Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS

Junyi Liu, Samuel Bayliss, George A. Constantinides
Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
{junyi.liu13, s.bayliss08, g.constantinides}@imperial.ac.uk

Abstract—Loop pipelining is probably the most important optimization method in high-level synthesis (HLS), allowing multiple loop iterations to execute in a pipeline. In this paper, we extend the capability of loop pipelining in HLS to handle loops with uncertain memory behaviours. We extend polyhedral synthesis techniques to the parametric case, offloading the uncertainty to parameter values determined at run time. Our technique then synthesizes lightweight runtime checks to detect the case where a low initiation interval (II) is achievable, resulting in a run-time switch between aggressive (fast) and conservative (slow) execution modes. This optimization is implemented into an automated source-to-source code transformation framework with Xilinx Vivado HLS as one RTL generation backend. Over a suite of benchmarks, experiments show that our optimization can implement transformed pipelines at almost same clock frequency as that generated directly with Vivado HLS, but with approximately 10× faster initiation interval in the fast case, while consuming approximately 60% more resource.

I. INTRODUCTION

High-level synthesis (HLS) tools have recently reached commercial maturity, enabling high hardware design productivity for field-programmable gate array (FPGA) technology. However, for many applications, there is still a considerable gap between the quality of results produced by HLS tools and those obtained by manual optimized RTL design. Computational bottlenecks are typically located in some critical loops of high-level programs, and hence loop pipelining has emerged as one of the preeminent optimization techniques in HLS.

\[
\text{for (int } i=LB; i<=UB; i++) \quad A[i] = A[i+m] + 1;
\]

Fig. 1: Motivational code.

The optimization method of this paper addresses the issue of loops with uncertain data access patterns, which causes existing commercial HLS tools to take an overly conservative approach to pipeline scheduling. In the motivation loop shown in Fig. 1, there is one uncertain variable \(m\) whose value is not known at compile time, in the read access pattern of array \(A\). The loop iterator \(i\) is bounded by two constant bounds \(LB\) and \(UB\). Whether the loop can be pipelined actually depends on the value of the parameter \(m\). If \(m=-1\), the result of each iteration has to be generated before the start of the next iteration, which implies an inter-iteration dependency. If \(m>0\), there will be no recurrence in this loop. This uncertain data dependency prevents modern HLS tools from exploiting loop pipelining by default.

This is the basic idea of our approach: synthesize a lightweight runtime check (in this case \(m=0\)) that switches the pipeline schedule. These lightweight checks can be introduced, alongside appropriate loop-pipelining directives, through source-to-source transformation applied before invoking a commercial HLS tool. The rest of this paper explains how this idea can be generalized into polyhedral analysis and automated within a tool flow.

II. RELATED WORK

In the recent work of loop pipelining in [1] and [2], the authors rely on knowing, at compile time, all the dependencies that exist between operations to exploit pipeline scheduling. There are also active HLS research efforts as in [3], [4] investigating loop pipelining for loops with irregular behaviours and structures. Polyhedral optimization has also been popularly applied for optimizing custom memory systems in recent HLS research. The previous works in [5]–[8] apply polyhedral analysis to study memory reuse or partition problems for improving loop latency and parallelism.

III. MOTIVATION

A. Loop Pipelining

Loop pipelining is implemented by overlapping the execution of loop iterations. Where read-after-write loop-dependencies exist in the original code (a value is written in one iteration and read in a subsequent iteration), a pipelined schedule must be constrained to preserve these dependencies [1], [2]. The constant interval between the start of successive iterations is called the initiation interval (II), and reflects the degree of parallelism, in the sense that for the same latency, a pipeline with smaller II has more iterations running in parallel at any given clock cycle. If we denote the latency of a single loop iteration and the loop trip count as \(L\) and \(N\) respectively, then the computation delay of the whole loop is equal to \(L + (N - 1) \times II\). When \(N\) is large enough, the loop delay is approximately equal to \(N \times II\). Therefore, the performance of a loop is mainly related to its \(II\). Unlike resource constraints that may vary with the requirements of different hardware implementations, loop-dependency constraints implied in the loops are quite intrinsic. A complex dependency constraint could significantly constrain our ability to reduce the II of a loop pipeline.

B. Loop Dependence Analysis

To analyze the data dependencies of a loop, we need to formally model the memory access sequence. These patterns are described by loop bounds and array indexing functions, in
which uncertain variables may participate. Two paired accesses are dependent if and only if the data point written in the current iteration will be read in a future iteration. If \( p \) is a vector of uncertain variables, the dependence iteration distance \( \delta(p) \) is the smallest number of iterations between the execution of such two dependent memory accesses.

