VLIWs, and How to Schedule Them

Rupanshu Soi

March 2026

1 Introduction

VLIW, short for Very Long Instruction Word, is a design philosophy for processor architecture that originated in the early 1980s, intended to extract large amounts of instruction-level parallelism (ILP) from scientific code. ILP is the ability of a processor to simultaneously execute, or significantly overlap the execution of, multiple instructions. The next section will lay out the philosophy in detail, but let me preface the discussion by stating that VLIWs are statically-scheduled machines, meaning the processor executes the sequential instruction stream produced by the compiler, without additional reordering or compaction (see Section 2.3). Therefore, the compiler must employ sophisticated scheduling algorithms to produce an instruction stream with high amounts of ILP; otherwise, the machine will perform poorly.

Why care about VLIWs? While it is true that VLIWs are no longer popular in general-purpose computing, digital-signal processors like the Qualcomm Hexagon are still VLIW. But that is not primarily why I am interested in the topic. My interest is due to the fact that several current AI accelerators have also converged on a VLIW architecture, including the Google TPU [6], Groq LPU [4], Intel Gaudi [5], and, to a large extent, the GPU.

In this report, I study two families of scheduling algorithms for VLIWs. The first family is of trace and percolation scheduling—their goal was to extract ILP from long sequences of instruction, across dozens of basic blocks. The second family is of modulo scheduling, which narrows its scope to loops only. I look at the historical context in which the ideas originated, what worked, what didn't, and what we can learn from the entire saga for programming current AI accelerators.

2 VLIW Architectures and the ELI-512

I must begin by setting the scene for this paper [3]. Joseph Fisher, as a graduate student at NYU in the late 1970s, invented trace scheduling, a technique for extracting instruction-level parallelism (ILP) across basic blocks. Trace scheduling was so much better than other techniques at extracting ILP, most of which were limited to a single basic block, that there was no machine to take advantage of this newfound parallelism.

When Fisher moved to Yale as an assistant professor in the early 1980s, he sought to build such a machine, calling them VLIWs. The paper coins the term, lays out the design philosophy, and details his group's efforts at building the first practical VLIW along with all the requisite system software.

2.1 Microcode

VLIWs can be seen as a descendant of so-called horizontally microcoded machines. In microcoded processors, each assembly instruction is not implemented by the hardware directly; instead, it is interpreted by the processor by executing microinstructions. Implementing a processor via microcode offers flexibility: complex instructions, such as x86's `mov`, may be implemented by a series of microinstructions, and architectural bugs may be fixed by modifying microcode alone, avoiding a change to the silicon.

Microcode is further divided into horizontal and vertical based on the amount of encoding present in microinstructions. Horizontal microcode is unencoded—each bit of a horizontal microinstruction steers exactly one control signal inside the processor, allowing exploitation of parallelism within the processor's datapath.[1] VLIWs lift the idea of steering several pieces of the processor in a single microinstruction from horizontal microcode to the assembly level.

[1] Vertical microcode, on the other hand, is highly encoded to save space and resembles a RISC-like instruction set. While horizontal microcode can be directly executed by the processor, execution of vertical microcode may require additional hardware logic, like decoders.

2.2 Characteristics of VLIWs

The paper laid out the three main characteristics of VLIWs:

1. There is a single, statically-scheduled instruction stream.
2. Each instruction may issue several operations, each of which runs on an independent functional unit on the machine. E.g., the ELI-512, the VLIW being built by Fisher's group at the time of writing the paper, could do 16 ALU operations, 8 memory references, and 32 register accesses in each instruction.
3. Each operation requires a statically-known number of clock cycles to finish.

The paper explains the advantages of VLIWs over vector machines. Fisher claimed the main drawback of vector machines is that they offer a limited amount of speedup—usually 2 or 3×—which is much less than the 10 or 30× he hoped to achieve with trace-scheduled code on VLIWs. Furthermore, vector machines can only speed up inner loops, meaning that the maximum possible speedup is further limited by Amdahl's law. Although Fisher seemed to view VLIWs and vector machines as mutually exclusive, it is possible to have the best of both worlds by making individual operations on a VLIW operate over vectors or tiles, resulting in a programming model resembling modern GPUs.

