Derivation of Efficient FSM from Loop Nests

Tomofumi Yuki*
*INRIA
Campus de Beaulieu
35000 Rennes
tomofumi.yuki@inria.fr

Antoine Morvan†, Steven Derrien†
†Université de Rennes 1
Campus de Beaulieu
35000 Rennes
{antoine.morvan,sderrien}@irisa.fr

***Abstract*—Pipelined execution is one of the most important optimizations in hardware design to improve hardware utilization rate, and hence the throughput. Loop pipelining is a transformation available in High Level Synthesis tools to execute multiple iterations of a loop in a pipeline.**

Nested loop pipelining is a related technique that improves hardware utilization rate when the iteration count of the innermost loop is small. However, it is also known to increase the complexity of the control, and hence degrading frequency.

In this paper, we present an automatic transformation targeting HLS that improves the effectiveness of nested loop pipelining, by efficient implementations of the control-path.

Specifically, we present (i) an analytical model that captures the trade-off between gain in cycles and loss in frequency, (ii), automatic derivation of efficient Finite State Machine from loop nests, and (iii) an efficient implementation of the derived FSM that improves the performance of synthesized hardware.

I. Introduction

Embedded systems have been, and are increasingly so, an important component of many electrical systems. As the need for energy efficiency grows, specialized custom hardware is receiving more and more attention. However, custom hardware design is an effort intensive procedure, taking many man-months if not years.

High Level Synthesis (HLS) is a response to the desire for faster "time-to-market". HLS tools derive hardware descriptions from behavioral specifications given in higher level languages (typically C/C++) than previous RTL methodologies. After decades of research, HLS tools have reached significant degree of maturity, and are now being adopted by the industry as part of their production process.

However, naïve application of HLS tools can lead to significantly inferior hardware design compared to manually designed counterparts [1]. The output produced by HLS tools is very sensitive to the control and data structures exposed at the source level and minor modifications can greatly impact the synthesized hardware.

The usual design flow using HLS tools includes a design space exploration step at the source level. This exploration is done by transforming the specification at the source level and by interacting with the HLS tool to apply automatic transformations and provide decisions about the target architecture.

One key transformation provided by HLS tools is the application of *loop pipelining*. This transformation improves the hardware utilization rate and reduces the execution time by exploiting the parallelism available in the loop nest. Moreover some tools propose the application of *nested loop pipelining* (or *outer loop pipelining*), namely the pipelining of several levels of loop. This transformation helps reducing the overhead due to *initiation* and *flush* phases that are predominant when the innermost loop has a small iteration count [2], [3].

The nested loop pipeline is controlled by a Finite State Machine (FSM) that is usually built by coalescing (flattening or collapsing) the loops to pipeline [4]. This coalescing can lead to an increase in the pipeline control complexity that may reduce the maximum achievable frequency [5]. The degradation in frequency due to inefficient controls due to coalescing may cancel the gains coming from increased hardware utilization. Thus, there is a need for deriving efficient controls to iterate over coalesced loops.

In this paper, we present the following contributions:

1) We develop an analytical model that capture the effectiveness of nested loop pipelining, with respect to the improved hardware utilization, and degradation in frequency when compared to single loop pipelining. This model can be used to guide the design space exploration by designers and/or HLS tools.
2) We propose a method to derive efficient FSMs from imperfectly nested loops with affine bounds.
3) We present a code generation strategy that takes abstract representations of the derived FSMs and generates C codes suitable for HLS.
4) We further improve the performance of the derived FSMs by looking several steps ahead in order to enable the pipelining of the FSMs.
5) We have implemented our approach as an automatic source-to-source transformation for HLS tools. We show that the state look ahead significantly improves the maximum achievable frequency when compared to conventional nested loop pipelining.

This paper is organized as follows. We develop a model for analyzing the effectiveness of nested loop pipelining in Section II. The derivation of FSM from loop nests is presented in Section III, followed by the presentation of our code generation strategy in Section IV. In Section V, we introduce state look ahead as a means to enable pipelining of the control path. We evaluate our approach through experimental validation in Section VI. We discuss related work in Section VII and conclude in Section VIII.

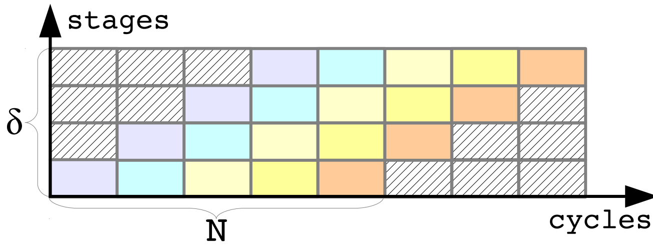


Fig. 1: Illustration of pipeline overhead for a loop with number of iterations $N = 5$, and pipeline latency (number of stages) $\delta = 4$. The shaded triangular regions are stages of the pipeline not utilized.

