Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis

Mythri Alle
IRISA/University of Rennes 1
mythri.alle@irisa.fr

Antoine Morvan
INRIA/ENS cachan
antoine.morvan@inria.fr

Steven Derrien
IRISA/University of Rennes 1
steven.derrien@irisa.fr

ABSTRACT

Research on High-Level Synthesis has mainly focused on applications with statically determinable characteristics and current tools often perform poorly in presence of data-dependent memory accesses. The reason is that they rely on conservative static scheduling strategies, which lead to inefficient implementations. In this work, we propose to address this issue by leveraging well-known techniques used in superscalar processors to perform runtime memory disambiguation. Our approach, implemented as a source-to-source transformation at the C level, demonstrates significant performance improvements for a moderate increase in area while retaining portability among HLS tools.

1. INTRODUCTION

High-Level Synthesis (HLS) enables the derivation of custom hardware from high-level algorithmic specification (in C, C++, SystemC, etc.) There exists several robust and mature HLS tools [1, 2] used as production tools by world-class chip vendor companies. Even though these tools provide impressive improvement in productivity, there is a large gap between "accepted" codes and "efficiently handled" codes. They hence rely on the designer to deeply restructure the program source code and to use sophisticated compiler directives (usually in the form of `#pragmas`) to drive the synthesis flow.

Many of the techniques used in HLS borrow from earlier research results on optimizing compiler back-end for DSP or VLIW processors. In such a compiler, instructions are statically scheduled. Such an approach prevents scheduling optimization opportunities for HLS tools. This choice in design flow has not yet been questioned and this is easy to explain: the application domain targeted by HLS mostly consists of kernels for which the aforementioned approaches perform relatively well.

However, as HLS usage becomes widespread, it is likely that user expectations and target application domains will expand beyond their current needs. For example, current generation HLS tools perform poorly when trying to schedule computations that have data-dependent memory accesses. For these kernels, it is generally not possible to assert at compile time that two array/memory accesses will *never* alias[1], even if they very rarely (or even never) do in practice. In such situations, HLS tools fallback to a conservative (worst case static) scheduling, which guarantees correctness but is inefficient.

While this problem may currently be perceived as a *corner case* by HLS tool providers, we believe that improving the support for such dynamic behavior is important to broaden the use of HLS to more application domains. For example, recent research work advocated the use of FPGA for accelerating data-analytic applications [3] and sparse linear algebra operations [4]. Both of these domains contain data-dependent memory accesses. For such application domains, providing efficient high-level design tools is a key issue and current HLS tools are clearly not ready for that.

In this work, we study how dynamic scheduling techniques can be used to improve the efficiency of HLS tools in the presence of complex data-dependent memory accesses. More precisely, we make the following two contributions:

- We propose a technique based on *runtime memory disambiguation* to improve the efficiency of loop pipelining in the presence of data-dependent memory dependencies.
- We implement our approach as a semi-automatic (i.e., user driven) source-to-source transformation, enabling portability among HLS tools.

We validate our approach on a set of representative kernels using two leading-edge HLS tools, with FPGA as target technology. Our results show that our technique can lead to significant improvement in throughput at the price of moderate area overhead.

The remainder of the article is organized as follows. Section 2 describes the problem we address in this work. Section 3 describes our technique and its implementation. Section 4 provides experimental validation of our approach. Conclusion and future direction are discussed in Section 5. We provide detailed algorithms and results in Appendix A and Appendix B respectively. Appendix C presents a survey of related work. Current limitations of our approach are discussed in Appendix D.

2. PROBLEM STATEMENT

Loop pipelining is a key optimization in High-Level Synthesis tools. It builds on the software pipelining technique

DAC '13, May 29-Jun 07 2013, Austin, TX, USA
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.

[1] That is, they will never access the same memory location.

proposed by Lam et al. back in 1988 [5]. In this technique, parallelism across loop iteration is exploited by initiating the next iteration of the loop before the completion of the current iteration. This allows instructions from several iterations to overlap and hence improve throughput. The throughput achieved is limited by the delay between initiations of two successive iterations. This delay is called as the initiation interval(II). Typically II is determined both by hardware constraints and data dependencies in the application. However, since HLS tools are not constrained by a predefine micro-architecture, data dependencies alone determine the minimum II that can be achieved. Thus, the efficiency of loop pipelining in the context of HLS is dependent on the accuracy of dependency analysis provided by these tools. Most HLS tools employ basic (hence conservative) dependency analysis techniques, which limit the applicability of loop pipelining.

For example, consider the loops shown in Figure 1. Loop `L1` contains a non-uniform, non-affine dependency that is difficult to analyze. Hence, HLS tools are forced to make a conservative assumption that the write in the current iteration is consumed in the next iteration. This prevents the tool from pipelining this loop. To cope with this limitation, HLS tool vendors provide additional user directives (in the form of **pragmas**). These directives allow users to bypass the tool dependency analysis and force the compiler to generate a pipelined schedule. For example, asserting that the minimum number of iterations between dependent memory operations is 5 allows the HLS tool to overlap five iterations of the loop. This enables the tool to pipeline the loop.

```
    #pragma pipeline, max_latency=5
L1: for (int i=1; i<N; i++) {
      // reuse distance >=5 when N>1
      z[i*N+2] = foobar(z[i-1]);
    }
L2: for (int i=0; i<N; i++) {
      addr = lookup[i];
      x[addr] = foo(x[addr]);
    }
```

Figure 1: Two types of dependences.

However, in many cases it is impossible to determine dependencies at compile-time. Such cases typically occur when data-dependent array accesses are involved in the loop body. This is the case of the loop `L2` in Figure 1. For such loops, the only solution to extract additional parallelism is to perform a dependency analysis at runtime, by using so-called *runtime memory disambiguation.*

2.1 Related Work

In this section, we discuss an earlier work that employed runtime memory disambiguation in the context of HLS. Interested readers are referred to a more detailed discussion of related work in Appendix C. Ravi et al. [6] proposed the use of runtime memory disambiguation to obtain higher throughput implementations for applications containing memory accesses that cannot be disambiguated at compile time. The basic idea behind their technique is to generate two different schedules and to choose one of them at runtime. One schedule assumes that there are no dependencies between memory operations and hence provides more opportunities to exploit parallelism. The other schedule is conservative and assumes there is a dependency. Verification operations are introduced to check for a dependency at runtime. Depending on the outcome of these operations, appropriate schedule is chosen at runtime. The verification operations check for a dependency violation between the current iteration and the previous iteration. This allows only two loop iterations to overlap. We extend this idea by allowing multiple iterations to overlap to achieve a higher throughput. When multiple iterations overlap, it is necessary to add multiple runtime checks, one for each pending memory operation in the pipeline. We discuss this in detail in section 3. Further, the earlier technique adds new states to the FSM to incorporate different schedules and verification operations. This could potentially lead to complex FSM, which may affect the clock frequency. We borrow the technique employed in superscalar processor by introducing necessary control to stall the pipeline. Though our technique also affects clock frequency, we expect it to have a smaller impact. In the following, we discuss two issues that affect efficiency of our approach.

