Area-Efficient Memory Scheduling for Dynamically Scheduled High-Level Synthesis

Xuefei He, Jianyi Cheng and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, UK
Email: {xuefei.he18, jianyi.cheng17, g.constantinides}@imperial.ac.uk

***Abstract*—In high-level synthesis, scheduling maps operations into clock cycles. It can either be done at compile time (statically) or run time (dynamically). There has been recent interests in dynamic scheduling as it can potentially achieve a better performance. The state-of-the-art dynamically scheduled HLS tool Dynamatic generates dataflow-style hardware in a netlist of pre-defined components connected using handshake signals. The memory operations are executed by a component named load-store queue (LSQ), which can achieve run-time out-of-order memory accesses for high performance. However, the additional logic for the LSQ leads to significant area overhead compared to static scheduling.**

In this paper, we propose an area-efficient approach for scheduling memory operations at run time. We approximate the memory dependence distance to its minimal value and efficiently parallelise memory accesses in dynamically scheduled hardware. Over several benchmarks from related works, our results show that our approach achieves on average 0.2× of the area-delay product compared to the original designs using LSQs.

I. Introduction

In high-level synthesis (HLS), scheduling is a process that maps operations into clock cycles. There are two types of scheduling. First, static scheduling determines the start time of each operations at compile time, which enables efficient hardware resource sharing. Second, dynamic scheduling determines the start time of each operation at run time, which enables better performance for input-dependent operations.

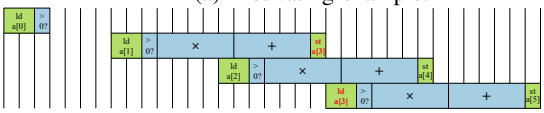
Dynamic scheduling generates hardware in a netlist of pre-defined components, connected using handshake interfaces. Each handshake interface represents a data dependence between the operations. This enables immediate execution when all the required inputs are valid.

The memory dependences can either be resolved by sequential execution [1], or load-store queues (LSQs) [2]. In order to achieve high performance, the state-of-the-art dynamically scheduled HLS tool Dynamatic [2] uses LSQs to resolve memory dependences. A LSQ allocates each memory operation to be executed into its queue and checks the dependences with its priorly executing memory operations. If there is no memory dependence, the operation can be executed at earliest time.

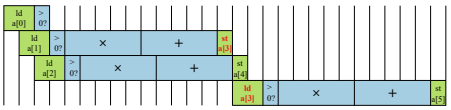
However, the area overhead of using LSQs is significant, which costs tens of thousands of LUTs. Sequential memory execution avoids such area usage, but the performance can be significantly restricted. This paper extends the state-of-the-art dynamically scheduled HLS tool Dynamatic and proposes an area-efficient approach for scheduling memory operations.

```
1 float a[M];
2 void condVecTrans() {
3   for (i = 0; i < N; i++)
4     // a[i] changes at run time.
5     if (a[i] > 0.0f)
6       // Memory dependence distance = 2.
7       a[i+2] = 2*a[i] + 1.0f;
8 }
```

(a) Motivating example.



(b) Static schedule has a conservative II for the if condition.



(c) Dynamic schedule has **2**× performance but costs **16**× area.

Fig. 1: Dynamic scheduling can achieve a better throughput, but the area overhead is significant. Our approach achieves the same performance as (c) but only costs **1.09**× area.

Our toolflow requires users to input the dependence distance of loops as constraints and preemptively starts certain iterations by the constraints while preserves inter-iteration dependences. Any intra-iteration dependence is resolved by sequentialisation inside basic blocks. The synthesised hardware benefits from the ability to skip not-taken control flow paths (over statically scheduled HLS) and a simpler, arbitration-only memory controller (over LSQs). Our contributions include:

1) A technique that dynamically schedules memory operations with a pre-defined memory dependence distances in efficient hardware;
2) A transformation pass that automatically integrates the proposed memory scheduler into dynamically scheduled hardware; and
3) Over a set of applicable benchmarks from several benchmarks, our approach achieves 0.2× area-delay product compared to the original designs with LSQs.