2.3 Comparison with Superscalars

Though absent from the paper, it is also instructive to compare VLIWs with superscalars, popular in general-purpose computing today. Like VLIWs,

superscalars satisfy property (2), but unlike VLIWs, they are not statically scheduled: the compiler emits a sequential instruction stream which may be reordered and compacted[2] by the processor during execution. Superscalars, like modern x86 processors, dedicate a lot of chip area and energy to instruction reordering and compaction, which makes the hardware more complex but simplifies the compiler. VLIWs pick essentially the opposite end of this trade-off: the hardware of a VLIW is much simpler, but obtaining maximum performance requires the compiler to employ sophisticated scheduling algorithms to find enough ILP.

[2] Compaction is the process of taking several instructions and "packing" them into a single new instruction.

2.4 Trace Scheduling

Fisher's vision for VLIWs was to provide speedups of an order of magnitude or more on scientific code. Individual basic blocks do not contain this amount of parallelism, necessitating an approach that extracts it from long sequences of instructions, spanning many basic blocks. The paper presents trace scheduling as this approach.

In trace scheduling, the program is first profiled to obtain *traces*, paths of execution that are taken with high probability. Note that traces may not contain back-edges; I explain how those are handled later. The trace with the highest probability of execution is scheduled, i.e., reordered and compacted, as if it were a basic block. Jumps within a trace can simply be ignored in this step. However, any code motion that would clobber the value of a variable read off the trace is not allowed. The trace is stitched back to the rest of the program by adding compensatory code at any point where execution may enter or exit the trace. Thereafter, this process is repeated for the trace with the next highest probability of execution, and so on until only code with little probability of execution is left.

How does trace scheduling handle back-edges, i.e., loops? The solution offered is simple if crude: unroll the loop's body a couple of times, and apply trace scheduling to it. Doing so amounts to a crude form of software pipelining, but was acceptable at the time because optimal approaches to software pipelining would not be developed for several more years. The paper is sparse on details of trace scheduling—they are published elsewhere—so I skip them in this report as well.

2.5 Compiler Optimizations

Bulldog[3], the first trace-scheduling compiler, was being designed and built at the time of writing the paper. Much like compilers today, Bulldog had to deal with anti-aliasing in order to do the code motions required for trace scheduling. General anti-aliasing is difficult and expensive, but here, as in many other areas, Fisher and his students relied on properties specific to scientific code. They observed that inner loops of scientific programs rarely had arbitrary pointers–memory references, when they existed, were almost always to arrays. This observation reduces the anti-aliasing problem to one of solving Diophantine equations. E.g., to determine if `A[2*i + 1]` aliases with `A[j]`, one only needs to solve `2*i + 1 = j`

[3] John Ellis's dissertation describing Bulldog won the 1985 ACM Doctoral Dissertation Award. Ellis was one of only three PhD students Fisher supervised in his career.

under ranges for `i` and `j`, and range analysis methods were known at the time.

Bank conflicts also rear their ugly head. The ELI-512's memory system was unusual: if the bank for a memory access was known, the access was serviced through a fast path. Otherwise, the access had to go through a "global arbitration system", the slow path, which sequentialized all accesses to determine the bank responsible for each one. One of Bulldog's optimizations therefore was to rewrite array accesses in a loop to use the fast path. It did so using a two-part scheme. Say the loop is `for i in l..u` containing the reference `A[i]`, and the number of banks is `B`. First, Bulldog would generate a prologue beginning at `i = l` using the slow path, exiting once `i mod B == 0`, which guarantees that `A[i]` thereafter would be in bank 0. After adjusting the main loop's bounds, Bulldog would unroll it `B` times, resulting in the references `A[i], ..., A[i + B - 1]`. Since the bank for each successive reference is one more than that of the previous, and the starting bank is known, Bulldog can code-generate the entire loop to use the fast path of the memory system. In a retrospective written almost two decades after the fact [2], Fisher conceded that the two-tiered memory system design was baroque, and it would have been sufficient to simply avoid references to the same bank.

2.6 Limitations

In my view, a central limitation of trace scheduling is the complexity of the method of adding compensatory code. I quote the TS paper [1]: "The details of the bookkeeping phase are very complex and their formal presentation is unintuitive. Instead of a careful algorithm, a more informal, hopefully clearer explanation is given." The complexity also precluded a formal proof of correctness of the algorithm. In the next section, we will see how percolation scheduling alleviates these problems.