2.2 Customizing Disambiguation Hardware

In the context of HLS, implementing runtime memory disambiguation has much fewer drawbacks than in a complex super-scalar processor. In HLS, the architecture is customized for a given application kernel, hence it is possible to analytically measure the benefit of dynamic disambiguation for this kernel (or part if it) before deciding to use it.

For example, if the HLS compiler front-end fails to analyze a memory dependence and if profiling pass shows that the probability for an actual manifestation of the dependency during the loop execution is low, using this technique is likely to be beneficial. The main challenge is to customize the memory disambiguation hardware to have “good enough” accuracy, while minimizing its hardware footprint and critical path length.

2.3 Working at the Kernel C Source Level

Our approach should ideally be implemented in a commercial HLS tools. However, these tools are closed source compiler infrastructures and cannot be modified/extended by third parties. Although it would be possible to use an open-source HLS tool, it turns out that they lack the level of robustness that we expect to be able to make our technique work properly.

One solution is to implement this approach as a source-to-source transformation operating directly at the C level. Source-to-source (S2S) compilers are well known in the parallel computing community [7, 8], but are rarely used in the context of HLS [9, 10].

In a S2S-HLS flow, the HLS tool is used as a back-end in charge of low-level scheduling/binding stages. This permits the S2S compiler to be vendor independent, and eases the evaluation of the approach on several different HLS back-ends. It however brings additional challenges, we cannot have access or control the low-level scheduling/binding decisions performed at the back-end level.

3. PROPOSED APPROACH

We propose to transform the loop structure in the initial C specification to include additional control logic for memory dependency hazard detection. This control logic is in charge of stalling the pipelined schedule when necessary. *Note that our technique does not pipeline the loop; it transforms the loop to allow pipelining of the loop.* This section provides a description of our technique and its implementation as an automatic source-to-source transformation.

3.1 An Illustrative Example

We first illustrate our approach using a simple example shown in Figure 2. In this kernel, which computes an image histogram, the array `hist[]` holds the frequency of occurrence of pixel intensity values in `pixel[][]`. Array `hist[]` is indexed by the value of the pixel at hand and its corresponding occurrence count is incremented.

```
uint i,j;
uchar val;
uchar pixel[W][H];
uint hist[256];
// histogram building loop
for (i=0; i<W; i++) {
  for (j=0; j<H; j++) {
   val = pixel[i][j];
   hist[val] = hist[val]+1;
  }
}
```

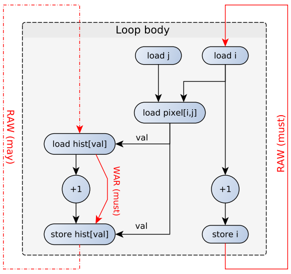

Figure 2: Histogram kernel and dependency graph

The innermost loop body contains a potential RAW (Read After Write) loop carried dependency over array `hist[]`. Since the access pattern of array `hist[]` depends on the data (`pixel[i][j]`) that is being processed, static dependency analysis will fail at disambiguating memory references for `hist[]`. The loop pipeline scheduler will hence have to consider this loop carried dependency when looking for a pipelined schedule. Assuming one cycle latency for the array read operation and one cycle latency for the addition, we will obtain a lower bound for $II = 3$ as illustrated in Figure 3.

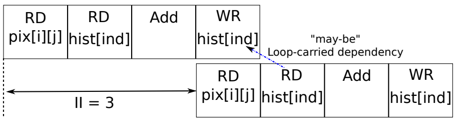

Figure 3: Original pipeline of the histogram kernel

However, since successive pixels within an image have a relatively low probability of having *exactly* the same value, this dependency will not manifest itself very often, making the statically pipelined schedule inefficient. The use of runtime disambiguation techniques brings the possibility of implementing a tighter ($II = 1 + \epsilon$) pipeline schedule, at the price of additional runtime checks to ensure the schedule correctness. These runtime checks perform runtime address disambiguation and consists checking if two pending memory reference addresses alias.

In the context of loop pipelining, there can be multiple instances of the same instruction in-flight in the pipeline. It is therefore necessary to check for a potential dependency violation for each of the pending memory operations.

We use a shift register to store the addresses accessed by write operations, on which other read operations could be dependent. In the following, we refer to such set of write operations as *dynamic writes*[2], and similarly all read operations that may alias as *dynamic reads*. Note that the memory operations that can be statically analyzed and disambiguated are not considered as *dynamic read/write*. All dynamic reads are checked against the addresses stored in the shift register to detect any aliases, i.e. dependence violations at runtime. The entire loop body, including the increment of the loop index, is guarded by the result of this check. If we detect an alias at runtime, no new computation is issued (this can be seen as dynamically introducing a *bubble* in the pipeline). In this approach, a distinct shift register is needed for every dynamic write instruction. Similarly, alias detection hardware is needed for related pairs of *dynamic reads/writes* (This issue is further discussed in Section 3.5).

To help the reader understand our approach, we show in Figure 4 the transformed code corresponding to the histogram kernel. An array `shift_reg[]` is introduced to store the addresses of pending dynamic write operations in the pipeline. Alias detection is performed by the loop labeled L2, which we completely unroll (using a `#pragma`) to enable parallel execution of the address comparisons (this is only possible if the `shift_reg[]` is mapped to registers). Both the depth of the shift register and the iteration count of the loop depend on the latency of the pipeline.

```
   #pragma II=1, ignore_dependency hist
   for(i=0; i<H; i++) {
L1:  while (j<W) {
       val = pixel[i][j];
       // dependency violation detection logic
       #pragma UNROLL
L2:    for(k=LATENCY-1, stall=0; k>0; k--) {
         stall = stall | (shift_reg[k-1] == val);
         shift_reg[k] = shift_reg[k-1];
       }
       shift_reg[0] = stall?(-1):val;
       if (!stall ) {
         hist[val] = hist[val] + 1;
         j = j + 1;
       }
     }
   }
 }
```

Figure 4: Transformed code for histogram kernel.

