Abstractions for Performant Hardware Designs

Improving [productivity/usability/utility] of high level hardware languageso

Anonymous Author(s)

Abstract

Bridging the performance and productivity gap between low level RTL languages and high level DSLs has become an increasingly important research problem as the demand for heterogenous compute environments grows.
abstract Abstract abstract Abstract abstract ....

Keywords compilers, FPGAs, domain-specific languages

1 Introduction

[MATT: Is "performant" is a real english word? Linguists are debating on the internet]

In recent years, impressive advances in data science have lead to a sharp increase in the demand for high level application developers. As technology scaling and power constraints become increasingly significant road blocks, engineers are deploying applications in heterogenous compute environments which include CPUs, GPUs, field-programmable gate arrays (FPGAs), and coarse-grain reconfigurable architecuters (CGRAs). CPUs are the most mature platform from the perspective of programming languages and paradigms, but recently the demand for CNNs and other neural networks has caused GPUs to come to the forefront of performance-oriented programming.

There has also been a lot of focus on making domain-specific languages (DSLs), such as TensorFlow and Pytorch, that target high-level domain experts and allow them to compile efficient code for CPUs and GPUs. These languages have also made it possible for machine learning to gain momentum in many domains, as the programmer does not need to be an expert in cutting edge machine learning research and high performance software to solve the problems that are important in their respective field.

FPGAs have also begun to gain popularity in the high-performance computing community due to their performance-per-watt advantages over GPUs and CPUs. Projects like Micosoft's Brainwave and Amazon's F1 Instances are proof that major players in the tech industry realize the potential of FPGAs. However, their widespread adoption is currently being hindered by their extremely long compilation flows and notoriously difficult means of debugging. For many decades, Verilog and VHDL were the only language options, and many RTL developers still prefer to use these languages in order to get total control over their designs and write optimal code. It is only in recent years that there has been wealth of projects aiming to raise the level of abstraction. Tools like SystemVerilog, Chisel, and Bluespec were developed to make RTL design more user-friendly, by providing a familiar development environment, integrated testing tools, and syntax to automate mundane and repetitive tasks. Even higher level tools, such as Vivado HLS, OpenCL, and Spatial are now available to capture the domain experts who want to use familiar languages like C++ and Java to write code for FPGAs.

Unfortunately, this branch of programming language research has been slow to bridge the gap between productivity and performance. In short, there is a tension between low-level hardware-description language (HDL) users and high-level domain experts. Any high level DSL that claims to generate efficient hardware is viewed with skepticism from the HDL community. Such a compiler has so much complexity and opens up enough questions that HDL designers are hesitant to adopt it. On the other hand, the domain experts are often confused and frustrated by the abstractions provided by high level languages and do not understand how to write code effectively. Furthermore, writing in C++ or Java allows so much flexibility that the compiler is required to do a significantly heavy lifting to figure out how to map the code to hardware.

We believe that these concerns of both the HDL programmers and domain experts are interesting and important problems to solve. It is the responsibility of the compiler community to embellish high level hardware languages with the correct abstractions to make FPGAs a viable tool in the heterogeneous compute environment. A compiler must satisfy the needs of RTL designers by making it easy for them to understand how their code is being mapped to hardware and

PL'18, January 01–03, 2018, New York, NY, USA
2018.

providing the right abstractions to allow them to optimize their applications. It must also satisfy the needs of domain experts by providing a constrained programming model that assists in writing code that maps well to the underlying hardware but does not sacrifice expressibility.

In this paper, we begin by summarizing the current state of programming languages for FPGAs and arguing that it is a problem that is solvable but still requires more research. We then explain in detail why a language built on parallel patterns and hierarchical control flow provides the correct programming paradigm to meet the needs of all levels of users. We will specifically discuss the following abstractions:

- Digestible view of controller hierarchy embellished with performance metrics and resource utilization information
- Sensitivity analysis on automated design space exploration
- Unrolling and controller binding directives
- Streaming control flow
- Loop flattening/fusion/perfection
- Iteration difference and segmentation analysis
- Read-after-write cycle optimizations [MATT: (because WAW, WAR, and RAW were already done correctly in Spatial, and RAW wasn't)]

In short, we intend demonstrate that today's high level hardware languages inherently lack the ability to help users understand their designs and figure out how to improve them, and that forcing the hardware language to assume the software paradigm is at best awkward, and at worst prohibitively confusing for someone who knows how to write good hardware.