The rest of the paper is organised as follows: Sec. II illustrates a motivating example. Sec. III introduces necessary background and related works. Sec. IV explains the details of our approach. Sec. V evaluates the effectiveness of our work.

II. Motivating Example

Here we use a motivating example to demonstrate the trade-off between area and performance using LSQs. Fig. 1a shows a code example that transforms a vector based on the values stored in its elements. In the figure, a `for` loop checks if an array element `a[i]` is positive. If so, it updates `a[i+2]` using an expression shown at line 7. Since the array elements in array `a` are changed by the loop at run time, estimating the `if` conditions at compile time is challenging.

Static scheduler conservatively assumes the worst case of the memory dependences, leading to a constant II of 7 in the schedule shown in Fig. 1b. However, dynamic scheduling monitors the control flow at run time, leading to a dynamic II shown in Fig. 1c. All the operations execute at the earliest time, which results in a doubled throughput.

Such a dynamic II is realised by a LSQ for `a`. The LSQ ensures that the memory dependence cannot break while parallelising the memory operation. However, the area overhead of using a LSQ is significant although it can achieve some speedup. The area-delay product of the statically scheduled design is 0.12× relative to the dynamically scheduled design.

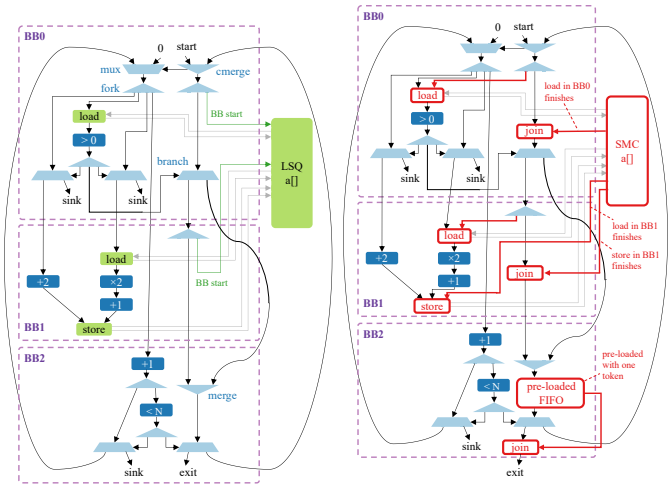
There is no memory scheduler for dynamically scheduled hardware that can dynamically parallelise memory operations with less area overhead. Dependence distance is the number of iterations that separate an operation from its dependants. This paper provides a solution by approximating the memory dependence distance from a variable to its minimal value. For the motivating example, our approach achieves the same performance as the dynamically scheduled hardware using a LSQ but with only 9% area overhead. The rest of the paper explains our approach in detail.

III. Background

In high-level synthesis, dynamic scheduling was initially proposed by Page and Luk [3]. It synthesises hardware from occam programs, which was later extended to support Handel-C [4]. Mapping C programs into a netlist of pre-defined hardware components has also been studied in both asynchronous world [5] and synchronous world [1], [2].

In the synchronous world, Huthmann *et al.* [6] propose an approach that statically maps operators into program states and dynamically schedule each program state. Dynamatic [2] and CIRCT [1] maps operators into a netlist of pre-defined components formalised by Carloni *et al.* [7]. Each dependence between two operations is translated into a handshake connection between these components. In comparison, a dependence only stalls individual operators in Dynamatic and CIRCT, while a dependence stalls the whole program state in [6]. In this paper, we use Dynamatic for prototyping.

Dynamatic uses LSQs to monitor memory dependences at run time [8]. A LSQ allocates memory operations into its queue in strict program order and early executes those that are independent from its prior and executing memory operations. Such an approach achieves a better performance with significant area overhead. CIRCT forces memory operations to execute sequentially [1]. Such conservatism simplifies the control logic but the performance is significant limited. Our approach extends the memory architecture used in CIRCT and supports parallelising memory operations. The hardware generated by our toolflow has better hardware performance than CIRCT and smaller area than Dynamatic.



(a) Dataflow graph using a LSQ. (b) Proposed dataflow graph.