The results of all the address comparisons are *or*ed to generate a `stall` signal. The value of this signal indicates if an immediate execution of the current loop iteration induces a memory dependency violation. If there is a violation, no new operation is issued until the pending conflicting memory operation has been completed. This is ensured by guarding all the instructions in the loop body using the `stall` signal as predicate. It is important to note that the loop increment is also guarded by this condition.

In this transformed loop, since the stall logic detects any possible violation and stalls the execution, the *while* loop (labeled L1) can now be safely pipelined, by forcing the HLS tool to ignore the target memory dependency. An illustration of the pipelined execution is provided in Figure 5. In the following sections, we discuss various aspects of this technique in detail.

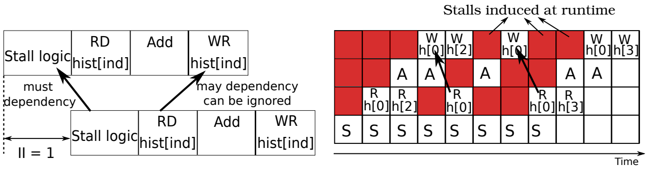

Figure 5: Pipeline of the transformed histogram kernel.

3.2 Identifying Ambiguous Memory Accesses

The first step in transforming the loop is to identify memory accesses that need to be handled by this approach. In

[2] Note that though we say writes, these operations could be memory reads in the case of WAR dependencies as explained later in Section 3.3.

this work, we restrict the use of the technique to arrays mapped to on-chip memory banks, where no inter-array alias is allowed (inter-array alias is usually not supported by HLS tools).

The detailed algorithm to identify ambiguous memory accesses pairs is provided in Appendix A (algorithm 1). This algorithm analyzes RAW, WAR and WAW dependencies. In the case of a RAW or WAW dependency we check that a memory read (resp. memory write) does not alias a pending write operation (we hence need to keep track of write addresses). In the case of WAR dependencies, we check issuing writes against pending reads (hence keeping track of the read address).

However, it is to be noted that not all dependencies require dynamic memory disambiguation. In the following, we explain which dependencies are relevant for our approach.

1. Fully-static dependences consist of memory accesses that can be statically analyzed and the compiler is able to prove the absence of alias. Obviously, these need not be considered by our approach.

2. Partially-static dependences consist of accesses that can be statically analyzed but for which the compiler identifies a subset of the loop iteration domain where alias happens (see Appendix C.1). Some of these references may be good candidates for our technique which could be more area efficient than fully static solutions [10].

3. Fully-dynamic dependences consist of all pairs of memory accesses that cannot be statically analyzed and for which designer cannot assert that alias will never occur. An example of such a dependency is shown in our earlier example in Figure 2.

3.3 Extracting Address Expressions

Once we have computed potentially aliasing memory accesses, we must identify their index/address expressions. In the case of multi-dimensional arrays, index expressions for a given dimension are ignored for disambiguation when they are found to be the same in both elements of the pair.

Once identified, the set of index expressions that require disambiguation must be *sliced*[3] out of the loop body where these indices are originally computed. An example of a program and the *slice* corresponding to the computation of the index `val` is shown in Figure 6.

```
for ( i = 2; i < n; i++ ) {
   inc = 10;
   if ( i == X[i] ) {
      val = Y[i];
      inc = inc + 30;
   } else {
      val = Y[i] + 10;
   }
   C[i] = foo(C[val]) + inc;
}
```

```
int slice(...) {
  if ( i == X[i] )
   val = Y[i];
  else
   val = Y[i] + 10;
  return val;
}
```

Figure 6: Example showing the program slice of index expression

This step is very important as this *slice* will be moved out of the initial body and will become part of the pipeline stalling control logic. Note that it is possible only if the execution of the slice is side-effect free and if the slice does not involve ambiguous memory accesses (these limitations are discussed in Appendix D).

3.4 Simplifying Disambiguation Logic

Since we are deriving a custom micro-architecture tailored to a specific kernel, we also have the opportunity to customize the disambiguation logic to minimize its hardware footprint. Instead of storing a complete pending address in the shift register it is possible to use a hash of its value, to store (and compare) a smaller number of bits. This kind of technique is used in super-scalar processors to reduce the hardware cost of the disambiguation engine [13, 14].

This saving in area comes at a cost: using fewer bits to encode the address will lead to false positives in the alias detection, leading to increases in number of (unnecessary) stall cycles.

However, because we are designing application specific hardware, we can tailor the choice of the hashing function to the kernel at hand (such a customization can be driven by memory access execution traces of the kernel). One can for example search for the sub-set of bits in the address that, when used as hash, minimize the numbers of conflicts[4]. More sophisticated hashing functions used for super-scalar processors and based on Bloom filters [13] or non-singular binary matrix multiplication [15] could also be considered. However, their larger hardware footprint makes them unpractical for our approach.

Depending on the hashing function employed, not all bits in the address are required to check if the pipeline has to be stalled. This in turn can shorten the combinatorial path of alias detection logic, as we may start detecting a possible alias even before all address bits have been computed.

3.5 Reducing Shift Register Area Cost

As mentioned earlier, we introduce a shift register to hold addresses (either for a read or for a write) for all in-flight memory operations involved in an ambiguous dependency pair. One obvious simplification is to make sure a same address sequence is not mapped to two different shift registers.

It is also possible to take advantage of memory operations occurring in mutually exclusive execution paths inside the loop body. In this case, the same shift-register and alias detection logic can be used for both memory operations. Finding the minimum number of such registers is a classical resource conflict allocation problem that can be modeled (and solved) as a graph coloring problem, where non-mutually exclusive registers are considered as conflicting nodes in the graph. Our current implementation however uses a simple greedy strategy detailed in Algorithm 2.