2 Background

While each language has its own advantages and disadvantages relative to others that exist at the same level of abstraction, we focus on what implications the programming paradigm in general. [MATT: bad sentence]

At the lowest level are RTL-based languages like Chisel, Bluespec, and SystemVerilog. These languages mark a significant improvement over Verilog and VHDL because they allow much of the repetitive work to be done with metaprogramming and thus make the user much more productive. They also provide intuitive verification and debugging interfaces and a wealth of optimizations that are tedious to do by hand. However, exploring a design space for an application at a relatively high level can become painful as things like banking and retiming is still a relatively manual task. Furthermore, the IR of these languages is so low level that it is difficult to provide design feedback to a user in a meaningful way and do optimizations at the algorithm level.

At the highest level, there are library-based approaches such as Halide and Tensorflow libraries that make it easy for a domain expert to write relatively simple code that translates to vast amounts of RTL code. However, these paradigms quickly break down as soon as a user wants to handle edge cases or add complexity to their design beyond what the underlying hardware libraries support.

Finally, tools such as Vivado HLS, SDAccel, Spatial, and OpenCL, which are embedded in the most popular general purpose programming languages, are a promising level of abstraction for capturing users at all levels of abstraction. However, these languages suffer from the issue that it is very easy for high-level users to quickly go off-the-rails and write code that is extremely hard to compile and optimize, leading to lots of confusion and frustration. Furthermore, because it is unclear how nested loops are scheduled and how snippets of code map to Verilog in the back-end, which turns away RTL designers who feel that they can write significantly better code by hand. Spatial aimed to solve this problem by building a new language embedded in Scala from that starts from parallel patterns and explicit memory declarations. While this appeases the RTL designers by giving visibility into how the state machines are hierarchically scheduled, it does not go far enough in meeting the demands of this community. It also allows enough flexibility and limited feedback that a domain expert who overcomes the learning curve still tends to gravitate towards software-like code that is inefficient for hardware. [KUNLE: Speaking from experience here from working with software people. Is this claim too controversial to make without a citation?]

[MATT: talk more about halide. Is it a good paradigm? What is it good at and what does it lack?] [MATT: What more is needed besides DSE? Why can't a compiler do structural rewrites as part of DSE?]

[MATT: Writing good apps is much more than transliteration from software to hardware]

[MATT: Instrumentation and understanding of hw performance is important, and lacking in existing ecosystems]

3 Language

[MATT: If you want to have a hardware language, you need to do it with parallel patterns. You also need to make it super clear to the user how the code maps to hierarchical control structures.]

A language must make it easy for users with diverse backgrounds to answer the most important questions about hardware design; where is the latency bottleneck and how are the resources being consumed. It must make it easy for software developers, who are accustomed to thinking about performance in terms of caching and instruction flow, to think about hardware concepts such as coarse-grain concurrency and operation pipelining. In order to be able to provide this

information in a logical way, a high level language must offer the following: explicit nesting of control structures, explicit memory hierarchy, and elaboration of a parameterized design space.

Control Hierarchy A high level hardware DSL must expose the control hierarchy to the user. This means that the loop and state machine constructs provided in the API must logically map to underlying nested control structures in hardware. Multi-level counter-chains and iterator parallelization must be easy to express directly in the loop syntax.

It must also distinguish between inner controllers, which contain only primitive operations (memory accesses, arithmetic, etc.) and outer controllers, which contain at least one control structure. Parallelization of outer controllers invokes unrolling that duplicates the child controller sub-trees, and parallelization of inner controllers vectorizes the iterators. Furthermore, it should be obvious how initation interval and body latency drive the performance of inner controllers, and how scheduling directives (i.e. parallel, pipelined, sequential, and fork) steer the children controllers of an outer controller.

This syntax allows the compiler to easily make powerful unrolling and scheduling transformations. However, when one actually tries to write a significantly complex application in the language, support for metaprogramming and composability can make it challenging for the developer to understand what is impacting performance the most. We introduce a way that performance information can be presented to the user in an intuitive way, as long as the host language provides an intuitive mapping of how structures in the source code are used to generate nested state machines in the backend. This way, it is possible to link source context with the compiler's view of the control structures. Runtime information can be computed in some cases and automatically profiled in others. This information can be embedded directly into the human-readable graph and overlayed on the originall application. Being able to inspect this graph at any depth makes it easy for the developer to learn how to tune their source code to remove bottlenecks.