3 Percolation Scheduling

Percolation scheduling (PS) [9], developed by Alex Nicolau, a student of Fisher, is a descendant of trace scheduling that boils down the code motions required in trace scheduling to local rewrites, resulting in a much simpler algorithm that is also more amenable to analysis. In PS, optimization is performed by the sequential application of four local, semantics-preserving rewrites: `delete`, `move-op`, `move-cond`, and `unify`. Rewrite application is aided and guided by several "layers". E.g., a layer may do alias analysis to remove false dependencies to enable the application of more rewrites, or guide the rewrite application by using heuristics.

3.1 `move-op`

To give a flavor for the technique, I discuss the `move-op` rewrite in detail. PS uses a specialized variant of control-flow graphs called "parallel program graphs". A node a is a tuple (O, T, N) where O is a set of operations

such as assignments or additions, T is a set of conditional jumps, and N is the default continuation of a, a control-flow edge to another node in the graph. Each test in T is a tuple (p, t, f) where p is a test, and t, f are continuations to be taken if p evaluates to true or false respectively. Tests along with their continuations are constrained to form a binary decision tree: some continuations point to other tests in T, giving the branches of the tree, and some point to other nodes in the graph, giving the leaves. There is a single root of the decision tree and no cycles. Execution of a parallel program graph begins at the root node of the graph (distinct from the roots of the decision trees). When a node is executed, operations in O and tests in T are executed in arbitrary order[4] though the reads of a variable are constrained to happen before writes to that variable. If T is non-empty, the decision tree is evaluated to determine the continuation (i.e., the control-flow edge along which execution will proceed), otherwise the default continuation N is chosen. The mapping to a VLIW like the ELI-512 is apparent.

[4] Ideally, with enough machine resources, the execution of operations and tests in a node would proceed concurrently.

The objective of PS is to maximize the number of parallel operations in the graph, a proxy for which is moving operations as close to the root node as possible ("percolation"). This is achieved by applying four primitive rewrites, one of which is `move-op`. As the name suggests, this rewrite moves an operation from a node b to a node a through edge $a \to b$. There are two preconditions for this rewrite:

1. o does not have a data dependence on any operation in a.
2. o does not write to a variable that is live on any outgoing edge from a except $a \to b$.

The rewrite is given as follows:

1. Make a copy b' of b.
2. For each node c in the parallel program graph, update each continuation of c that points to b to point to b' instead.
3. Add o to the set of operations in a, and remove it from the set of operations in b.

Repeated application of this rewrite will move operations closer to the root of the graph, increasing ILP in the program, but at the cost of creating an extra copy of o at each step. The extra copies may lead to an exponential blow up in code size (see Section 3.4). Of the other rewrites, `delete` removes empty nodes, `move-cond` moves conditional jumps, and `unify` deduplicates operations across two nodes by moving them to a common predecessor node.

3.2 Layers

Since the rewrites in PS operate over local information, lacking any global view of the program, higher-level guidance is necessary to make the technique more effective. This guidance is organized into so-called layers, of which there are six in total. I discuss three to give a flavor for the technique. The support layer performs alias and flow analyses to remove extraneous data dependencies, enabling the application of more rewrites. The guidance layer uses several heuristics to prioritize rewrite applica-

tion. E.g., it prioritizes movement of nodes that have a high probability of execution. This heuristic is similar to TS but more general as it applies to individual nodes, not entire traces. This layer may also allow a user to guide rewrite application manually–the system would handle the nitty-gritty and provide feedback if the user's choice was unsound. In this, Nicolau can be seen to anticipate modern user-schedulable languages like Halide 30 years in advance. Until this point, PS has incorporated no machine-specific information. This problem is addressed by introducing the mapping layer which includes a machine model, and thereby enables, for instance, register allocation and resolution of resource conflicts. Unfortunately, more detail about how these are done in PS is not provided in the paper.

3.3 Comparison with Equality Saturation

It is instructive to compare PS with equality saturation (EqSat), a technique for program optimization based on applying rewrites that has gained popularity in recent years [15]. To give a one sentence summary: semantics-preserving rewrites are applied over an e-graph, a data structure for storing equivalence classes (e-classes), and afterward the best program, as defined by a cost function, contained in the e-graph is extracted. The main similarities between PS and EqSat are that both rely on semantics-preserving, local rewrites and attempt to guide rewrite application by some means.

