairs
  - Description: Returns a set of instruction pairs which exist in some SCoP and have RAW dependencies between them.
- `getWAWlist`
  - Returns: A set of instruction pairs
  - Description: Returns a set of instruction pairs which exist in some SCoP and have WAW dependencies between them.
- `getBase`
  - Argument: Instruction I
  - Returns: An array base
  - Description: Returns the base of the array which I accesses
- `isInScop`
  - Argument: Basic block BB
  - Returns: bool
  - Description: To query whether any SCoP contains BB
- `getScopID`
  - Argument: Basic block BB
  - Returns: int
  - Description: Returns an integer uniquely identifying the SCoP which contains BB
- `getOtherInsts`

- Returns: A list of instructions
- Description: Returns all memory instructions in SCoPs which do not require an LSQ connection

C MemElemInfo API

For MemElemInfo, an *LSQset* object represents a set of instructions which should be connected to the same LSQ.

- `getLSQList`
  - Returns: Set of *LSQsets*
  - Description: Returns a set of references to *LSQsets*, each of which contains those instructions whose hardware components should be connected to the same LSQ.
- `getInstLSQ`
  - Argument: Instruction *I*
  - Returns: An *LSQsets*
  - Description: Returns a reference to the *LSQset* containing *I*
- `BBhasLSQ`
  - Argument: Basic block BB
  - Returns: bool
  - Description: To query whether BB needs to be connected to any LSQs
- `getBBLSQs`
  - Argument: Basic block BB
  - Returns: Set of LSQs
  - Description: Get a set of LSQs to which the block connects
- `needsLSQ`
  - Argument: Instruction *I*
  - Returns: bool

  - Description: To query whether the component for I needs to be connected to a LSQ

Bibliography

[1] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[2] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[3] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. Elastic systems. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 149–158. IEEE, 2010.

[4] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd annual Design Automation Conference*, pages 657–662. ACM, 2006.

[5] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[6] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[7] Tobias Christian Grosser. *Enabling polyhedral optimizations in llvm.* PhD thesis, 2011.

[8] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[9] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. Elastic cgras. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 171–180. ACM, 2013.

[10] Lana Josipovic, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):125, 2017.

[11] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136. ACM, 2018.

[12] Chris Lattner. Introduction to the llvm compiler system. In *Proceedings of International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Erice, Sicily, Italy*, page 19, 2008.

[13] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[14] Chris Lattner and Vikram Adve. Llvm language reference manual, 2006.

[15] Reid Spencer and Gordon Henriksen. Llvm's analysis and transform passes, 2012.

[16] Michael B Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136. IEEE, 2012.

[17] Philip C Treleaven, David R Brownbridge, and Richard P Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys (CSUR)*, 14(1):93–143, 1982.

[18] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.