Understanding performance can be broken down into two concepts that are only relevant if the host language exposes the controller hierarchy: initiation interval/body latency (inner controllers) and scheduling/congestion (outer controllers) [MATT: redo this paragraph... Have it make sense and be clear] All of these things are often related to design choices that are completely under the control of the developer, as long as the developer knows where to focus their attention. Without being exposed to this information, the user's only tool is to blindly explore the design space and see what parallelizations and tiling factors have what impacts on performance. This is a terrible way to do design.

Memory Hierarchy Being able to explicitly declare memories that exist on-chip and off-chip is another important feature a language must have. When users try to write significantly large applications, resource utilization becomes an important problem that the user should have some control over based on the source code. Specifically, `DRAM`s can be allocated by either the host or the accelerator, but always exist off-chip and can only be accessed at the burst granularity. `SRAM`s, `Reg`s, and `FIFO`s reside on-chip and can be accessed in a single-cycle at the element granularity.

A high level compiler must do heavy lifting to figure out how to bank and buffer the memories. Banking refers to physical partitioning of a logical memory in order to guarantee parallel accesses to a single-ported memory can all be served in the same cycle. Buffering refers to physical duplication of a memory in order to protect multiple accesses that happen in different stages of a coarse-grain pipeline. Seemingly innocuous decisions in the source code can lead to drastically different buffering and banking behavior, so the user should be able to understand how these decisions are affecting how resources are being consumed. An explicit memory hierarchy in addition to succinct syntax for communicating between off-chip and on-chip memories that can be used natively in the nested control hierarchy exposed in the source code, make it possible for the user to understand how certain restructuring of the source code expose trade-offs between resource utilization and latency/performance.

Design Space Exploration In addition to structural tweaks a user must do, there are many things that can be done automatically by a compiler. Specifically, a compiler should do a relatively good job at quickly exploring how different parameters affect the performance of the app. In order to let the compiler do this, there should be a concise way to elaborate the design space of a particular set of parameters. If this evaluation is fast enough, it allows the user to search the design space without too much head ache.

However, there are many cases in complex applications where the bottleneck is related to the way a controller subtree is composed and tweaking the value of parameters can only improve performance a little. DSE may return a bunch of points with similar performance, but the developer's intuition may indicate that the algorithm actually can run faster. A tool that automatically explores the constrained space provided by the source code could be even more useful if it could "describe" how different parameters affected the application. We propose that the developer should be exposed to the "sensitivity" of the application with respect to each design parameter, so that they can focus their attention on the regions in the application that are most dominant in affecting performance and/or resource utilization, and highlight why the performance they hope to achieve may not be possible anywhere in the design space of a fixed controller structuring.

For the rest of this discussion, we will assume the host DSL exhibits these design features. We will use these features as

the launching point into understanding how languages can be enhanced to make developers more productive and able to produce better code.

4 Feedback

[MATT: instrumentation counters and html display]

[MATT: quick dse before unrolling and after unrolling. Sensitivity analysis]

4.1 Understanding Performance

The control structure primitives in a high level language appear to be similar enough to software that they may give a false sense of security when the user attempts to nest and tile loops in an intelligent way. However, one of the most important concepts in learning how to write good hardware at a high level of abstraction is the distinction between inner controllers and outer controllers. While the same syntax is used for both, they have very different implications in terms of performance. Inner controllers are loops that contain only primitive operations, such as arithmetic and memory accesses. Outer controllers are those that only contain other controllers (either outer or inner). Being able to view the application from any subtree relative to any outer controller is the key to optimizing code, and will be discussed in great detail later in this section.

The performance of inner controllers are entirely captured by two numbers, body latency and initiation interval. Body latency is the amount of cycles it takes for the final operation in the body to be enabled relative to the first operation. Initiation interval is the amount of cycles the controller must wait before it can issue the next iteration of its counter, and is dictated by the longest accumulation loop in the dataflow graph. One bit of complexity that will be discussed later is that inner controllers can be "stream" based, in which there is an additional constraint on the availability of inbound or outbound streams, which throttles the execution of the controller in a data-dependent way.