The differences are more interesting. First of all, while the goal in PS is to maximize ILP, the goal in EqSat is more general: to find the best program under a given cost function. In principle, any property of the program, including ILP, may be optimized via EqSat, though in practice there are several challenges.[5] Second, the method of guiding rewrites is different. Guidance takes two forms: e-class analyses and rewrite scheduling. E-class analyses are akin to the support layer in PS; analyses may be used to attach metadata to e-classes, which may then be used to guide rewrites. Rewrite scheduling is loosely analogous to the guidance layer, allowing the user to prioritize a set of rewrites over another, or apply a set of rewrites until a condition is reached, etc. Third, EqSat has a completely separate phase for extracting the final program from the myriad represented in the e-graph. This is, of course, absent from PS which has no notion of equivalence classes.

[5] This margin is too small to list the myriad challenges one encounters when applying EqSat to an application domain, but, the program representation and cost function are important and often difficult design considerations.

3.4 Limitations

While PS marks a significant advance over trace scheduling, including a proof of soundness and termination of the algorithm, the main limitation is that a full implementation of PS was not finished when the paper was written. Therefore, many heuristics remain unproven, and the paper does not perform an experimental evaluation of PS, or comparison against prior approaches.

Taken together, trace and percolation scheduling may be viewed as

"static branch prediction". In the former, the trace is a sequence of branch predictions, and in the latter, a branch is predicted when a conditional jump is percolated via the `move-cond` rule (not discussed in this report). Compensation code, added at the end in trace scheduling and at each step in PS, restores execution if the prediction turns out to be wrong. This perspective also brings to light a central limitation of both techniques. Since a program of size N may have $O(\exp N)$ execution paths, in the worst case both techniques will have exponential running time, and may also blow up code size by an exponential amount, by selecting each execution path and "predicting" it.

4 Modulo Scheduling

Now I turn to the second family of scheduling algorithms. Although the term never actually appears in it, this paper [10] introduces modulo scheduling, which remains the de facto standard approach to implementing software pipelining today (see Section 6). While modulo scheduling is like trace scheduling in that both were invented for horizontally microcoded machines and not VLIWs (see Section 2.1), it represents a marked departure from Fisher's vision—Rau and Glaeser were interested in optimizing loops only, reasoning that loops constitute a large fraction of the runtime of scientific code.

Modulo scheduling is concerned with computing a schedule, i.e., given a loop, an assignment of the instructions in the loop's body to clock cycles. A crucial design choice is that the schedule is constrained to be identical for each loop iteration. Valid schedules must satisfy two kinds of constraints: latency and resource. The latency constraint ensures that an instruction is not scheduled before all its inputs have been produced. Formulating this constraint requires knowing the exact cycle count latency of each instruction or functional unit on the machine. Note that machines considered in the paper, like VLIWs, did not have caches, so even memory operations would execute in a fixed amount of time.

The resource constraint ensures that no functional unit is overutilized by the schedule. A key insight of the paper is to consider resource usage modulo II, short for "initiation interval", defined as the number of clock cycles between initiation of successive iterations of the loop being scheduled. This constraint is implemented by making a table where the rows run through clock cycles 0 to II - 1 (both inclusive), and the columns run through functional units (multiple copies of the same functional unit get their own column). When an instruction is being placed into the schedule, its resource usage is marked in the table. A placement is invalid if its resource usage must overlap with the usage of a previously-scheduled instruction.

A modulo schedule is optimal if it achieves the minimum possible II, as that corresponds to maximum throughput. The paper gives an algorithm for optimal modulo scheduling for loops without loop-carried dependences, and a heuristic backtracking algorithm for loops with loop-

carried dependences. The problem is much harder in the latter case and requires expensive branch-and-bound[6] methods to solve optimally. In the heuristic algorithm, if a schedule cannot be found at the minimum II, the II is increased by 1 and the process repeats. Note that increasing the II decreases throughput but simplifies the scheduling by introducing more slack in the schedule. A detailed treatment of modulo scheduling and these issues can also be found in chapter 10 of the Dragon book.

[6] Branch-and-bound is a general class of optimization algorithms that rely on splitting the current space into sub-spaces (branching) and eliminating a sub-space if it provably cannot contain the optimal solution (bounding).

4.1 Interconnects or Registers?

The rest of the paper is concerned with a new architecture containing a "polycyclic interconnect"—a network connecting all the functional units on the processor with each other, so values produced by one functional unit could directly flow to the consuming functional unit, without having to be temporarily stored in registers. Furthermore, the interconnect could insert a programmable delay in the transmission of values, and this capability is relied upon for realizing a modulo schedule.