In addition to the number of shift register components, we also need to determine their individual depths. These depths depend on the number of stages separating ambiguous access pairs in the loop pipeline. In our running example in Figure 2, the number of entries in the shift register depends on the schedule of the write operation in the pipeline and its corresponding read. In this example, the write is scheduled at stage 3, while the read is scheduled at stage 1. This hence requires a two-entry shift register.

Since we are implementing this technique as a front-end optimization, such a detailed knowledge about the pipelined schedule may not be available. In our current implementation, we assume the worst-case latency, which is the latency of the pipeline itself. We assume that this information is provided by the user through a compiler directive. This could be improved and automated as most HLS tools provide a script based interface to access cycle accurate schedule in-

[3] Program slicing [11, 12] is a well studied program transformation that we don't detail here.

[4] Our work on automating the selection of a hashing function is still on-going.

formation.

4. IMPLEMENTATION & RESULTS

In this section, we first briefly describe the implementation of our technique in an existing compiler infrastructure and then present some experimental results

4.1 Implementation in an S2S Compiler

The technique discussed in this paper was implemented as a part of GeCoS[5], a source-to-source compiler framework targeted at High-Level-Synthesis. To implement this transformation we leverage the compiler high-level intermediate representation that offers an SSA based representation, while preserving the program structure. Our SSA model is able to deal with both scalars and array accesses (using *may/must* dependency information) as we assume no inter-array alias and do not support pointers.

For these experiments, we consider that all the memory read/write pairs that do not alias have been flagged either by the compiler or by user directives. The remaining ambiguous access pairs are then considered for runtime dependency analysis. In our current prototype, the choice of the hashing function is limited to simple function where the hash is a concatenation of the n least significant bits from index expression of each dimension of the array.

4.2 Chosen Benchmark

There are currently very few publicly available benchmark suites for HLS tools. The CHStone[16] seems the only one containing kernels with loops and array accesses. However it only contains examples that are well supported by most HLS tools (regular access patterns, no complex loop carried dependencies, etc.). It is therefore irrelevant in our case.

To validate our approach we hence chose a set of kernels with memory dependencies for which our approach can be applied. The first set of kernels exposes *partially-static* memory dependencies. They are the same as those used by Morvan et al. [10] in their work and correspond to loop nest that underwent a loop coalescing transformation. This set includes `BBFIR`, `Matrix-mult`, `Floyd-Warshall` and `Jacobi-2D` kernels. For the sake of completeness, we compared our results with their approach (which provides a compile-time schedule correction mechanism). This comparison is discussed in Table 1 of Appendix B.

The second set of kernels contains *fully-dynamic* dependencies. It consists of `Knapsack`, `histogram` and `tree-traversal` kernels. For these loops, performance improvements provided by our technique are data-set dependent. Rather than trying to characterize realistic workloads (which is inherently difficult since these kernels can be used in a wide variety of applications), we provide performance improvement by using synthetic benchmarks with two different probabilities (10% and 50% chances) for a memory dependency to manifest during the kernel execution.

4.3 Experimental Results

We used two leading edge commercial High-Level Synthesis tools, which we are not allowed to name due to licensing issues. The first one, that we will name LEC-HLS, is considered as being one of the most efficient and robust HLS tool currently on the market. The second one is targeted for a FPGA technology vendor X and is now sold as a standard component of their FPGA design flow. We will call this second tool FPGA-HLS.

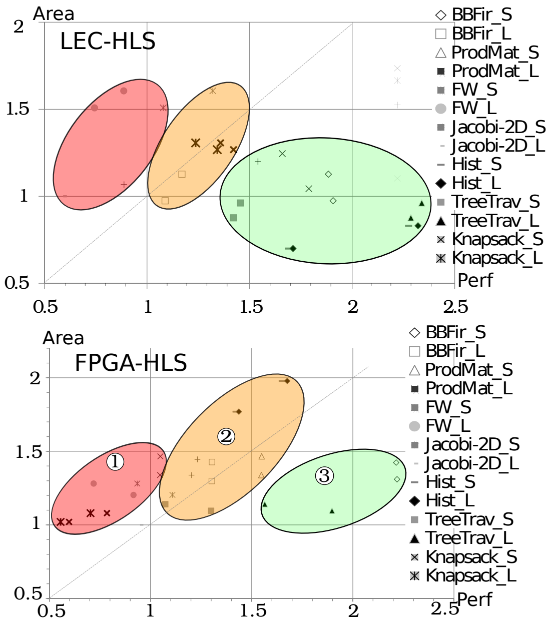

Figure 7: Area/performance trade-off for LEC-HLS and FPGA-HLS

Each kernel was transformed into several versions, each one with a different hashing strategy. The goal is to explore the trade-off between the alias detection accuracy (minimizing false positive dependencies) and area and clock speed overhead of the detection logic. We synthesized both the original and transformed kernels for Virtex-6 and Stratix-IV FPGAs and compared their area, frequency, clock cycles count (#cycles) and overall performance (combining both frequency and #cycles). Due to page limit, the complete results are provided in Appendix. We also discuss the impact of increasing pipeline depth on area and performance in Appendix B.

We provide a graphical summary of the results in Figure 7. We normalized the area and execution time of each transformed kernel by scaling them w.r.t the area and execution of their corresponding original kernel. For few kernels, inner-most loop did not contain any dependencies and hence it was possible to pipeline the loop without employing our technique. For such kernels, the comparison shown in the figure is with that of the pipelined version. We refer to this implementation as pipeline-1D. For such kernels, our technique is applied on the coalesced loop [10], which allows pipelining of outer levels of the loopnest in addition to the inner-most loop.

Performance is shown for two versions of each benchmark kernel (the L-suffix stands for large iteration count, the S suffix stands for small iteration count). Area is provided for full address alias detection (e.g. no hashing) and for a hashing based on the 4 least significant bits of the address. On average, we observe an increase of 18% overhead in area and a reduction of 74% in execution time for LEC-HSL tool and 29% and 25% for FPGA-HLS tool over pipeline-1D implementation.

To quantify the benefit of our approach, we compared our results against a theoretic/hypothetical optimization which, given an extra area of x% would decrease execution time by the same amount. This reference is represented in Figure 7

[5] http://gecos.gforge.inria.fr/doku.php

as dashed diagonal lines. Additionally we represent in the figure three regions named ①,② and ③, which correspond to different cases of performance/area trade-offs.