II. Pipeline Overhead

Loop pipelining is a critical optimization in hardware synthesis that significantly boosts its computational throughput. In this section, we analyze the overhead of loop pipelining and the effectiveness of nested loop pipelining.

A. Nested Loop Pipelining

Loop pipelining comes with the usual overhead for any pipelining; pipeline fill and flush. Given a δ-stage pipeline, it takes $\delta - 1$ cycles before all stages in the pipeline are activated, assuming initiation interval of 1. Similarly, towards the end of the pipelined loop, there are no more iterations to issue, and thus some of the pipeline stages become inactive before the others. These pipeline overheads are illustrated in Figure 1.

With small values of N, the pipeline overheads dominate, resulting in low hardware utilization rate. Pipelined execution of a single loop with N iteration takes $N + \delta - 1$ cycles. If the pipelined loop is surrounded by another set of loops with M iterations, it takes $M(N + \delta - 1)$ cycles, excluding loop control overhead. In other words, additional $M(\delta - 1)$ cycles are spent when only the innermost loop is pipelined.

This has led to what are called *nested loop pipelining*, where loop nests are pipelined instead of the innermost loop itself. Then the fill and flush overhead are only present at the start/end of the full loop nest, significantly reducing the overhead. This can be seen as converting loop nests into `while` loops, and letting the HLS tool pipeline the `while` loops when implemented as a source-to-source transformation.

Using the above example loop nest with M and N iterations, the total number of cycles are reduced to $MN + \delta - 1$, which is $(M - 1)(\delta - 1)$ cycles less than only pipelining the innermost loop. This eliminates most of the wasted cycles, except for the unavoidable $\delta - 1$ cycles at the end of the pipelined execution.

B. Effectiveness of Nested Loop Pipelining

However, it is not always beneficial to apply nested loop pipelining. When compared to the design with only the innermost loop is pipelined, the control becomes much more complex, since the pipeline spans across multiple loops. This drawback is reflected in elongation of the critical path, i.e., decrease in achievable frequency of the clocks. Thus, it is important to distinguish when to apply nested loop pipelining.

If nested loop pipelining is beneficial can be analytically reasoned. The important parameters are:

- N: The number of iterations in the innermost loop that is pipelined in both cases.
- M: The number of iterations in the outer loops that are pipelined only when nested loop pipelining is used.
- δ: Number of pipeline stages.
- f^*: Frequency with nested loop pipelining, normalized to that of single loop pipelining.

Let C^* be the number of cycles with nested loop pipelining, normalized to the number of cycles with single loop pipelining. Using the analysis developed in Section II-A, we obtain the following:

$$\begin{aligned} C^* &= \frac{MN + \delta - 1}{M(N + \delta - 1)} \\ &= 1 - \frac{(M-1)(\delta - 1)}{M(N + \delta - 1)} \\ &= 1 - \frac{\frac{M-1}{M}(\delta - 1)}{N + \delta - 1} \end{aligned}$$

Let the ratio of pipeline stages with respect to the inner most loop trip count be denoted as $\alpha = \frac{\delta - 1}{N}$. Substituting $\delta - 1$ with αN yields:

$$\begin{aligned} C^* &= 1 - \frac{\frac{M-1}{M}\alpha N}{\alpha N + N} \\ &= 1 - \frac{\frac{M-1}{M}\alpha}{\alpha + 1} \end{aligned}$$

With large values of M, C^* asymptotically approaches the following:

$$\begin{aligned} C^* &\approx 1 - \frac{\alpha}{\alpha + 1} \\ &= \frac{1}{\alpha + 1} \end{aligned}$$

Thus, the execution time with nested loop pipelining, normalized to that of single loop pipelining, can be approximated by the following:

$$T^* = \frac{C^*}{f^*} \approx \frac{1}{f^*(\alpha + 1)} \tag{1}$$

Figure 2 illustrates the trade-off between reduction in cycles due to nested loop pipelining, and degradation in frequency. Our aim is to improve the design of the control-path such that the effectiveness of nested loop pipelining is strengthened.

III. FSM Derivation

In a custom hardware design flow, the mapping of loops to hardware is an important step to achieve efficient accelerators. Parallelism and locality are two factors to optimize in order to achieve good performance. However the control logic of the loops has to be efficient enough to avoid hindering these two factors. FSM is one way to efficiently map loops to harware as FSMs are straightforward to implement in hardware.