The performance of outer controllers can be understood by their scheduling directive. These directives consist of pipelined, sequential, streaming, parallel, and fork. [MATT: show sample waveforms of theses]. While these are not unfamiliar concepts to most software and hardware designers, it is important to think of outer controllers in the context of "effective" initiation interval and body latency. For pipelined controllers, the "effective" initiation interval is equal to the total execution time per iteration of its longest stage (assuming it runs in steady state for a long time). Therefore, it is advantageous to use outer controllers as sparingly as possible, as fusing an outer and inner controller changes the initiation interval of the outer iterator from a value equal to the entire execution of the inner controller to the initiation interval of the inner controller itself. The rule of thumb that developers using high level hardware languages should have is that loops should only be nested if there is a DRAM transaction at that particular level of the hierarchy, fusing controllers explodes the initiation interval, or fusing fundamentally causes an incorrect result.

4.2 Instrumentation

Having a logical presentation of profiling information that fits naturally with the paradigm of the language is a necessary feature a language must support for a user to write applications that can be proven to be well written. One would only be writing code for FPGA if performance is the main concern, but there are no low level nor high level languages today that make performance profiling easy to extract and understand. High level languages claim that there are optimization transformers built into the compiler, but if they shy away from doing major restructuring of the compute graph (as they should), there are many missed opportunities for improving performance that must be done at the source code level.

It may seem obvious, but in order to translate high level descriptions of applications into high performance hardware designs, understanding the relationship between source code loop structures and generated hardware state machines is an extremely important feature that many languages do not intuitively support. Having a depth-agnostic view of the control tree makes it very easy for the user to identify and resolve bottlenecks effectively.

Below we outline the requirements of a useful instrumentation interface:

- Allow the user to explore the controller graph at any depth of the hierarchy without overwhelming the user with the rest of the dataflow graph
- Provide source context information justifying why a given controller fits in a given spot in the graph
- Initiation interval and latency values (for inner controllers) and scheduling directives (for outer controllers)
- Inbound and outbound stream interfaces (for stream controllers)
- Buffering and banking information for memories in the application
- Cycle counts for every controller in the hierarchy, broken down by total cycles active, total iterations completed, cycles active per execution of parent, executions completed per iteration of parent, total cycles stalled (outbound streams not ready), total cycles idle (inbound streams not valid)

Figure xx shows an example of a mapping between source code and controller hierarchy borrowed from an implementation of Convolution?

In the case of inner controllers, this view allows the user to figure out why certain controllers take so much time. If

initiation interval is greater than 1, this generally results in a huge loss in performance and the user should look at the primitives in that loop to figure out why a cycle exists and if anything can be done to rewrite it and shrink the number of cycles in the loop. In the case of low-rank accumulations, sometimes reordering loop iterators or moving non-static counter bounds higher up in the sub-tree can extend the iteration cycle.

In the case of stream controllers, a user can look at idle and stalled cycles to figure out if there is any particular controller holding up the execution of the pipeline. It is generally tricky to determine from the source code alone which controllers will run faster than others, but having this instrumentation feedback makes it very obvious to detangle this information and spend attention parallelizing or rewriting the most impactful controllers.

4.3 Parameter Sensitivity

In many applications, the compiler can provide feedback about how to optimize before it has to execute the hardware. A good hardware DSL should provide syntax for elaborating a design parameter space, and the compiler should be able to reason about this space to figure out which design points perform better than others.

The compiler can should be able to estimate runtimes of certain controller subtrees (such as DRAM transactions) by learning from enough training data and feeding congestion information into a model function. For dynamic controllers, there should also be syntax for the user to annotate features such as the maximum expected value of an input argument or data-dependent control parameter.

In the worst case, the compiler should help the user identify bottlenecks in the nested controller graph during compilation. A preliminary model of the controller graph should be generated before unrolling, as parallelization parameters are consumed by the unroller and baked directly into the IR. A final model of the unrolled graph should also be created but only for a user to rapidly estimate runtimes for various designs without needing to go through the painful process of running cycle-accurate simulations for each design.

Spatial boasts having these features, but we propose one more addition to assist in developer productivity. That is the user should be able to see the application's overall sensitivity to each parameter choice. I.e. given fixed values for all other parameters, what impact does each parameter have when it is changed to an adjacent value while the others remain constant. At the very least, this information helps the user prune their design space significantly if they can learn that the bottleneck always exists in a certain portion of the app and some of the design points tweak parameters in unrelated regions. In the best case, it helps the user figure out if a particular section of the application needs to be rewritten or refactored, because it turns out to be the bottleneck but is not sensitive to any parameter choice related to that controller.