- Region ① contains all transformed kernels for which our optimization is counter-productive. Those include (in both HLS tools) `Floyd-Warshall` and `Jacobi-2D` (for large iteration counts). They correspond to situations where the number of false dependence eliminated using our technique is limited and where the degradation in clock frequency due to the alias detection logic is high.

- Region ② contains all transformed kernels where our optimization helps improving overall performance, but for a considerable area overhead. These include `Matrix-mult` and `knapsack` kernels. In both these kernels the inner-loop is parallel and can be pipelined without using our technique. Hence, the scope for performance improvement is limited.

- Region ③ contains all transformed kernels for which our optimization significantly improves the efficiency of the accelerator at a minor area overhead. These include `tree-traversal` and `BBFIR` in both tools. In LEC-HLS tool, `histogram` also belongs to this region. In FPGA-HLS tool, though we could achieve performance improvement for `histogram` kernel, the area overhead is also significant. So this belongs to Region ② in FPGA-HLS tool. area overhead.

Interestingly, kernels in regions ② and ③ are different from one tool to another. However, it seems that the LEC-HLS better supports our technique. Nevertheless the results show that the approach is beneficial to kernels where data-dependent memory accesses prevent pipelining (for ex: `tree traversal`, `histogram`). It also performs relatively well in less favorable cases where inner loop can be pipelined without our technique (for ex: `BBFIR`, `knapsack`).

5. CONCLUSION

In this paper, we have proposed an original technique for improving the ability of current HLS to deal with loops involving dynamic memory dependencies. Our approach was implemented in a source-to-source compiler and shows significant performance improvements.

We consider our work as the first step toward more aggressive dynamic scheduling technique which could borrow from approaches used in high-performance processors. We believe that there exist many interesting challenges that could benefit from advanced source-to-source transformations mixing aggressive static analysis features with customized speculative micro-architectures.

Acknowledgments

This work was funded in part by French ANR Compa and INRIA-STMicroelectronics Nano2012-S2SHLS projects.

6. REFERENCES

[1] M. Graphics, "Catapult-C Synthesis." http://www.mentor.com.

[2] Xilinx corp., *Xilinx Vivado Design Suite User Guide : High-Level Synthesis*, ug902 (v2012.2) ed., 2012.

[3] B. Betkaoui, D. Thomas, W. Luk, and N. Przulj, "A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study," in *International Conference on Field-Programmable Technology*, pp. 1 –8, December 2011.

[4] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse LU Decomposition using FPGA," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing*, 2008.

[5] M. S. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *Proceedings of the ACM SIGPLAN conference on Programming Language design and Implementation*, PLDI '88, pp. 318–328, 1988.

[6] S. Ravi, G. Lakshminarayana, and N. K. Jha, "Removal of Memory Access Bottlenecks for Scheduling Control-flow Intensive Behavioral Descriptions," in *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pp. 577–584, 1998.

[7] R. Keryell, C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, and P. Jouvelot, "PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers," Technical Report 289, CRI, École des mines de Paris, Apr. 1996.

[8] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[9] J. M. Cardoso, J. Teixeira, J. C. Alves, R. Nobre, P. C. Diniz, J. G. Coutinho, and W. Luk, "Specifying Compiler Strategies for FPGA-based Systems," *IEEE Symposium on Field-Programmable Custom Computing Machines*, vol. 0, pp. 192–199, 2012.

[10] A. Morvan, S. Derrien, and P. Quinton, "Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion," in *International Conference on Field-Programmable Technology*, pp. 1–10, IEEE, Dec. 2011.

[11] M. Weiser, "Program Slicing," in *Proceedings of International Conference on Software engineering*, ICSE '81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.

[12] F. Tip, "A Survey of Program Slicing Techniques.," technical report, CWI, The Netherlands, 1994.

[13] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 399–, IEEE, 2003.

[14] L. Baugh and C. Zilles, "Decomposing the Load-store queue by Function for Power Reduction and Scalability," *IBM Journal on Reearch and Development*, vol. 50, pp. 287–297, Mar. 2006.

[15] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic Memory Disambiguation using the Memory Conflict Buffer," *ACM SIGOPS Operating Systems Review*, vol. 28, pp. 183–193, Dec. 1994.

[16] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis.," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

APPENDIX

A. DETAILED ALGORITHMS

In this section, we provide formal specification of the algorithms. Algorithm 1 is used to identify ambiguous memory operations. The input for this algorithm is a dependency graph that includes anti, output and true dependencies.

Algorithm 1 Identify the dependent memory operations

```
Require: $DG$ dependency graph that includes anti and output
  dependencies, $\Delta$ the latency of the pipeline
  procedure IdentifyDependentPairs($DG, \Delta$)
      for all $d \in DG$ do
          if $depDist(d) \geq \Delta$ over entire iteration space then
              continue;
          end if
          if $d$ holds over whole iteration space then
              continue;
          end if
          $src \leftarrow source(d)$
          $tgt \leftarrow target(d)$
          $depPairs \leftarrow depPairs \cup (src, tgt)$
      end for
  end procedure
```

Another optimization we briefly mentioned in section 3.5 is to reuse shift registers across memory operations that are on mutually exclusive paths. To motivate the need for such an optimization, consider a case shown in Figure 8.

```
for ( int i = 0; i < N; i++ ) {
    for ( int j = 0; j < W; j++ ) {
        if ( j > w[i] ) {
lab1:       T[i][j] = f(T[i][a[i]);
            output[i] = f(T[i][w[i]);
        } else {
lab2:       T[i][b[i]] = g(T[i-1][b[i]]);
        }
lab3:   output[i] = k(T[i][j]);
    }
}
```

Figure 8: An example kernel

Using a simple scheme, we would require three shift registers, one per each memory operation. However, shift registers can be shared between two write operations (at labels `lab1` and `lab2`) since they are on mutually exclusive paths. Algorithm 2 describes the method to assign shift registers to memory operations.

B. DETAILED RESULTS

In this section, we summarize the experimental results. Table 1 summarizes the hardware area (in terms of LUTs, Flipflops and DSPs) and the frequency of each of the implementations obtained using LEC-HLS tool. Table 5 summarizes the area and frequency of the implementations obtained using FPGA-HLS tool. The performance comparison of these implementations is summarized in tables Table 3 and Table 4 for LEC-HLS and FPGA-HLS tools respectively. We compare the results obtained using LEC-HLS tool with the results of approach proposed by Morvan et al.[MDQ11].