The paper claims that this architecture is superior for modulo scheduling to conventional register-based machines because the interconnect is large enough to support any valid modulo schedule. That is, given any modulo schedule satisfying the latency and resource constraints, it is always possible to program the delays of the interconnect to realize the schedule. As the delays can be generated by a separate lower-level system, this guarantee simplifies the scheduling algorithm by allowing it to be oblivious of the interconnect. In contrast, the paper tries to argue, modulo scheduling for conventional machines with registers is much more complex, as the register file is rarely large enough to support every modulo schedule. Therefore, modulo scheduling for conventional machines needs to be done in concert with register allocation. If a schedule cannot be register-allocated, the scheduler must increase the II and try again.

However, I remain unconvinced by this argument. I see no fundamental reason why the interconnect is superior to registers for modulo scheduling. The difference is simply that the paper assumed (perhaps because it was true for the processor it was written for) that the interconnect had enough capacity to store all the values in flight at any instant, but the register file may not. The paper notes that a capacity of 4 words between every two functional units is sufficient. But this makes the comparison unfair, as, if we assume the register file had enough capacity for any valid modulo schedule, then modulo scheduling need not be done in concert with register allocation for register-based machines too!

4.2 Limitations

Although dataflow and systolic architectures today connect some functional units directly to one another, the all-to-all crossbar of the polycyclic interconnect is rarely seen. The main limitation of modulo scheduling compared to the previous family is its limited support for control-flow. The algorithm as presented in this paper cannot handle control-flow in-

side the body of a loop. Modulo scheduling was extended for this setting by Monica Lam [7], but that work came much later. In contrast, trace and percolation scheduling handle control-flow in a natural way.

5 Optimal vs. Heuristic Modulo Scheduling

By the early 1990s, there was a divide in the theory and practice of modulo scheduling: industrial compilers relied on modulo scheduling heavily but implemented it via heuristics. Researchers, especially Guang Gao's group, had developed optimal methods for modulo scheduling using integer-linear programming (ZLP). In 1995, Monica Lam, who had also developed a heuristic algorithm for modulo scheduling, challenged Gao to compare his group's optimal approach to a well-tuned industrial implementation, leading to the birth of this paper [11].

The first contender was the industrial compiler developed by Silicon Graphics (SGI). Its opposition, representing the optimal approach, was developed at McGill University. The McGill compiler contained an in-house ZLP solver which found the pipeline with the highest throughput under its machine model. The study was performed by integrating the optimal pipeliner of the McGill compiler into the other one, and measuring the performance differences that resulted on the SPEC92 floating-point benchmarks.

The bottom line of the paper is that the heuristic compiler worked really well. The optimal approach found a tiny speedup (<5%) in a small number of cases, 2/14. In 4/14 cases, the two approaches were tied. For the rest, the heuristic pipeliner generated faster code! How could a heuristic approach outperform an optimal one? Part of the reason was that bank conflicts were not part of the McGill compiler's machine model. The SGI compiler included heuristics to reduce bank conflicts, allowing it to come out on top.

5.1 Heuristics Galore

The SGI compiler used a branch-and-bound algorithm with aggressive pruning. An important heuristic is scheduling priority, i.e., the order in which instructions are considered for scheduling. It is difficult to tell in advance which heuristics will be most effective. E.g., topological order may work well because it allows the placement of consumers as close to the producer of a value as possible, resulting in shortened live-ranges. But reverse topological order may also work well for exactly the opposite reason: by scheduling consumers before the producer, one knows the latest point in time at which the producer may be scheduled to ensure the value will be ready on time. Doing so also shortens live-ranges. Indeed, the SGI compiler used several different scheduling priority heuristics, removing any of which was known to cause a performance regression in their tests. For some benchmarks with multiple loops, each loop required scheduling with a different heuristic to achieve the maximum overall performance.

Interestingly, the paper found that the scheduling priority heuristics were effective for the optimal pipeliner too! The order in which the McGill ZLP solver traversed the branch-and-bound search space turned out to be the most important factor affecting whether it could solve the problem in time. Even today, different ZLP (and SMT) solvers have wildly different performance characteristics for different problems due to reliance on different internal heuristics, so much so that standard advice suggests calling several different solvers in parallel and taking the first solution. I believe the paper's observation is an instance of the same underlying phenomenon.