Fig. 2: Our approach uses a set of components distributed in the dataflow graph to schedule the memory operations.

IV. Methodology

In this section, we first formalise the problem of scheduling memory operations. Then we demonstrate the proposed memory architecture for parallelism. Finally, we illustrate our toolflow integrated into Dynamatic for prototyping.

A. Problem Formulation

We first formalise the problem of scheduling memory operations. Let M be the set of all the memory statements in the program: $M = \{m_1, m_2, m_3, ...\}$, where $m_{i,j}$ represents the jth iteration of m_i. Let $D \subseteq N^4$ be the memory dependence set. $(i_1, j_1, i_2, j_2) \in D$ denote that m_{i_2,j_2} depends on m_{i_1,j_1}. Let $x \prec y$ denote the execution of y starts after the execution of x finishes. The memory dependence constraint is as follows:

$$\forall i_1, j_1, i_2, j_2 : (i_1, j_1, i_2, j_2) \in D \Rightarrow m_{i_1,j_1} \prec m_{i_2,j_2} \quad (1)$$

A LSQ seeks a schedule with the best performance at run time under the constraint above. This constraint is relaxed in CIRCT for sequential execution. Let $o(x)$ denote the index of the execution of x in strict program order:

$$\forall i_1, j_1, i_2, j_2 : o(m_{i_1,j_1}) < o(m_{i_2,j_2}) \Rightarrow m_{i_1,j_1} \prec m_{i_2,j_2} \quad (2)$$

Our approach also relaxes constraint 1 but less conservative than constraint 2. We approximate the memory dependence distance to its minimum d_{min} at compile time.

$$\forall i_1, j_1, i_2, j_2 : (i_1, j_1, i_2, j_2) \in D \wedge j_1 < j_2 \Rightarrow d_{\text{min}} \leq j_2 - j_1 \quad (3)$$

$$\forall i_1, i_2, j : j > d_{\text{min}} \Rightarrow m_{i_1,j-d_{\text{min}}} \prec m_{i_2,j} \quad (4)$$

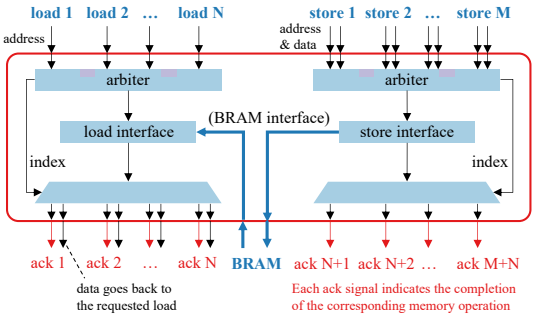


Fig. 3: The proposed Static Memory Controller (SMC) for N loads and M stores, connected to a 2-port BRAM. The black edges represent handshake connections.

The schedule is still dynamic because the data dependences and operation latencies remain dynamic, which could affect the start time of these memory operations.

B. Proposed Memory Architecture

Here we show how to generate efficient hardware that satisfies constraint 4. Dynamatic generates dataflow graphs that can be directly translated into hardware. Fig. 2a shows the dataflow graph of the code in Fig. 1a using a LSQ. In the dataflow graph, data operations are parallelised and scheduled using traditional elastic components introduced in the Dynamatic paper [2] such as muxes, forks and branches. A merge arbitrarily selects an input to output, and a cmerge is a merge but also outputs the index of the taken input. For the memory operations, a LSQ allocates the memory operations in a basic block (BB) into its queue when the BB starts execution (annotated as green edges). The LSQ checks dependences between the operations in the queue and executes them at the earliest time.

The dataflow graph using the proposed memory architecture is shown in Fig. 2b. Instead of a single LSQ, there are four types of new components used: 1) each load or store component now has an additional control input to enable the start of its execution; 2) a static memory controller (SMC) is used as a memory arbiter; 3) join components are used for synchronisation; and 4) a pre-loaded FIFO is a FIFO that contains initialised control tokens.