Another interesting aspect to study is the impact of pipeline depth on the efficiency of the our technique. Increasing pipeline depth leads to an increase in the number of entries in the shift register and the comparison logic. The secondary effect on area is due to the increase in number of bits used in the hash function. As the pipeline depth increases, the number of false positives would also increase.

Algorithm 2 Assigning shift registers for dependent memory operations

```
  procedure AssignShiftReg($depPairs, latency$)                    ▷
  If latency is not available for all operations, we assume a worst
  case latency of $\Delta$
      $regMap \leftarrow$ MergeDepPairs(depPairs);
      for all $reg \in regMap$ do
          $maxLat \leftarrow$ maxLatency(regMap[reg], latency)
          $numEntries[reg] \leftarrow maxLat$
      end for
  end procedure
  procedure MergeDepPairs($depPairs$)
      $workingSet \leftarrow depPairs$
      while $workingSet \neq \emptyset$ do
          $mergedSet \leftarrow \emptyset$
          for all $pair \in workingSet$ do
              $tgt \leftarrow target(pair)$
              if $tgt$ is mutually exclusive to $mergedSet$ then
                  $mergedSet \leftarrow mergedSet \cup pair$
              end if
          end for
          $tgtOps \leftarrow tgtOps - mergedSet$
          $regMap[reg_i] \leftarrow mergedSet$
          $i \leftarrow i + 1$
      end while
      $returnregMap$
  end procedure
  procedure maxLatency($regPairs, latency$)
      $maxLat \leftarrow -1$
      for all $pair \in regPairs$ do
          $src \leftarrow source(pair)$
          if $latency[src] \geq maxLat$ then
              $maxLat \leftarrow latency[src]$
          end if
      end for
      $return(maxLat)$
  end procedure
```

Application	Version	Hardware characteristics			
		ALUT	REG	DSP	Freq (MHz)
BBFIR	original	553	152	4	185
	[MDQ11]	649	241	4	241
	FullAddr	534	175	4	173
	4bit Hash	469	144	4	175
ProdMat	original	489	215	4	272
	[MDQ11]	559	226	4	231
	FA-16bit	499	230	4	230
	8-bit	466	215	4	225
	4bit Hash	295	276	4	242
FloydWarshal	original	383	74	0	271
	[MDQ11]	859	87	0	210
	FA-16bit	569	206	0	160
	8-bit	504	172	0	173
	4bit Hash	475	160	0	173
Jacobi-2D	original	1012	845	8	164
	[MDQ11]	1417	975	8	172
	FA-16bit	1143	825	8	143
	8-bit	1065	827	8	150
	4bit Hash	929	754	8	143
Knapsack	original	372	84	0	282
	FA-8bit	429	167	0	336
	4bit Hash	423	155		370
Histogram	original	159	71	1	372
	FA-8bit	126	65	1	318
	4bit Hash	112	49	1	320
TreeTraversal	original	370	123	0	295
	FA-8bit	307	167	0	227
	4bit Hash	273	159	0	222

Table 1: Hardware characteristics of different implementations using LEC-HLS tool

In order to maintain the same rate of false positives, we have to increase the number of bits used for hashing the address of memory operations. The number of additional bits required depends on the actual data pattern. Further, increase in the number of pipeline stages also increases the penalty we pay in the case of an actual dependency. Table 2 shows the results we obtained for an example kernel by increasing the number of stages from 5 to 22. The example contains a series of 32 bit multiplications guarded by conditions based on the output of previous multiplication. We obtain implementations with various pipeline stages by controlling the required clock frequency. To measure the impact on performance, we assume uniform probability for a memory dependence to manifest at runtime. The table shows the incremental performance/area overhead over the implementation with fewer stages.

Pipeline Depth	HashSize (in bits)	ALUT	REG	Freq (in MHz)	% inc Area	% inc perf.
5	8	746	193	31.3	-	-
8	10	913	345	57	33.9	20.88
13	12	1073	604	99.1	33.3	7.3
22	13	1074	1113	148.6	30.4	2.49

Table 2: Impact of pipeline stages on Area and Performance

Application	Version	Hardware characteristics			
		ALUT	REG	DSP	Freq (MHz)
BBFIR	original	216	290	3	185.6
	FA-8bit	383	339	3	186
	4bit Hash	346	311	3	186
Prodmat	original	231	304	3	185.7
	FA-16bit	422	444	3	186
	8bit Hash	396	388	3	186
	4bit Hash	356	360	3	185.8
FloydWarshall	original	233	155	0	311.5
	FA-16bit	428	214	0	208.5
	8bit Hash	331	166	0	214.1
	4bit Hash	307	160	0	275.9
Jacobi-2D	original	385	307	4	182
	FA-16bit	892	517	4	162
	8bit Hash	578	422	4	165.2
	4bit Hash	522	404	4	167.5
Knapsack	original	340	201	0	342.9
	FA-8bit	378	206	0	235.9
	4bit Hash	367	185	0	188.9
Histogram	original	62	29	1	528.5
	FA-8bit	107	73	1	325.6
	4bit Hash	104	57	1	380
TreeTraversal	original	188	83	0	411
	FA-8bit	192	117	0	264.9
	4bit Hash	192	105	0	321.2

Table 5: Hardware characteristics of different implementations using FPGA-HLS tool

C. DISCUSSION ON RELATED WORK

Runtime dependence analysis (and also dependency analysis) is a widely studied topic and is studied in various contexts. In the following, we summarize some of the important research work.

C.1 Compiler Based Dependency Analysis

Dependency information is important for parallelizing compilers. Hence, static dependency analysis[6] has been (and is still) a very widely studied topic. Earlier works have focused on regular (i.e. affine) access patterns such as those found in scientific codes. The developed approaches range from fast but very conservative, to exact ones [Fea91, Pug91]. More recent works focus on extending the scope of these analyses to more irregular computation patterns [CBF95, OR12].