5 Performance

In this section, we discuss the considerations that must be put into a high level compiler that will allow it to generate high performance code. There are three concepts that we will discuss: unroll bundling style, coarse-grained controller binding, and loop perfection. We limit this discussion to designs that use pipelined and sequential scheduling, as stream scheduling makes it impossible for the compiler to reason about the relative behavior of the iterators and operations inside.

5.1 Unroll Bundling

Loop unrolling is a well-defined and well-understood concept in compilers. Unrolling an outer controller involves making a copy of each stage of the child pipeline for each unrolled lane and dispatching the appropriate counter chain values to these lanes. However, there are two ways to schedule the unrolled lanes relative to each other, and being able to choose between one and the other can have implications on the performance of the overall design. The two options are to unroll as a metapipeline of parallels (MoP) and as parallel of metapipelines (PoM). Figure xx shows the difference. Under MoP, each stage in the original, unrolled controller becomes a parallel controller consisting of each lane's copy of that particular child. Under PoM, the entirety of the unrolled body is a parallel controller, and each lane is allowed to run through the entire pipeline independently of the other lanes.

The advantage of MoP is that it is less resource intensive. All of the unrolled lanes operate in lockstep, only one copy of the parent controller's counter chain is required. Furthermore, pipelines which are relatively well balanced across lanes do better under MoP and result in less costly banking schemes for memories accessed in the pipeline. However, there are two factors that normally indicate that PoM is a better unrolling strategy. First, if the runtimes of certain children are dynamic relative to the unrolled lane they are derived from, PoM allows them to run freely relative to each other. Otherwise, the effective runtime of the parent controller is always determined by the longest stage on any given iteration. Second, if the pipeline has dram transactions that are not well-balanced, PoM allows them to dephase and naturally minimize congestion on the DRAM bus. It is also important to consider that nesting controllers introduces a few cycles of communication overhead. For this reason, if a pipeline is well balanced and all of the stages are relatively short, MoP introduces more overhead as every stage gets its own nesting penalty, whereas in PoM, there is only a single nesting penalty regardless of how many stages the pipeline is.

Choosing which scheme is better at compile-time is a tricky problem. At the very least, the language should allow a user who knows what they are doing to choose between these two schemes either globally or per-controller in their

source code. However, there are heuristics that the compiler can use to decide between the two schemes. In our work, we have two models that are trained to determine the latency of accesses to DRAM. One model is lattice regression (probably too much to explain here?) and the other is simple regression that fits a bunch of training data. We can use this model to determine the latency of competing DRAM accesses under MOP and decide whether the DRAM transaction stages will be the bottleneck. Furthermore, we can analyze how much more costly banking will be under PoM. If resource utilization is the largest limitation in the application, MoP should be used. If the DRAM transaction stages are the longest children relative to the other stages, then PoM should be used.

[MATT: Expand on describing the heuristics]

[MATT: give concrete examples of PoM vs MoP for a given app]

5.2 Streaming Controller

Previously, we describe how unrolling decisions can be used to improve the performance of an application with outer-loop parallelization. However, there is another technique that can be used specifically in cases where either the outer controller's children runtimes are highly dynamic to the point where there is no way to statically analyze them, or when the parent controller runs for few iterations relative to the depth of the pipeline. In these cases, the user should be allowed to define stream-based control, where the execution of different stages in a controller are determined by availability of input data, rather than managed by centralized controllers. However, stream controllers are notoriously difficult to work with relative to conventional sequential and pipelined controllers, and can be tricky to use as they often lead to deadlocks in hardware that are difficult to capture in software simulations.

However, here we outline when stream controllers are appropriate and why a user should consider using them in certain applications. In highly dynamic applications, such as those that consist of a centralized controller who is arbitrating incoming packets and offloading to worker units (i.e. Grazelle), it is very inefficient to use software-like control structures. The other case is when an outer controller runs for very few iterations relative to the number of stages in the pipeline, and each of the children stages runs for relatively many iterations. In this case, the outer controller does not run in steady-state for many iterations, so gating the execution of later stages in a pipelined fashion is inefficient if the data from earlier stages becomes available to later stages. Not only does this allow eager execution of the stages and reduces the effective latency and initiation interval of the outer controller (since child stages can overlap immediately), but it also provides the opportunity for the user to significantly reduce the resources required to implement the algorithm. Rather than allocating entire memory structures to hold intermediate data, and force the compiler to buffer the memory to protect each stage's view of it, the user can use FIFOs that can be significantly smaller as data can be loaded and drained simultaneously.