The basic technique to derive an FSM from a hierarchical loop nest is to syntactically coalesce (flatten or collapse) the

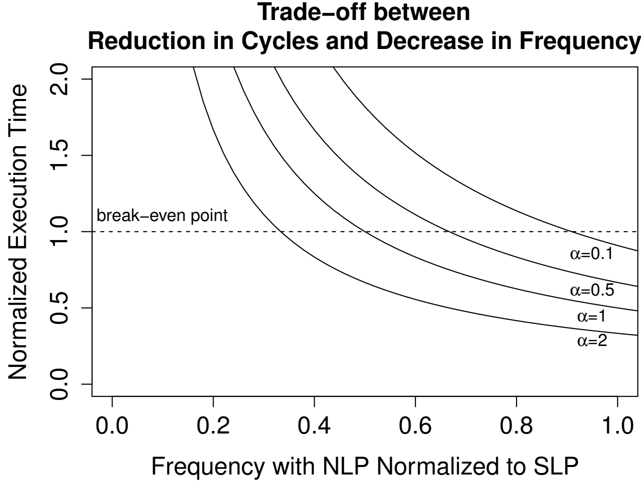


Fig. 2: Illustration of the effectiveness of nested loop pipelining according to Equation 1. The dashed line shows the break-even point where nested loop pipelining does not do any better due to excessive degradation in frequency. The plot show multiple curves corresponding to different values of α. With high values of α, i.e., with small inner most loop trip counts, nested loop pipelining is likely to have greater benefits.

```
for (i = 0; i < 100; i++)
  for (j = 0; j < 2; j++)
    S(i,j);
```

(a) Original sample loop.

```
i = 0, j = 0;
while (i < 100)
  if (j < 2) {
    S(i,j);
    j++;
  } else
    j = 0, i++;
```

(b) AST-coalescing.

```
i = 0, j = 0;
while (i < 100)
  S(i,j);
  if (j < 1)
    j++;
  else
    j = 0, i++;
```

(c) 0-overhead coalescing.

Fig. 3: Difference between (b) AST rewriting and (c) zero-overhead loop coalescing. Using AST rewriting, whenever j reaches 2, that is 1/3 of the iterations, time is spent on control only. Using zero-overhead loop coalescing, j never reaches the value 2.

loop nest using AST rewriting [4]. Figure 3b illustrates such loop coalescing on a simple example. This example shows that this syntactic coalescing can lead to inefficient implementation as 30% of the iterations (when $j = 2$) are dedicated to control.

More advanced techniques can derive an FSM where the control visits only values of the iterators where memory operations are computed. This is known as deriving a *zero-overhead loop*. Figure 3c illustrates an FSM where j never reaches 2. Previous approaches for deriving such FSM are very limited as they can only handle perfect rectangular loop nests with constant bounds [6], [7].

In this work we borrow a technique from automatic parallelization targeting general purpose machines. Loop transformations are often applied in automatic parallelization to expose parallelism, improve locality, and so on. The state-of-the-art automatic parallelizers abstract loop nests in a specialized intermediate representation to ease analyses and transformations.

One of such intermediate representation, called the polyhedral model, represents statements executed in loop nests as mathematical objects. These mathematical objects are converted back to loop programs after performing analyses and transformations. In automatic parallelization, the mathematical objects are usually converted back to loop nests [8], [9], [10].

However, a method for enumerating the same sequence of operations using automata has been proposed by Boulet and Feautrier [11]. Although their initial motivation was to generate low level code for programmable processors, this technique is well suited for hardware synthesis. The automaton generated using their approach actually happens to be a zero-overhead loop that can be implemented in the form of an FSM.

Their approach consists in generating a `next` function that, given the current value of the iterators, computes the value of the immediate successor following the loop execution order. This function is then used as the transition for the FSM.

This method is applicable to a subset of loop nests where the loop bounds are affine expressions of the surrounding loop iterators and program parameters. In addition, `if` statements can also be handled provided that the conditions are also affine.

Within the polyhedral model, one does not represent loops, but statements. For each statement within a loop nest is associated its *iteration domain*. The iteration domain of a statement S is denoted as D_S, and is a set of integer points in a multidimensional space. This set of points represents the values that the iterators of the loops surrounding S, the *iteration vector*, can take, and is defined by a set of m affine constraints representing a polyhedron:

$$D_S(\vec{n}) = \{\vec{x} \mid A_S.\vec{x} + B_S.\vec{n} + \vec{c_S} \geq 0\}$$

where $A_S \in \mathbb{Z}^m \times \mathbb{Z}^q$, $B_S \in \mathbb{Z}^m \times \mathbb{Z}^p$ and $\vec{c_S} \in \mathbb{Z}^m$, q represents the number of dimensions of the iteration domain and p the number of parameters. The particular instance of statement S for iteration vector $(\vec{n}, \vec{x})$ is the *operation* $S(\vec{n}, \vec{x})$.