Dependency information obtained from such static analysis is used by compilers when performing optimizations. In the context of loop pipelining, the dependency information is an important factor that determines the efficiency of the pipeline schedule. In many cases, a loop-carried dependency will hold only over a sub-set of the iteration space and sophisticated analysis can compute this information. This information can be used to perform an index set splitting transformation, which isolates the iteration subset that can be pipelined (or parallelized) [GFL00]. Another approach, specifically targeted at HLS corrects the loop schedule by inserting wait-states at compile time. This approach was proposed by Morvan et al. [MDQ11] for correcting dependency violations occurring after a loop coalescing transformation.

C.2 Software Runtime Dependency Analysis

The idea of performing software based runtime dependency analysis was proposed as early as 1989 by Nicolau [Nic89]. The basic technique is to introduce checks in the source code to disambiguate memory references at runtime. This allows compiler to aggressively schedule operations assuming memory references do not alias on one path. Huang et al. [HH94] present a similar technique for architectures that support conditional execution. They employ predicates to guard the statements instead of explicit branches.

Salami et al. [SCAV02] extend this technique to disambiguate an entire loop instead of disambiguating at the level of iterations. This was proposed in the context of multimedia applications. Another work by Rus et al. [RRH03] uses a representation called *RT_LMAD*, *Run-Time Linear Memory Access Descriptor* to summarize the memory references in the program. This representation is used to disambiguate memory references at the level of loops efficiently. Both of these approaches aim at detecting loops containing *DOALL* parallelism.

C.3 Hardware Runtime Dependency Analysis

The idea of performing memory dependence analysis at the micro-architectural level dates back to early 80s. Smith [Smi84] proposed Decoupled Access Execute architectures to perform "an associative compare of each newly issued load address with all the addresses in the Write Address Queue" to ensure *store forwarding* for aliasing load/store pairs.

This kind of mechanism was later extensively used in Load-Store Queues (LSQ) of wide issue out-of-order superscalar processor architectures to handle dependence violations due to speculative execution of load/store operations. Such super-scalar processors have very deep execution pipelines (31 stages for the Pentium D). This mechanism is therefore quite costly in terms of transistor count. For such architectures, the challenge is to provide a scalable mechanism to deal with hundreds of in flight instructions [BZ06, SDB$^+$03]. In the context of embedded hardware platforms, the energy consumed by such an approach also needs to be taken into account. In most of the embedded platforms even when the processor pipeline depth and issue width remain limited, the performance improvement provided by this mechanism rarely outweighs its area and

[6] The book of Allen and Kennedy [KA02] offers a good survey of the topic.

Application	Size	orig	[MDQ11]	FA-16bit	8bitHash	4bitHash
BBFIR	256 × 8	22413.3	9279	-	11895.24	11751.18
	1024 × 32	221790.66	148050	-	189456.84	203727.09
ProdMat	4^3	1182.72	294	295.1	301.92	280.16
	128^3	15457812.48	9078597	9101657	9311372.64	8640282.72
FloydWarshall	16^3	31454.08	22816	25750	23813.6	29131.2
	128^3	10760494.08	11650905	13107350	12132844.24	14453814.8
Jacobi-2D	30 × 16	91816.2	44000	62539.5	59587.02	103431.03
	30 × 256	14382664.2	11262604	13751755.5	13264369.02	24248289.03
Knapsack	128 × 16	1034.48	-	-	454.672	612.5
	1024 × 64	8238.32	-	-	3549.456	4812.5
Histogram	128	8223.54	-	-	5646.96	5782.5
	1024	231898.42	-	-	99143.44	101425.5
TreeTraversal	128 × 4	9065.94	-	-	6670.62	6375.24
	1024 × 16	246500.82	-	-	199221.66	183598.92

Table 3: Execution time (in ns) for different implementations synthesized using LEC-HLS tool

Application	Size	orig	FA-16bit	8bitHash	4bitHash
BBFIR	256 × 8	24914.1	-	11162.6	11156.4
	1024 × 32	230067.6	-	176408.6	176552.0
ProdMat	4^3	882.8	569.6	569.6	570.2
	128^3	11818828.6	11270127.0	11270127.092	11282710.04
FloydWarshall	16^3	17976	19749.9	19231.06	16217.4
	128^3	7046592	10058046.5	9800961.6	7692168.3
Jacobi-2D	30 × 16	63828.9	52830.9	51795.5	53055.3
	30 × 256	11255393.7	12143992.5	11907988.2	11852163.3
Knapsack	128 × 16	728.4	-	441.6	512.6
	1024 × 64	5814.1	-	3468.3	4046.0
Histogram	128	4673.7	-	4354.0	3595.5
	1024	127063.4	-	81187.4	66969.9
TreeTraversal	128 × 4	7467.8	-	9512.3	12489.6
	1024 × 16	203049.8	-	284335.1	359979.2

Table 4: Execution time (in ns) for different implementations synthesized using FPGA-HLS tool

energy overheads.

Interestingly, some mixed static/dynamic approaches have also been proposed. In the work of Gallagher et al. [GCM+94] the hardware support for runtime dependency analysis is only activated for load/store pairs that are flagged by the compiler as being possibly dependent and use compiler generated repair code in case of a dependency violation. This technique was later extended by Mahadevan et al. [MNJH00] in the context of modulo scheduled loops targeting the EPIC IA64 architecture.

C.4 Runtime Scheduling in HLS

High-Level Synthesis frameworks focus on kernels that exhibit limited data-dependent behavior[7] inside loop kernels. For this reason, most of HLS tools have been relying on static scheduling techniques.

Several research work [KW02, RB94, GSK+01, VCG05] have addressed issues related to runtime scheduling (in which we include speculative techniques) in order to support efficient scheduling (in both latency and area) over possibly complex control-flow execution paths. However these contributions do not employ runtime dependency analysis techniques and assume that memory disambiguation is performed at compile time (hence conservatively).

More recently, Thielman et al. [THK11] studied the automatic generation of speculative application specific microarchitecture from high-level specifications. While similar in its goal, their approach differs from ours in two ways: (1) They focus on value prediction techniques to reduce impact of external memory accesses. However, they do not consider runtime hardware memory disambiguation techniques. (2) They focus on a specific architectural model (the PreCORE machine); whereas our approach aims at a seamless integration into existing HLS flows.

D. LIMITATIONS OF THE APPROACH

In this section, we discuss the limitation of the approach both from the point of efficiency and from the applicability of the technique.

D.1 Performance Bottleneck

It is important to understand how our stall logic mechanism is implemented to realize its limitations.