[MATT: show either waveform or html to prove the point here. Maybe talk more about deadlocking risk. Discuss how a compiler could transform between stream/pipeline OR discuss why it is philosophically wrong for a compiler to do so]

5.3 Controller Binding

For users who come from a software background, it is often not intuitive to think about how operations and loop sub-graphs are scheduled when writing the source code. Specifically, the user may express two loops one after another, but they only read and write to non-overlapping sets of memories. In this case, it should be the responsibility of the compiler to bind these two stages and execute them in parallel if such a transformation is more optimal. Deciding whether this transformation is optimal, however, can be tricky.

In general, the heuristic that should be used is that if the two stages involve communication with DRAM, then binding them in parallel may result in sub-optimal congestion. Furthermore, if a controller does not run in steady state for many iterations, or if every stage runs for relatively few cycles, then adding the additional level of nesting could cause communication overhead that slows down the overall pipeline.

On the other hand, there are many cases where this optimization is useful. One example is if there are highly imbalanced controllers than can be bound, and a memory is buffered across them. This means that binding would improve the performance of the controller while the pipeline fills and drains, and the memory would require fewer buffers.

5.4 Loop Perfection

While loop perfection is not a new idea, it is especially important in the context of hierarchical control structures. In general, a deeper nesting hierarchy will result in worse performance, especially when controllers run for relatively few cycles This concept is described in section xx (reference feedback), but here we discuss techniques for alerting the user when fusion should be attempted and heuristics when the compiler should be smart enough to transform automatically.

Any time two controllers are perfectly nested, meaning that there are no transient primitives in the outer controller used to set up the counter chain or enables of the inner controller, and there is only one child controller, the compiler should automatically fuse controllers. While it is possible that the compiler may end up deciding it needs a higher initiation interval of the fused controller (i.e. the inner controller's iterators cannot be analyzed statically and there is a new low-rank accumulation into a memory [MATT: show example of what I mean here?]) it will be equivalent in latency and use less hardware to the unfused version. In most

cases, the runtime will be faster and fewer resources will be consumed.

Cases that are relatively tricky (not impossible) for a compiler and should be left up to the user in high level hardware DSLs (and probably be done by higher level compilers that target the hardware DSL), is fusion that predicates operations. For example, an outer controller whose first stage computes an intermediate result in a low rank accumulation over many cycles, and whose second stage is a simple mem copy-out operation could be fused if the primitives that originally belonged to the first stage are predicated by the condition that the outermost iterator is on its final value. This way, the effective initiation interval of that outer controller can be small, buffering can be eliminated, and behavior can be left untouched.

6 Correctness

In a high level hardware language, correctness and performance are often adversaries of one another. The user has significant power to parallelize and pipeline their application, but in many cases, some performance is lost in order to ensure the result does not change.

6.1 Iteration Difference Analysis

Iteration difference analysis is required to maintain correctness in applications where the user writes arbitrary accumulation cycles within an inner controller. We argue that it is important that the user is aware of the results of iteration difference analysis so that they can focus their attention on parts of the application where initiation interval is bloated to guarantee correctness but the developer knows information about the data and can rewrite the code or relax the constraints. Iteration difference analysis can especially result in sub-optimal code when the user tries to perform loop perfection that requires low-rank updates to a memory relative to a group of iterators. Very simple rewrites can speed up the application in cases where multi-iterator loops can have their iterators reordered to extend the interval between interfering accesses.

[MATT: scrub the pseudo code here]

In the case where accesses to a memory have at least one affine component that depends on loop iterators, the above analysis will compute the effective distance between the read and the write in an accumulation loop. This information is important to know in cases where there is a write to an address that the reader will need on a future iteration. The initiation interval of the controller will be defined as the total latency of the accumulation loop divided by the iteration difference. It is important to note that in cases where components of the access pattern are affine but the iterators' start, step, or stop values cannot be statically determined, the analysis must assume that the iterator can wrap on every iteration and therefore a conservative iteration difference

```
1   for all inner controllers c that accumulate on
        memories m:
2     A ← all (m,r,w) accumulation cycles
3     for all A, collect:
4       I = iterators used in access pattern
5       Ī = all iterators in scope of outermost in
            I
6       R = Mapping from each element in Ī to
            number of ticks of innermost iterator
            before incrementing
7       Compute iteration distance between reader
            and writer
8       Convert iteration distance to tick count (
            including out-of-pattern iterators)
9       (m,r,w) \leftarrow minimum tick distance
10      if par for innerloop iterators > 1:
11        for lane in iterator vectors:
12          if lane depends on any prior plane:
13            lane.segment = plane.segment + 1
```