Since the dependence iteration distance is variable in our target loops, we can evaluate a safe region of \( \delta(p) \), where no read access ever executes before the completion of its dependent write access during pipeline execution. The motivational example loop in Fig. 1 is used for illustration. As shown in Fig. 2, according to the given loop scheduling, the latency \( L \) between read access \( A[i+m] \) and write access \( A[i] \) in iteration \( i \) is the period when the start of dependent read access \( A[i+\delta(p)+m] \) in iteration \( i+\delta(p) \) will violate the inter-iteration loop dependency. If the target initiation interval is equal to \( II \), the number of iterations scheduled to be executed in latency \( L \) is equal to \( \lceil L/II \rceil - 1 \). To avoid the conflict of inter-iteration dependency, the iteration distance \( \delta(p) \) should satisfy one of the conditions in (1), which denotes the safe region of \( \delta(p) \). Intuitively, dependences between a write and a future read should either not exist (non-positive \( \delta(p) \)) or should be enough iterations away that they do not impact scheduling decisions.

\[
\begin{align*}
\delta(p) & \leq 0 \\
\delta(p) & \geq \lceil L/II \rceil
\end{align*}
\]

(1)

C. Proposed pipeline optimization

In current HLS tools, only the worst case of uncertain data dependency is considered for loop pipelining. To fully unleash the potential of parallelism, we can implement conditional loop pipelining in the hardware, which is optimized for runtime performance. The conceptual architecture of proposed pipeline is shown in Fig. 3. The pipeline is able to speed up when the safe region detector determines that loop dependency does not limit the loop parallelism. Since the conditions of safe region can be calculated at compile time with our method, the hardware complexity of the detection logic can be minimized.

IV. POLYHEDRAL RUNTIME OPTIMIZATION

A. Parametric polyhedral analysis of safe region

In this work, we will use a parametric polyhedral model to analyze the uncertain loop dependences. An example loop shown in Fig. 4 is analyzed as an illustration.

The uncertain variables in the memory access patterns can be represented by a parameter vector \( p \in \mathbb{Z}^{4p} \) of the polyhedral model, where \( d_p \) is the number of uncertain variables. In the example loop shown in Fig. 4, \( n \) and \( m \) are two uncertain variables, used in a loop indexing expression respectively. The iteration domain of this sample loop can be represented by an inequality constraint shown below.

\[
\begin{pmatrix}
-1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 1 & 0 & -1
\end{pmatrix}
\begin{pmatrix}
i \\
j \\
m \\
n
\end{pmatrix}
\leq
\begin{pmatrix}
-LB \\
UB \\
-LB \\
0
\end{pmatrix}
\]

The memory read access \( A[i-1][j+m] \) in Fig. 4 has an array indexing expression with one uncertain variable \( m : [i-1,j+m]^T \), where

\[
\begin{pmatrix}
i-1 \\
j+m
\end{pmatrix} = \begin{pmatrix}1 & 0 \end{pmatrix} \begin{pmatrix}i \\
j
\end{pmatrix} + \begin{pmatrix}0 & 0 \end{pmatrix} \begin{pmatrix}m \\
n
\end{pmatrix} + \begin{pmatrix}1 \end{pmatrix}
\]

By linking the array indexing expressions of one write access and one read access to the same array, we can obtain the iteration dependency map between two dependent iterations accessing the same memory element, which signifies the possible presence of a data dependence. The equality constraint of the iteration dependency map can be used to formulate the dependence vector distance function for \((v' - v)\), where \(v\) and \(v'\) represent the iteration vectors of memory write (source) and read (sink) accesses respectively. For the example loop in Fig. 4, the dependence vector distance function of the memory write access \( A[i][j] \) is

\[
v' - v = -\begin{pmatrix}0 & 0 \\
1 & 0
\end{pmatrix} \begin{pmatrix}m \\
n
\end{pmatrix} + \begin{pmatrix}1 \end{pmatrix} = \begin{pmatrix}1 & -m
\end{pmatrix}
\]

Knowing a time stamp function \( t(p) \), we calculate \( t(p)(v' - v) \) to transform the vector distance into the scalar form, which is equivalent to the dependence iteration distance \( \delta(p) \) introduced in Section III. Therefore, the conflict region \( C \) is a set of parameter values \( p \), such that \( \delta(p) \) does not satisfy either condition in (1).

The example loop in Fig. 4 has the time stamp function \( t(p)^T = [n - LB + 1, 1] \), so that \( t(p)^T(v' - v) = n - LB + 1 - m \). This result indicates that the write access \( A[i-1][j+m] \) of each iteration has its dependent read access \( A[i][j] \)
appearing in its next \((n - m - LB + 1)\)th iteration. Therefore, the conflict region of \(p\) in this example loop is

\[
C = \{[m, n]^T | 1 \leq n - m - LB + 1 \leq [L/II] - 1 \land [m, n]^T \in Z^2\}.
\]

With this \(C\) of the example loop generated, two constraints of \(p\) for safe region can be easily obtained as

\[
\begin{align*}
    n - m & \leq LB - 1 \\
    n - m & \geq [L/II] + LB - 1
\end{align*}
\]

The previous constraints include those vectors \(p \notin C\) when iteration dependency map exists. In addition, there are potential constraints of \(p\) for safe region includes those vectors \(p\) resulting in no iteration dependency, i.e. for which iteration dependency map does not exist, which is \(n + m \leq LB - 1\) in the example loop. This kind of constraint corresponds to the case that \(A[i-1][j+m]\) in any iteration does not read the data written by \(A[i][j]\) in any previous iteration.