First, memory operations are scheduled by the edges in the dataflow graph instead of inside the LSQ. Second, the memory controller SMC now only performs arbitration. The design of SMC is shown in Fig. 3. It interfaces with a BRAM block can execute at most one load and one store operations in each clock cycle. The SMC connects to N load and M store components. Since there may be multiple requests in parallel, a round-robin arbiter is used for each port of the BRAM. For the load request, the fetched data is sent back to the load determined by the index from the arbiter. Apart from that, each memory access has an acknowledge (ack) signal at the output indicating the completion of the operation.

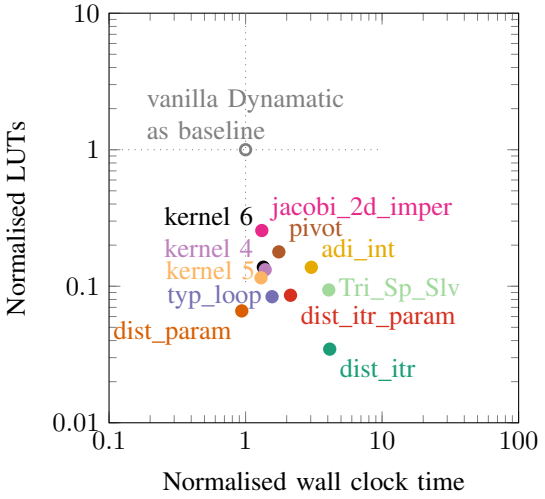


Fig. 4: Comparison of our approach with vanilla Dynamatic.

Third, the edges for scheduling the memory operations are inserted using the same algorithm in CIRCT. The following always hold: In each BB, the first memory operation is triggered by the start of the current BB, *e.g.* the load in BB0; the rest of memory operation is triggered by the completion of all its prior memory operations in the same BB, *e.g.* the store in BB1. The branch that triggers the start of the next BB requires the completion of all the memory operations in the current BB. This is synchronised by the join components. The generated edges enable dynamic starting times of memory operations, but the memory operations execute in strict program order.

Finally, a pre-loaded FIFO is inserted right before the exiting branch of the loop to pipeline iterations. Without the FIFO, the join in BB0 and BB1 stalls the start of the next iteration until all the memory operations in the current iteration finish. The number of the control tokens initialised in the FIFO is $d_{\text{min}} - 1$. These tokens do not contain values but the each token starts the next iteration if the loop exit condition is false. This enables pipelining of memory operations across iterations. Also, the join in BB2 stalls the start of the next BB until all the memory operations in the loop finish to avoid inter-loop memory dependences.

C. Tool Flow

We integrate our work into Dynamatic HLS tool [2] for prototyping (our contributions in italic): 1) the input C program is translated into LLVM IR; 2) each operation is translated into a node in the dataflow graph, and the data dependences between operations are translated into edges between nodes; *3) the proposed memory components are inserted;* 4) the dataflow graph is buffered to improve performance [9]; and *5) the dataflow graph is translated into RTL code as the final design.*

V. Experiments

In this section, we evaluate our work on open-sourced benchmarks. We compare our design with the designs generated by Xilinx Vivado HLS and vanilla Dynamatic in total

TABLE I: Comparison of our approach with vanilla Dynamatic and Vivado HLS. ADP = normalised area-delay product relative to vanilla Dynamatic. Fmax in MHz and execution time (Exec time) in μs. d_{min} = minimal dependence distance.