Figure 4 illustrates a loop nest and its polyhedral representation. The same idea carries over to programs with multiple statements and with imperfectly nested loops as shown in Figure 5. The mathematical formalism is applicable to unions of polyhedra, making handling of imperfectly nested loops no different from perfect loop nests.

From the polyhedral representation, the goal is to construct a function `next` that, given an iteration vector $\vec{x}$, gives its immediate successor in the iteration domain following the lexicographic order [11]. The technique consists in building a relation that associates an iteration vector to all its successor, and selecting the smallest one (the immediate successor) by computing the *lexicographic minimum* of this relation:

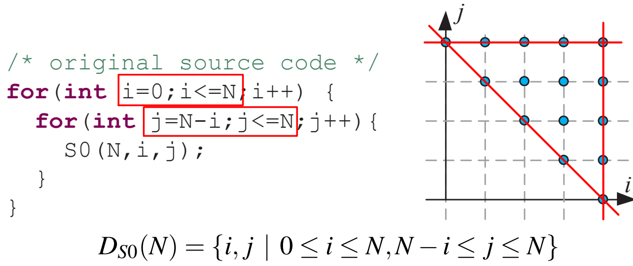


Fig. 4: Example loop nest and its corresponding polyhedral representation.

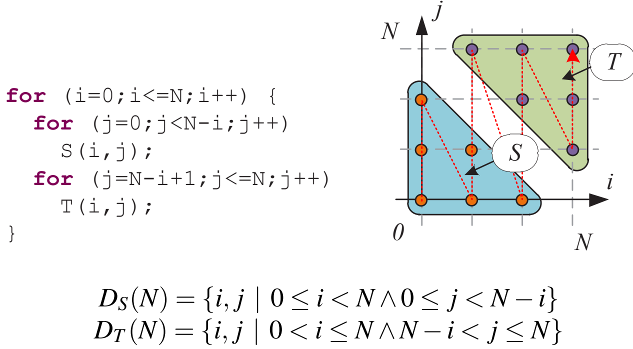


Fig. 5: Imperfect loop nests are represented as an union of polyhedra. The same theory applies on this union.

$$\texttt{succ}(\vec{n}, \vec{x}) = \{\vec{y} \mid \vec{y} \succ \vec{x}, \vec{y} \in D_S(\vec{n}\}$$
$$\texttt{next}(\vec{n}, \vec{x}) = \texttt{lexmin}(\texttt{succ}(\vec{n}, \vec{x}))$$

We use the Integer Set Library [12] to find the lexicographic minimum. The result is a piece-wise affine function representing the `next` function. The representation of the `next` function for the example in Figure 4 is shown in Figure 6.

IV. FSM Code Generation

The `next` function is an abstract representation of the transition required to generate an FSM. In this section, we describe how to generate code out of this function while targeting hardware synthesis.

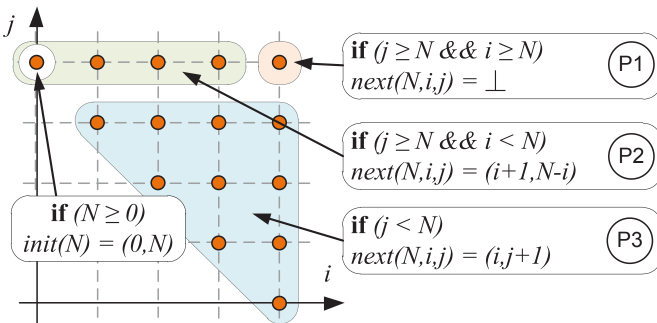


Fig. 6: Illustration of the `init` and `next` functions for the example in Figure 4. $P1$ through $P3$ are the different pieces of the `next` function.

```
// (1) Initialize
done = 0;
x = init(n);
while (!done) {
  // (2) Compute Controls
  bool ctrl_1 = f1(n,x);
  bool ctrl_2 = f2(n,x);
  ...
  // (3) Commands
  if (ctrl_1) S0(n,x);
  ...
  // (4) Compute Next
  if (ctrl_2) x_next=next_1(n,x);
  if (ctrl_3) x_next=next_2(n,x);
  ...
  if (ctrl_4) done = 1;
  // (5) Fire
  x = x_next;
}
```

Fig. 7: Template used when generating code from the polyhedral representation of the FSM.

A. Code Template

We use the code template illustrated in Figure 7 in order to maximize the Common Sub-expression Elimination (CSE) opportunities and to enable the parallel evaluation of the guards. The five steps are described in the following.

1) The FSM initialization consists in assigning values to the iterators according to the parameters values. The `init` function is built by computing the lexicographic minimum of the iteration domain. Figure 6 illustrates the `init` function for the example in Figure 4.

2) All the affine constraints to be evaluated for the transitions are gathered before the guards. The aim is to make the application of CSE easier.