### B. Source-to-source Transformation

The polyhedral analysis to generate the safe region introduced above is implemented as an algorithm using the Integer Ser Library (ISL) [9]. To make our new loop optimization compatible with a commercial HLS tool, we integrated our analysis algorithm into a source-to-source code transformation framework shown in Fig. 5. In this paper, we select Xilinx Vivado HLS as the HLS back-end tool to generate RTL code from original and transformed C code. The HLS tool is firstly used to synthesize the original code to generate the scheduling information for the future loop analysis. The loop information is captured by two open-source tools. The Clang front-end parser [10] generates an abstract syntax tree (AST) from the input C code. The Polyhedral Extraction Tool (PET) [11] extracts the loops as the static control parts (SCoPs) with ISL from Clang AST. PoTHoLeS [12] is a polyhedral compilation tool developed by us based on ISL, which conducts user-specified loop analysis and transformation. Finally, the transformed C code is generated by PoTHoLeS. Fig. 6 illustrates the code transformation result of our framework. The input 2D loop has constant loop bounds and one uncertain variable \(m\) in write access \(A[j] [i - m]\). Currently, the potential violation of uncertain memory access patterns to real memory bounds is not considered in the generation of safe region.

Fig. 5: Tool flow of code transformation framework.
TABLE I. Pipeline performance and resource usage results.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clock (ns)</th>
<th>Iteration Cycles</th>
<th>Initiation Interval</th>
<th>Before Loop Cycles</th>
<th>LUT</th>
<th>FF</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig Tran</td>
<td>Orig Tran_S</td>
<td>Orig Tran_F</td>
<td>Orig Tran_S</td>
<td>Orig Tran_F</td>
<td>Orig Tran</td>
<td>Orig Tran</td>
<td>Orig Tran</td>
</tr>
<tr>
<td>typ_loop</td>
<td>3.574</td>
<td>3.272</td>
<td>20</td>
<td>20</td>
<td>21</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>row_col</td>
<td>3.146</td>
<td>3.100</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>pivot</td>
<td>2.556</td>
<td>2.523</td>
<td>49</td>
<td>-</td>
<td>55</td>
<td>47</td>
<td>-</td>
</tr>
<tr>
<td>tri_sp_slv</td>
<td>3.137</td>
<td>2.888</td>
<td>25</td>
<td>25</td>
<td>28</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>jacobi_2d</td>
<td>2.523</td>
<td>2.887</td>
<td>55</td>
<td>55</td>
<td>60</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>adi_int</td>
<td>3.889</td>
<td>3.634</td>
<td>68</td>
<td>63</td>
<td>66</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.098</td>
<td>1.099</td>
<td>1.06</td>
<td>1.00</td>
<td>0.10</td>
<td>1.50</td>
<td>1.70</td>
</tr>
</tbody>
</table>

Table I provides detailed results of pipeline performance and resource usage. “Tran_S” and “Tran_F” correspond to the slow and fast execution modes of transformed pipelines. Column title “Cycles Before Loop” means the number of cycles for the operations executed before the start of the pipelines. It should be noted that the benchmark pivot has the uncertain variables analyzed to be always inside the safe region. Our implementation only generates the fast pipeline mode for pivot.

Firstly, the HLS tool is able to achieve clock periods of transformed pipelines very close to their original implementations. The fast pipeline mode has the geometric mean of single iteration latency increased by 6%, which is due to more opportunities for HLS scheduling to achieve a small initiation interval. Having proved the absence of data dependencies in our analysis, the backend tool is instructed to ignore all dependencies using a #pragma. This allows the geometric mean initiation interval of the fast pipeline mode to be 10× faster than the original code. According to Fig. 3, the detector logic is observed to increase the geometric mean “Cycles Before Loop” by 50%. However, this is still a small number of cycles if the loop body has a fairly large number of iterations, which also indicates that the complexity of the detector logic is lightweight.

As shown in Table I, the overall resource increase is around 60%. Vivado HLS is observed to be able to apply resource sharing for floating point units and memory ports implied in two different loop modes. Meanwhile, some other resources such as memory address calculation are not shared by Vivado HLS, which contributes to a big part of resource overhead of our transformed pipelines. Besides, the increase of DSPs is also related to the increase of parallelism that requires more operations running at the same time, especially for those large loops like jacobi_2d and adi_int.

VI. CONCLUSION

In this paper, we proposed a new optimization method for one class of loops with uncertainty. This method combines compiler-based analysis and runtime optimization. With the experiments over a suite of benchmarks, we show that the fast runtime mode of the optimized pipelines reduces the initiation interval by 90%, i.e., a 10× speedup. This comes at the cost of 60% increase in resource usage. In future work, we intend to lift the restriction of scheduling that is linear in the uncertain parameters, allowing for more complex loop iteration spaces to be analyzed.

ACKNOWLEDGEMENT

The authors acknowledge the support of Innovation Technologies, the Royal Academy of Engineering, and EPSRC (EP/K034448/1, EP/I020357/1, EP/I012036/1).

REFERENCES