Slide 1:

Hi everyone, I am Disha. Welcome to my dissertation defense and thanks a lot to all of you for your consideration in allowing me to defend remotely.

Slide 2:

This is the agenda for today. First, we will discuss what is behavioral synthesis and what is a loop in the context of behavioral synthesis. Then, we give a quick understanding of the overall project for certifying behaviorally synthesized pipelines.

Then we focus on how and why to certify the loop pipelining transformation and the need for a certified loop piplining algorithm which is the crux of my dissertation.

We disucss our approach of creating a framework of certified pipelining primitives which are essential to build pipelining algorithms. We show how to create a certified loop pipelining algorithm from ground up using our franmework. In parallel, we also give a quick idea of how we prove the framework and the algorithm in a mechanical theorem prover called ACL2. Then, we share the experimential results to shoe that our approach works for industrial strength designs. WE discuss the related work to emphasize on the novelty of our approach. We conclude with a brief summary and some insight into possible future work.

Slide 3:

So, first some background.

Slide 4:

Behavioral synthesis is an automated design process that takes high level specification of a hardware design in say C language and compiles it to low level hardware description language using sophisticated transformations. The transformations are divided into three broad categories: compiler transformations, scheduling transformations and resource allocation and control synthesis. We are not gonna talk about the other transformations in detail today. My area of research and the topic of conversation is specific to loop pipelining.

Slide 5:

So, what is a loop in behavioral synthesis. Let’s take a look at this example.

We are given a C code in the beginning of behavioral synthesis. We go through some transformations and come to an intermediate design before pipelining.

There are some things we need to concern ourselves with.

First, our design now has a schedule, concept of clock cycle. So, our loop here is divided into 3 scheduling steps X, Y and Z. We have entry block which signifies things before the for loop and exit block for things after the for loop. Each rectangular box here represents a scheduling steps i.e, a combination of micro steps which can be done in a single clock cycle. So, one iteration of a loop here takes 3 clock cycles.

We call this representation a Clocked Comtrol Data Flow Graph (CCDFG)

SSA, because of that we need phi-construct

Most of the operations here are straightforward except the phi-construct.

Slide 6:

The phi-construct says that the value of a variable is dependent on the previous basic block, so for the first iteration where the previous basic block is Entry, both a and I get the value 0, for all the subsequent iterations the previous basic block is Z from within the loop so a and I get the value a’ and I’ respectively.

Slide 8

Now, given a sequential loop with 3 scheduling steps, if we consider a pipeline interval of 1, then this is what the pipelined loop looks like. This stage represents the pipeline full stage, if we unroll it any more, then adding more iterations merely increases the number of iterations in pipeline full stage.

Slide 9

As you can see, pipelining improves throughput and reduces latency of the synthesized hardware. 3 iterations of the sequential loop takes 9 ierations as against 5 iterations in pipelined structure so this trasfrmaton is very important.

Slide 10

This is the overall certification framework proposed for certifying behaviorally synthesized pipelines. It can be divided into 3 separate and different PhD thesis ☺ The compiler and simple scheduling transformations which do nt change the structure of a CCDFG much are tested usng front end checker which uses SEC. After the loop pipelining transformation has been applied, the pipelined CCDFG has an identical structure with the pipelined RTL, So we can again use SEC and verify. This is known as Back End Checker. Loop pipelining transformation however can not be certified using convential methods because of huge semantic gap between sequential and piplelined CCDFGs.

Slide 12

**Implementation is not disclosed by vendors, so we cannot certify the transformation by theorem proving**

Slide 13

**Sequential equivalence checking does not work due to huge abstraction gap between the two designs**

Slide 14

Hao et al. proposed a reference pipeline approach to verify loop pipelining transformation. They proposed that if we use the pipeline parameters from behavioral synthesis tools, and create our own certified loop pipelining algorithm to create a pipeline reference model, then we can use SEC between the pipeline reference model and pipelined RTL. If this passes, then we would have proved by an alternare route that sequential CCDFG Is same as pipelined CCDFG.

They showed the viability of their approach across industrial-strength designs, however their algorithm was not certified rendering the certification flow unsound.

Slide 15:

My specific goal is to create a pipeline algorithm which can be proved by theorem proving. So, first I want an algorithm that can generate pipelined design from sequential design. Also, I want to prove once and for all that the algorithm is formally correct.

Slide 17

We have built a framework of certified pipelining primitives which we have shown are essential to build a simple pipelining algorithm. You may have more optimizations, but these are the fundamental features which form the backbone of a pipeline generation algorithm. Using our framework, we build a certified loop pipelining algorithm. We also have proven a key invariant which connects the loop in sequential For each of these tasks, I propose a simple algorithm to do that. So, all we need to do is prove these tasks to be correct by theorem proving, then we can create a simple algorithm through their combination which would be correct too.

Slide18

Before we understand the primitive and the entire algorithm, let us ynderstand the challenge in loop pipelining. Here we have overlapped only 2 iterations of an uunrolled loop. The first discrepancy we see is when the exit condition becomes true in the second condition. In the sequential CCDFG, it means we would have executed the first iteration X, Y, Z and a small portion of X which is before the conditional branch. IIf we follow the pipelined CCDFG structure, we exit from X but we have not exexuted Z of the previous iteration yet. The next problem is with the phi-construct. We expect the previous basic block to be wither Entry or Z, other wsie the meaning is undefinecd. Here the previous basic block is Y. Now, even if we resolve the phi-construct correctly, there are data hazards. Note that I’ is read in X while it is written in Z. If we go sequentially then we write I’ in Z of first iteration and read in X of second iteration as so on which is correct. In the overlapped structure, however, we are trying to read I’ in X before it has been written in Z. Another problem is that a’ is written in X of an iteration and then read in Z of the same iteration before being written again in next iteration. If we overlap the iterations such that X of second iteration occurs before z OF FIRst iteration, then the value of a’ is overwritten in X before Z has a chance to read it which is a pronlem. Besides these isseus for unrolled loops, we also eend ro find a way to prove correspondence netween backedes of sequential and pipelined CCDFG’s.