3) The FSM commands execute the operations from the original loop body. Since the FSM iterates over the whole iteration domain, without distinguishing statements, we insert guards to execute the appropriate statements.

4) As illustrated on the Figure 6, the `next` function is a piece-wise affine function. Each piece represents a different transition with a different expression for the immediate successor. Using the last transition means the iteration has been completed. In comparison with the method of Boulet and Feautrier, we do not use the `else` construct. Because of the complexity of the guards, the HLS tools are not always able to determine that the pieces are guarded by mutually exclusive domains. Therefore the `else` construct would insert a dependency chain, lengthening the critical path. By avoiding such construct, all the guards can be evaluated in parallel.

5) Finally, the FSM transition is fired by updating the values of the iterators.

```
//Initialize
done = 0;
if (N >= 0) { i = 0; j = N; }
else done = 1;
while (!done) {
  //Compute Controls
  bool ctrl_a0 = ((j == N) && (N >= i+1));
  bool ctrl_a1 = (N >= j+1);
  bool ctrl_a2 = !(ctrl_a0 || ctrl_a1);
  //Commands
  S0(N,i,j);
  //Compute Next
  if (ctrl_a0) { i_n1 = i+1; j_n1 = -i+j-1;}
  if (ctrl_a1) { i_n1 = i; j_n1 = j+1;}
  if (ctrl_a2) done = 1;
  //Fire
  i = i_n1; j = j_n1;
}
```

Fig. 8: The final code for the example in Figure 4.

B. Merging of Pieces

One important optimization when implementing FSMs in hardware is to minimize the number of branches in transitions and the number of conditions to evaluate. The `next` function found by the library often contains pieces that can be merged, but are not found by the simplification implemented in the library.

As an example, the original `next` function found by ISL have two pieces that together express the piece `P1` in Figure 6:

(1) $\texttt{next}(N,i,j) = (i, j+1)$ if $i < N$ & $j < N$
(2) $\texttt{next}(N,i,j) = (N, j+1)$ if $i = N$ & $j < N$

Although it is not difficult to identify that the two pieces can be merged into one, in the eyes of a human, more complicated variations of such cases can be difficult to find.

These simplification opportunities can be characterized as when the two expressions are equivalent in context. In the above example, $(i, j+1)$ and $(N, j+1)$ is not equivalent in general, but given the context $i = N$, they are equivalent. This is the key reason why the two pieces can be merged.

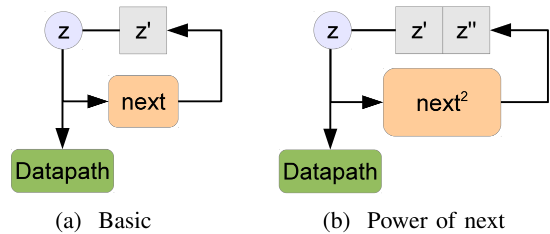
We use this notion of equivalence to find pieces that can be merged. Although this does not necessarily minimize the number of transitions and/or conditions to evaluate, we found this optimization to be highly effective in reducing the number of transitions.

C. Final Output

The FSM code generated for the example in Figure 4 is shown in Figure 8. Since the loop nest contains only one statement, there is no need for guards in the command part.

V. State Look Ahead

The two important factors that impact the effectiveness of nested loop pipelining are the ratio of pipeline latency against innermost loop trip count (α), and degradation in frequency (f^*). If we manage to limit the degradation in frequency, then nested loop pipelining becomes beneficial to smaller α, not to mention that its general performance is improved. In this section, we propose to look ahead multiple states rather than the immediately following state to improve the achievable frequency of FSM implementations.



(a) Basic (b) Power of next

Fig. 9: Illustration of lookahead using power of the `next` function. In the basic implementation, the `next` function is used to compute the immediate following iteration. Using $\texttt{next}^2$ allows two iterations ahead to be computed, with a two stage pipeline.

A. Pipelining the Computation of Next Iterations

When the data-path is pipelined, the control-path, which is the computation of next state, may become the bottle neck, since each stage of the pipeline can be shorter than the computation of the next iteration. Thus, we seek to enable pipelining of the control-path to improve the overall critical path length. It is necessary to compute multiple iterations ahead in order to pipeline the control-path.

The key is to compute the n-th power of the `next` function so that n iterations ahead are directly computed. We expect computing $\texttt{next}^n$ to be less expensive than applying `next` n times. If this is the case, pipelining the computation of $\texttt{next}^n$ using n stages reduces the critical path length.

B. Implementation of Look Ahead

The high-level structure of the control-path implementing look ahead is depicted in Figure 9. Additional registers are introduced to hold the value of future iterations (and hence becoming a shift register). The transformed C code is nearly identical to the one without lookahead, except for copying of values corresponding to the additional registers.