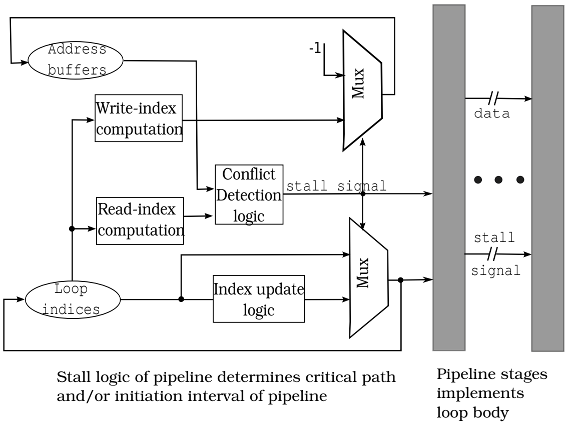

Figure 9: Block diagram to illustrate the technique

Figure 9 shows the organization of our pipeline stall logic. It comprises two critical loops that limit the initiation interval of the pipeline. One dependency is through the increment of the loop index variable and the second is the update to the shift register. Both these operations are conditional and are control dependent on the `stall` signal. Thus, the

[7] When it does, the data dependent behavior does not impact parallelization opportunities.

computation of `stall` signal is on the critical path. Further, update to the shift register requires computation of index expressions of memory operations that are being analyzed by our technique. This computation may become critical depending on the complexity of the index computation.

For example, consider a hypothetical example shown in Figure 10. To obtain accurate information about the write accesses to array `T` we need to evaluate the condition `j > w[i] * w[i]`. Evaluating such complex conditions may stretch the initiation interval and/or increase the clock width thereby degrading the performance obtained.

```
for ( int i = 0; i < N; i++ ) {
    for ( int j = 0; j < W; j++ ) {
        if ( j > (w[i] * w[i]) ) {
            T[i][j] = max(T[i-1][j],T[i-1][j-w[i]] + v[i
                ])
        } else {
            T[i][j-1] = T[i-1][j];
        }
    }
}
```

Figure 10: A hypothetical example to illustrate the critical path of our approach

One way to work around this problem is to compromise on accuracy and record both writes without evaluating the condition. Note that, in such cases, we need to provide two different shift registers, one for each write operation. In such cases, the optimization discussed in Section 3.5 (of merging shift registers on mutually exclusive paths) cannot be performed. Another way to improve performance is to use speculation to reduce the critical path by allowing the pipeline to proceed as long as there is no change to the global state. We are still looking at the details of this technique.

D.2 Scope of the Approach

Our approach cannot find a schedule with an $II = 1$ for applications that contain dynamic memory references in the program *slice* (refer section 3.3) of the index expression. To be able to pipeline such a loop we need to use speculation. To illustrate such a case, consider the example in Figure 11. In this example, the `index1`, which is used to access data from array `Y`, is based on a read to the same array `Y`. The reads and writes to the array `Y` are data dependent and cannot be statically disambiguated. Hence, we cannot read value from `Y` before computing the stall logic. However, we require a read to the array for computing the stall logic. This causes a circular dependency and can only be broken using speculation.

```
for ( int i = 0; i < N; i++ ) {
    index0 = X[i];
    index1 = foo(Y[index0]);
    Y[i] = bar(Y[index1]);
}
```

Figure 11: Chained ambiguous memory access pairs

E. References

[BZ06] L. Baugh and C. Zilles. Decomposing the Load-store queue by Function for Power Reduction and Scalability. *IBM Journal on Reearch and Development*, 50(2/3):287–297, March 2006.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy Array Dataflow Analysis. *SIGPLAN Notices*, 30(8):92–101, 1995.

[Fea91] Paul. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 1991.

[GCM+94] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wenmei W. Hwu. Dynamic Memory Disambiguation using the Memory Conflict Buffer. *ACM SIGOPS Operating Systems Review*, 28(5):183–193, December 1994.

[GFL00] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index Set Splitting. *International Journal of Parallel Programming*, 28(6):607–631, December 2000.

[GSK+01] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Speculation Techniques for High Level Synthesis of Control Intensive Designs. In *Proceedings of the 38th conference on Design automation*, pages 269–272. ACM Press, June 2001.

[HH94] AS Huang and S Httang. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. *SIGARCH Computer Architecture News*, pages 200–210, 1994.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[KW02] Apostolos A. Kountouris and Christophe Wolinski. Efficient Scheduling of Conditional Behaviors for High-level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 7(3):380–412, July 2002.

[MDQ11] Antoine Morvan, Steven Derrien, and Patrice Quinton. Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion. In *International Conference on Field-Programmable Technology*, pages 1–10. IEEE, December 2011.

[MNJH00] Uma Mahadevan, Kevin Nomura, Roy Dz-ching Ju, and Rick Hank. Applying Data Speculation in Modulo Scheduled Loops. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 169–. IEEE Computer Society, 2000.

[Nic89] Alexandru Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions On Computers*, 38(5):663–678, May 1989.

[OR12] Cosmin E. Oancea and Lawrence Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, volume 47, page 509. ACM Press, June 2012.

[Pug91] William Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*, pages 4–13. ACM Press, August 1991.

[RB94] Ivan Radivojevic and Forrest Brewer. Incorporating Speculative Execution in Exact Control-dependent Scheduling. In *Proceedings of the Design Automation Conference*, DAC '94, pages 479–484, 1994.

[RRH03] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *International Journal of Parallel Programming*, 31(4):251–283, August 2003.

[SCAV02] Esther Salamí, Jesús Corbal, Carlos Álvarez, and Mateo Valero. Cost Effective Memory Disambiguation for Multimedia Codes. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, pages 117–126, New York, NY, USA, 2002. ACM.

[SDB+03] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pages 399–. IEEE, 2003.

[Smi84] James E. Smith. Decoupled access/execute Computer Architectures. *ACM Transaction on Computer Systems*, 2(4):289–308, November 1984.

[THK11] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. Precore-A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation. *International conference on Field Programmable Logic*, pages 123–129, September 2011.

[VCG05] Girish Venkataramani, Tiberiu Chelcea, and Seth Copen SC Goldstein. HLS Support for Unconstrained Memory Accesses. In *IEEE International Workshop on Logic Synthesis (IWLS)*, Lake Arrowhead, CA, 2005.