Slide 20

To tackle these issues, we have built a framework of 5 certified primitives.

Slide 21

We have divided our approach into 8 broad steps: The first two steps are to identify succint primitives essential for all pipelining algorithms and certify their correctness standalone. Besides primitives, our algorithm aslso has other components such as for identifying loop dependencies, variables and microsteps causing data hazards. We create the loop pipelining algorithm using our primitivrd and certify our algorithm. Certification of algorithm requies us to prove 3 thins

Make sure that the primtives are certified

Application of primitive

Invariant

Test on desing.

We now look the each of the challenges and the solution.

Slide 22

As mentioned earlier, whenever we exit from an iteration in a pipleiee, we need to make sure that all the scheduling steps of the previous iterations are complete before we Exit.

Slide 23

Reasoning about branches is a complex task. We have come up with an invariant that lets us remove and add branches if certain well formed conditions are met. Note that there are three types of CCDFG’s here. S, Sloop and SpreExit. We know that our CCDFG has a well-defined structure. We note that if we execute S such that S exits in the (k + 1) st iteration, then it is the same as executing Sloop which is S without the conditional branch for k iterations and executing SpreExit which is a collection of msteps in S before the exit conditional branch.

Slide 24

The first step of the algorithm is to remove branches. We need to identify the branches and apply branch primitive.

Slide 25

The proof of the branch primitive is fairly intuitive but we prove it by induction. In the algorithm we prove …

Slide 26

The second problem was with phi-construct and its resolution.

Slide 28

The phi-removal primitive says that if we can statically determine the previous basic block, then we can replace the phi construct with the corresponding assignment statements. For example here,

Slide 29

Now, we know that phi resolution is different when we enter the loop for first time vs when we go for every other new iteration. So, we unroll the loop once to ensure that previous basic block is sdetermuned statically. This can be proved by simple inducton on the no of iterations. We call this first step Spre.

Slide 31

Then we apply the phi-eliminatin primitive. As you can see,

Slide 32

The proof of this step is also based on induction along the lengths of CCDFG. Note that one phi-constructs leads to many assignment statements so there is a mismatch in lengths of CCDFG. Also, we need to make sure previous basic block is determined correctly and well-formed-conditions are maintained after every application of phi-removal-primitive before applyting a new primitive,

Slide 34

The next problem is that we are trying to read I’ even though it has not been written yet.

To take care of this, we introduce interchange primitive. Interchange primitive says that if there are two microsteps and they do not have any read-write hazards meaning that what m writes n does not read and vice versa and they both dnt write the same variable, then executing m followd by n is same as n followed by m. This is proved by analyzing the semantics of all possible statements.

Slide 35

To get rid of WAR hazards, in the pipelining algorithm, we have a step called data propagation.

We first identify all the variables that cause this hazard, then for each variable, we move the step which is causing the conflict to beginning of loop. This is done by applying interchange primitive multiple times till we reach the beginning. Note that by definition itself we know that there would be no read write hazards in these statements. For example, here i=I’ is moved to beginning.

Slide 38

Now we move the conflct step to previous iteration by moving it to Spre and move it to end of Sloop and remove it from SpreExit. This step can be visualized here. For k=0 we can see this is true.

For rest. Explain

Slide 41

Next problem is that a’ is overwritten before being read.

We introduce a primitive called shadow-register primitive which basically stores these values in temporary registers so that the value is not overwritten before it is being read. Over here,

Slide 42

In the algorithm, we call this step, add shadow registers where we first identify the steps which can cause RAW hazards and then apply shadow regsistre primitive for each conflicted step. Here,

The proof :

Slide 45

Now, since we have removed the data hazards, we use the superstep construction primitive/algorithm to overlap the iterations. But simply overlapping an unrolled loop makes it very difficult to put the backedge back in place. We need a way to define the correspondence netween the two edges.

Slide 46

We have come up with a unique invariant which is unlike any other invariant used in the proofs of microprocessor pipleines so far. Our invariant says that executing pipeline prologue + k iterations of pipeline full stage is same as **Executing k iterations of sequential loop + m blocks of first iteration + (m-I) blocks of second iteration + …**

Slide 47

For base case where k = 1,

Pipeline prologue + 1 iteration of pipeline full stage is same as executing 1 iteration of sequential loop followrd by 2 blocks of first iteration + 1 block of second iteration.

As you can see that z and x now have no read write hazards since we have already made sure of that.

Slide 48

Now, assuming that invariant is true fpr k steps, if we add one more pipeline full stage to both LHS and RHS and then we interchange x and z since no data hazars, we get what we expected.

Slide 49

We can see

Slide 50

The superstep construction primitive says that we can overlap iterations if no data hazars and we can prove invariant.

Slide 51

Superstep construction replaces spre and sloop

Slide 52

Now we interchange Ppost and SpreExit which is ths same as a subset of X and Z, so we know that there would be no data hazards

Slide 53

Now we note that this matches exactly with this and so we can apply the reverse of branch primitive to add branches back.

Slide 56

We have tested this on industrial-strength designs across different domains

Related work

Summart

Future work