With the optimizations described in Section IV-B, the number of transitions as well as constraints to evaluate are kept small. Thus, evaluating $\texttt{next}^n$ involves less work than applying `next` n times. For instance, the `next` function for a 2-dimensional rectangular domain has 2 transitions with total of 3 unique constraints to evaluate. The $\texttt{next}^2$ function has 3 transitions with 4 unique constraints, rather than 4 transitions with 6 constraints, which would be the case if the `next` function was composed.

C. Final Output with Look Ahead

The final code for the example in Figure 4 with 2-state look ahead is shown in Figure 10. The overall structures of the code stays the same, but now the HLS tools can pipeline the computation of the `next` function.

For this example, the merging of pieces works particularly well. Notice that there are only three transitions with 2-state

```
//Initialize
done = 0;
if (N >= 1)
{i = 0; j = N; i_n1 = 1; j_n1 = N-1;}
else done = 1;
while (!done) {
  //Compute Controls (look ahead: 2)
  bool ctrl_a0 = ((j >= N-1) && (N >= i+1));
  bool ctrl_a1 = (N >= j+2);
  bool ctrl_a2 = !(ctrl_a0 || ctrl_a1);
  //Shift done flag
  done = done_1;
  //Commands
  S0(N,i,j);
  //Compute Next (look ahead: 2)
  if (ctrl_a0) {i_n2 = i+1; j_n2 = -i+j;}
  if (ctrl_a1) {i_n2 = i; j_n2 = j+2;}
  if (ctrl_a2) done_1 = 1;
  //Fire
  i = i_n1; j = j_n1;
  i_n1 = i_n2; j_n1 = j_n2;
}
```

Fig. 10: The final code for the example in Figure 4 with 2-state look ahead. Note that there is an additional set of loop iterators, `i_n2` and `j_n2`. The computation of `next` now give two states ahead. (Notice the $j+2$ in the second piece.)

look ahead, even though 1-state look ahead already had three transitions.

Since now we the `next` function computes two states ahead, the are two cases at the boundary of j:

- $(i+1, N-i)$ if $j = N$
- $(i+1, N-i-1)$ if $j = N-1$

The former case is when j is exactly on the boundary, so that the iteration two steps ahead is the second iteration in the next column. The latter case reaches the first iteration in the next column, since the current iteration is still one step away from the boundary. However, substituting N with j for the first case, and $N-1$ with j for the second case gives $(i+1, j-i)$ as the common expression to compute the `next` state. Our implementation is able to find such cases and simplify the state transitions using the equivalence in context check.

VI. Experimental Validation

In this section, we present the results of our experiments to show that the combination of FSM with state look ahead leads to improved frequency.

A. Benchmark Kernels

We used four simple kernels that capture commonly seen loop structures, with trivial data-path to focus on the control-path synthesis. All kernels access an array of integers and simply increment each element by one. Each iteration of the loop access distinct elements in the array, and thus there is no dependence between loop iterations. The four loop structures we model are:

- `rect 2d`: 2-dimensional loop nest with loops bounded from above by N and M forming a rectangle.
- `rect 3d`: 3-dimensional version of the above, forming a cuboid.
- `triangular 2d`: 2-dimensional loop nest with the inner loop bounded by the outer loop indices (i.e., $j \leq i$), forming a triangle.
- `triangular 3d`: 3-dimensional version of the above, forming a tetrahedron.

These loop structures are by no means comprehensive, but many kernels for embedded applications, such as image processing, matrix multiplication, QR decomposition, and so on, fall under one of these loop structures.

B. Experimental Setup

The kernels described above were processed through our prototype implementation of the flow that automatically generates C code that implements FSM-based controls. The original code and the generated codes were given to a HLS tool[1] that produce VHDL code. The HLS tool we used is the only tool that we are aware of that can perform automated nested loop pipelining. We have selected this tool to compare the performance of our approach with the nested loop pipelining currently offered by tools.

The four versions we obtain are:

- `SLP`: Single Loop Pipelining. The original code with the inner-most loop pipelined by the HLS tool.
- `NLP`: Nested Loop Pipelining. The original code with the entire loop nest (2D or 3D) flattened and pipelined by the HLS tool.
- `FSM-LA1`: FSM-based control by our tool with state look ahead of 1, that is, to computes the immediate next iteration. The `while` loop in the generated code was pipelined by the HLS tool.
- `FSM-LA2`: FSM-based control by our tool with state look ahead of 2. The `while` loop in the generated code was pipelined by the HLS tool.

The HLS tool was configured to target Altera Stratix IV FPGA with effort put on latency. We run the HLS process a number of times with different target frequency to find the highest target frequencies (in 10MHz intervals) where the HLS tool finds a feasible schedule. Then the generated VHDLs with the top few target frequencies were synthesized using Quartus.