Figure 1. Iteration difference and segmentation analysis *m*.

value must be used. The most common example of this is when the stop value of a counter is derived from an argument in. However, to gain performance, the user should be alerted when this is the case and be given advice on how to fix this to get better performance, whether by reordering the iterators to put the dynamic one towards the outermost iterators or assure the compiler that this particular iterator will run for "many" cycles and its tick cycle should be treated as an arbitrarily large value.

The key difference that makes this analysis different from the equivalent pass in a software compiler is that parallelization may lead to segmentation. If the innerloop's iterators have parallelization, this value should be taken into account and a modified step size should be used for the iterator. However, depending on the access pattern, it is possible that there is a dependency between lanes of the vectorized loop. This happens often in dynamic programming, and a user unfamiliar with thinking about problems in hardware may not immediately understand why parallization of the bottleneck loop may result in equivalent or even worse performance. The segmentation information should be computed before unrolling and retiming analysis, and only used in the very last retiming transformation pass. An access with segmentation greater than 0 means that the node should not begin its execution until at least one cycle after the latest node with a segmentation value that is 1 smaller than itself. Any node in the dataflow graph which depends on this access should also be delayed by the same number of cycles required to place this initial access to its appropriate delay.

Segmentation and iteration difference analysis (i.e. data safety via initiation interval) are two key concepts that make high performance hardware difficult to write even in a high level DSL. A lot of the loss in performance when users try to parallelize and pipeline their algorithm is due to these two constraints enforced by the compiler to ensure correctness

for all parallelization and pipeline directives. In very limited cases, however, the user should be allowed to override this behavior if they come from an RTL background and know exactly what they want their hardware to look like and know that the design is "safe."

6.2 Program Order

In the majority of cases, an un-timed dataflow graph is sufficient to perform retiming analysis, and nodes can be retimed solely based on the timing of their input nodes. However, there are a few cases where it is important, especially in a high level language that aims to target software developers, to respect program order when performing the retiming analysis. Two such cases are when loop-breaks are used and when registers contain read-after-write cycles within an inner controller.

Loop-breaks are an important feature for any dynamic or sparse algorithm that are, at best, awkward to express in the language API, and at worst, result in incorrect or unpredictable generated hardware. However, because they are needed in a lot of algorithms that people want to implement of FPGAs, a DSL should support them. In the past, languages have supported arbitrary FSMs to implement while loops, but these controllers necessarily have an initiation interval equal to the body latency, even if the "state" variable is being used as an iterator. In the software programming paradigm, most languages support a way to "break" out of a loop when a condition is met. In a hardware DSL, having this same feature would allow for better performing code.

One way to support them is to allow the user to declare a "while" register above the loop that they may want to break out of, and write to the register based on some condition inside of the loop body. Many times, users precede mutations to memories by a write to this register, it will guard the mutations. However, this fundamentally works against retiming, which depends only on dataflow order. Therefore, a compiler should explicitly place any node that comes after the write to the "while" register in program order strictly after the latency of the write. While this does not affect initiation interval, it does affect overall loop latency but this is an acceptable tradeoff as initiation interval is generally more impactful on performance.

In cases where a user unfamiliar with software chooses to write to a register and then read from that register later in the same inner controller, the same analysis should be done. One practical case where this may happen is if the user wants to do an accumulation in an inner loop, and use the value of that accumulation later in the loop. Program order of nodes should be injected as gating directives to the retiming analyzer.

The one tricky exception is when a register is written inside of one branch of an if-else statement, and used by a later branch in the same if-else block. The read will appear to succeed the write in terms of program order, but the retiming should not actually occur. It is important to have an analysis pass prior to retiming which annotates register reads and writes with an effective "branch" ancestor in which they occur, so that retiming can be done correctly. /mattshould I write pseudocode here? It's a little less straightforward when people have deeply nested if-else blocks