Benchmarks	Dynamatic					Our work							Vivado HLS					
	LUTs	DSPs	Cycles	Fmax	Exec time	d	LUTs	DSPs	Cycles	Fmax	Exec time	ADP	II	LUTs	DSPs	Cycles	Fmax	Exec time
dist_itr	15741	2	133	108	1.23	1	545	2	610	120	5.10	0.14	9	258	2	902	120	7.52
dist_param	18081	2	119	109	1.09	8	1193	2	123	120	1.02	0.06	2	265	2	902	120	7.52
typ_loop	18661	0	256	110	2.32	8	1564	0	283	78	3.63	0.13	3	63	0	721	258	2.79
jacobi_2d_imper	22766	11	6758	87	77.3	8	5826	11	10880	107	102	0.34	26	632	5	27002	120	225
Tri_Sp_Slv	18298	8	226	86	2.63	1	1719	8	1003	94	10.7	0.38	13	495	5	1191	120	9.9
adi_int	129114	81	9374	70	134	1	17768	81	27450	67	407	0.42	35	2301	16	11361	120	94.7
dist_itr_param	18605	2	319	109	2.93	8	1598	2	615	98	6.25	0.18	9	276	2	2101	120	17.5
pivot	22064	8	38423	81	474	1	3940	8	64174	77	831	0.31	13	641	5	83442	120	695
kernel4	21465	6	3157	70	44.9	1	2837	6	4362	70	62.5	0.18	6	182	3	3585	128	28.0
kernel5	19310	5	15523	100	155	1	2226	5	19010	94	202	0.15	13	464	5	25501	120	213
kernel6	20141	5	5381	89	60.5	1	2777	5	6666	82	81.7	0.19	10	525	5	7233	120	60.3
Geom. mean	1×	1×	1×	1×	1×	-	**0.1**×	1×	2.1×	1×	2.1×	**0.2**×	-	0.02×	0.8×	3.7×	1.5×	2.9×

area and wall clock time. To ensure fairness, we use the same optimisation technique in Vivado HLS and Dynamatic – only pipelining. The latency in clock cycles was obtained using Vivado XSIM simulator, and the area was obtained from the post Place & Route report in Vivado. The FPGA device we used for result measurements is xc7z020clg484. The version of Xilinx software is 2019.2.

A. Benchmarks

We aim evaluate the effectiveness of our approach for general cases, although our approach is ideal for benchmarks that have input-dependent control flow and constant memory dependence distance. We choose an open-source benchmark set by Liu *et al.* [10] and some benchmarks from Livermore loops [11]. The source[1] and results[2] are both open-sourced.

B. Overall Results

The results over these benchmarks are shown in Fig. 4. In Fig. 4, each point shows the normalised area and performance of our designs compared to the design generated by vanilla Dynamatic. Left means better performance and down means smaller area. Most of the points are in the fourth quadrant, meaning that our designs are smaller but slower than the designs from vanilla Dynamatic. The area reduction is because of replacing LSQs with our efficient memory controller; and the performance loss is because of the restriction of constant memory dependence distances.

Detailed results are shown in Table I. Compared with vanilla Dynamatic, our approach saves significant LUTs but has worse throughput because of approximating the memory dependence distance. Compared to Vivado HLS, our approach has larger area but better throughput in most benchmarks because of preserving the dynamically scheduled control and data path. On average, our approach achieves 0.2× area-delay product compared to vanilla Dynamatic.

VI. Conclusion

The state-of-the-art dynamically scheduled HLS tool can achieve speedup compared to statically scheduled HLS tools. However, it also causes significant area overhead when resolving memory dependences. We propose a novel approach that efficiently pipelines the memory operations with significantly smaller area overhead. Our results have shown that our approach can achieve on average 80% area-delay product reduction when switching from vanilla Dynamatic to our toolflow. Our future work is to automate determining the minimal memory dependence distance.

[1] DOI: 10.5281/zenodo.7217711 [2] DOI: 10.5281/zenodo.7217998

References

[1] CIRCT contributors, "Handshake dialect," https://circt.llvm.org/docs/Dialects/Handshake/, 2022.

[2] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CA: ACM, 2018, pp. 127–136.

[3] Ian Page and Wayne Luk, "Compiling occam into Field-Programmable Gate Arrays," in *FPGAs, W. Moore and W. Luk, Eds., Abingdon EE&CS Books*, 1991.

[4] Celoxica, "Handel-C," 2022. [Online]. Available: http://www.celoxica.com

[5] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2021, pp. 1–8.

[6] J. Huthmann, J. Oppermann, and A. Koch, "Automatic high-level synthesis of multi-threaded hardware accelerators," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[7] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[8] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 125:1–125:19, Sep. 2017.

[9] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 186–196. [Online]. Available: https://doi.org/10.1145/3373087.3375314

[10] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for hls," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 159–162.

[11] J. T. Feo, "An analysis of the computational and parallel complexity of the livermore loops," *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167819188900373