C. Highest Target Frequency

Figure 11 show the highest target frequencies that the HLS tool managed to schedule. The version with single loop pipelining, `SLP`, reaches the maximum frequency reachable by the target FPGA (450MHz). The nested loop pipelining by the HLS tool is comparable to the FSM-based control with state look ahead of 1.

With look ahead of 2, there is a noticeable improvement in the target frequency. This supports our hypothesis that the computation of two iterations ahead is faster than computing the next iteration two times.

[1] We cannot disclose the name of the HLS tool we used due to its licensing.

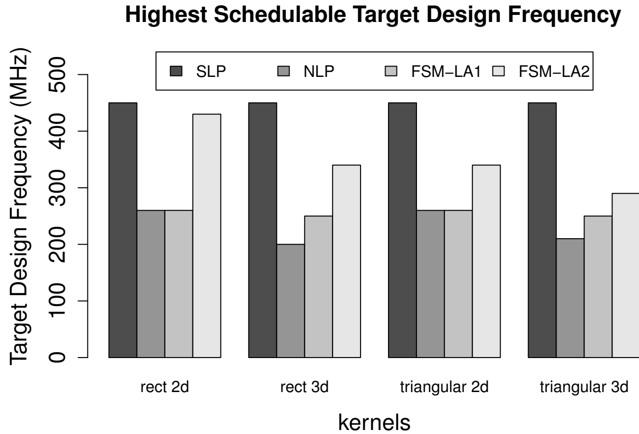


Fig. 11: Highest design frequency where the HLS tool was able to find a schedule for different implementations of the control path and pipelining. The results show that using look ahead improves the achievable frequency as we expect.

D. Synthesized Hardware

Table I gives a summary of the hardware synthesized by Quartus. Using FSM-based control with 2-state look ahead, we see 40% to 50% improvement in design frequency compared to nested loop pipelining performed by the HLS tool. Frequency improvement by 40-50% translates to around 30% reduction in total execution time.

This is with the exception of `triangular 3d`, where Quartus reported 324MHz whereas the target frequency by the HLS tool is only 210MHz. The frequency generally match the trend in seen in the highest target frequency for other kernels.

The area cost is comparable for those that only compute the immediate next iteration. `FSM-LA2` that computes two iterations ahead use around twice as much area compared to the other versions. The increase in area is a combination of the cost of adding pipeline stage to the control-path and the increased complexity of the control due to look ahead.

Note that since there is very little data-path for the kernels we used, and thus most of the area comes from the control-path. With designs with more complicated data-path, the impact of area overhead in the control-path decreases.

E. Impact on Effectiveness of Nested Loop Pipelining

The performance of nested loop pipelining can be significantly improved by using FSM-based control with pipelined control-path. However, this is not the end of the story. Recall the analytical model developed in Section II. The performance of nested loop pipelining should be compared with single loop pipelining, since nested loop pipelining is only beneficial with relatively small trip counts.

As illustrated in Figure 2, both the ratio between the pipeline latency and inner most loop trip count (α) and the design frequency influence if nested loop pipelining is beneficial. For example, if the frequency degrades by 50%, α must be 1,

TABLE I: Hardware characteristics of different implementations. FSM with 2 state look ahead is faster than others that perform nested loop pipelining, but uses close to twice as much area due to pipelining of the control-path.

Kernel	Version	ALUT	Logical Register	Frequency (MHz)
rect 2d	SLP	98	91	525
	NLP	116	93	322
	FSM-LA1	106	62	349
	FSM-LA2	201	182	454
rect 3d	SLP	160	145	500
	NLP	158	115	260
	FSM-LA1	199	94	334
	FSM-LA2	348	215	387
triangular 2d	SLP	94	83	532
	NLP	82	65	286
	FSM-LA1	89	90	460
	FSM-LA2	131	111	433
triangular 3d	SLP	163	137	524
	NLP	154	113	324
	FSM-LA1	187	71	259
	FSM-LA2	384	290	346

i.e., the trip count must be comparable to pipeline latency, for nested loop pipelining to be no slower than single loop pipelining. We limit the degradation to around 20% for most cases, which in turn means that the break-even point is when $\alpha = 0.2$, or when the trip count is 5 times larger than the pipeline latency.

In practice, the data-path is much more complicated than the kernels we used to focus on control-path. The HLS tools now have to aggressively pipeline the data-path to achieve higher frequency, or have to operate on lower frequency. In both cases, this favors nested loop pipelining, by further reducing the degradation in frequency, or by increasing the pipeline latency.

Indeed, even with simple computations such as matrix product, or morphology (image processing), the frequency for single loop pipelining and 2-state look ahead becomes the same. In this case, nested loop pipelining is at least as good as single loop pipelining even with large innermost loop trip count, and significantly faster with small trip counts.

VII. RELATED WORK

In this section, we relate our work with previous research, and highlight the novelty of our approach.

A. Nested Loop Pipelining

The benefits of nested loop pipelining have been repeatedly demonstrated for programmable processors [2], [3], and for hardware synthesis [4], [13]. These work also try to limit the degradation in performance by proposing dedicated facilities, and by proposing designs for efficient execution of coalesced loop nests.

However, the class of loop nests are limited to rectangular bounds, and some of the earlier work is further limited to constant bounds and/or perfectly nested loops. In contrast, we can handle any loop nest with affine bounds, and imperfect loop nests are not an issue.

Morvan et al. [14] use a similar method for loop coalescing. However, their paper primarily focus on the legality of nested loop pipelining, and its correction. They also do not perform any state look ahead, which was shown to be an important optimization of the control-path.

B. Zero-overhead Looping

A related concept is zero-overhead loops, where the aim is to improve the performance of loops by eliminating the bound checks. In programmable processors, this is achieved by using a specialized architecture for zero-overhead loops that are configured at run-time before entering loop nests [6], [7]. However, such architecture is also limited to rectangular loop nests.

VIII. Conclusion

In this paper we have presented a method for improving the effectiveness of nested loop pipelining. We borrow a technique from automatic parallelization to derive efficient FSM-based controls for imperfect loop nests with affine bounds; a much wider class of loops than most previously proposed techniques. The performance of derived control is further improved by pipelining of the control-path, enabled by state look ahead.

Our approach increases the applicability of nested loop pipelining in two ways. The increase in achievable frequency widen the range of input data sizes where nested loop pipelining is beneficial. We also handle more general class of loop nests enlarging the design space.

As the HLS tools gain maturity, source-to-source transformations originally developed for general purpose processors are now being applied to HLS [14], [15], [16]. Although direct application of loop transformations for general purpose processors may also be beneficial for hardware design, we believe that there is a need to adopt such techniques for HLS context. The derivation of FSM is one such example, as FSM-based implementation is usually considered inefficient for general purpose processors.

Acknowledgements

This work was funded in part by the INRIA STMicroelectronics Nano2012-S2S4HLS project, and the European Commission Seventh Framework Programme (FP7/2007-2013) with Grant Agreement 287733.

References

[1] T. Hussain, M. Pericàs, N. Navarro, and E. Ayguadé, "Implementation of a reverse time migration kernel using the hce high level synthesis tool," in *Proceedings of the International Conference on Field Programmable Technology*, 2011, pp. 1–8.

[2] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-Dimension Software Pipelining for Multidimensional Loops," *ACM Trans. on Architecture and Code Optimization*, vol. 4, no. 1, pp. 7:1-7:12, 2007.

[3] T. Yu, Z. Tang, C. Zhang, and J. Luo, "Control mechanism for software pipelining on nested loop," in *Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, 1997, pp. 345–.

[4] B. Ylvisaker, C. Ebeling, and S. Hauck, "Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests," University of Washington, Tech. Rep., 2010.

[5] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems," in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, pp. 52–.

[6] N. Kavvadias and S. Nikolaidis, "Elimination of overhead operations in complex loop structures for embedded microprocessors," *Computers, IEEE Transactions on*, vol. 57, no. 2, pp. 200–214, 2008.

[7] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, Y. Paek, V. Cao, and C. Burns, "Compiler transformations for effectively exploiting a zero overhead loop buffer," *Software: Practice and Experience*, vol. 35, no. 4, pp. 393–412, 2005.

[8] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, vol. 26, no. 7, 1991, pp. 39–50.

[9] Fabien Quilleré and Sanjay Rajopadhye and Doran Wilde, "Generation of Efficient Nested Loops from Polyhedra," *International Juornal on Parallel Programming*, vol. 28, no. 5, pp. 469–498, 2000.

[10] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, 2004, pp. 7–16.

[11] P. Boulet and P. Feautrier, "Scanning Polyhedra without Do-loops," in *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 1998, p. 4.

[12] S. Verdoolaege, "ISL: An Integer Set Library for the Polyhedral Model," in *Proceedings of the International Congress on Mathematical Software*, 2010, pp. 299–302.

[13] H.-S. Yun, J. Kim, and S.-M. Moon, "Time Optimal Software Pipelining of Loops with Control Flows," *International Journal of Parallel Programming*, vol. 31, pp. 339–391, 2003.

[14] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 3, pp. 339–352, 2013.

[15] C. Alias, A. Darte, and A. Plesco, "Optimizing remote accesses for offloaded kernels: application to high-level synthesis for fpga," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013, pp. 575–580.

[16] J. a. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "Lara: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, ser. AOSD '12, 2012, pp. 179–190.