5.2 Takeaways

Even though, at the time, the paper vindicated the heuristic approach, I found its heuristics ugly. In contrast, the simplicity of the optimal pipeliner stands out. The prevailing view even today is that production compilers must be blazing fast, precluding the use of solvers for NP-hard problems such as scheduling. I submit two counterpoints to this view. First, how is one to design effective heuristics? I believe the development of optimal methods can be an extremely useful tool to inform heuristic design (and on this point the paper agrees with me). Otherwise, developed heuristics may be ad hoc and ungeneralizable to new programs or architectures. Second, I disagree that production compilers always need to be fast. Only during interactive development does one require a fast compiler. It is perfectly acceptable for production compilers to have a slow, optimal path that is only used during production builds or continuous integration. Widespread adoption of languages like Rust has begun to normalize slower compilation times for production builds. I only hope this trend continues.

6 Conclusion

This report compared two families of compiler techniques for scheduling VLIWs—trace and percolation scheduling, and modulo scheduling—in the appropriate historical context, as well as from a modern perspective. In the end, Fisher's vision of speedups of 10-30× were never realized. Later studies about the limits of ILP [14] indicated that real programs, evaluated under realistic assumptions about the compiler, could only achieve speedups of 5-7× with ILP.

AI accelerators have converged to a VLIW design mainly due to reasons of power efficiency—superscalars dedicate a lot of power and chip area to runtime reordering and compaction, which is avoided in a VLIW. Adoption has also been helped by AI being a limited domain, so programs have additional structure which can be relied upon by compilers.

In the beginning, I asked what we can learn from these papers for programming AI accelerators. Some colleagues and I have already applied optimal modulo scheduling to GPUs [13], generalizing prior work which

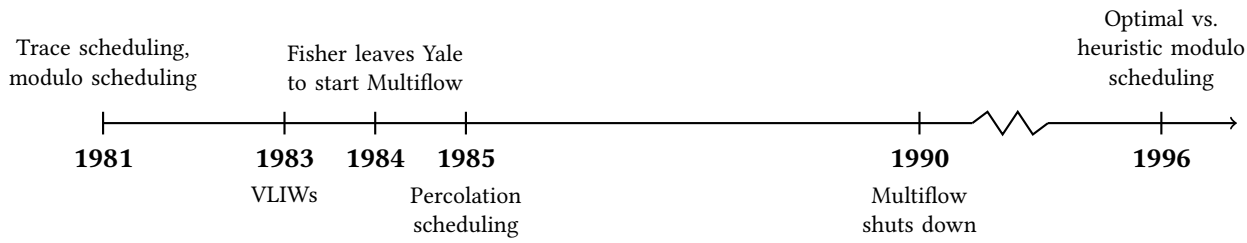

Figure 1: Timeline of events.

derived schedules by hand [12]. Our work would not have been possible without the research covered in this report.

What about trace and percolation scheduling? The key advantage of these techniques over modulo scheduling is their superior handling of control-flow. But, at least for the time being, neither do popular AI architectures require complicated, data-dependent control-flow, nor does such control-flow achieve peak performance on accelerators. In these settings, modulo scheduling, perhaps extended to handle control-flow [7], works well, though the situation might change with the evolution of AI.

I should like to end with an open research problem in applying VLIW techniques to accelerators. As noted previously, VLIWs did not have caches, but, for instance, GPUs do, meaning that certain memory operations might have variable latency depending on whether they get a cache hit or miss. Variable latency throws a wrench in the works for all three scheduling algorithms, as one of the core assumptions is a fixed and statically-known latency for each instruction. Extending VLIW techniques to handle programs in which a small fraction of instructions exhibit variable latency is ongoing research.

A Timeline

Figure 1 gives an inexhaustive timeline of events relevant to the papers discussed in this report. Fisher left his professorship at Yale to found Multiflow to commercialize VLIWs. After a brief period of success, it shut down as it could not compete with single-chip CPUs, mainly from Intel. Moore's law was then alive and single-chip CPUs could be clocked higher and consumed less power than Multiflow's chips. Lethin [8] provides a good summary of the entire story.

References

[1] Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". In: *IEEE Transactions on Computers* C-30.7 (1981), pp. 478–490. DOI: 10.1109/TC.1981.1675827.

[2] Joseph A. Fisher. “Retrospective: very long instruction word architectures and the ELI-512”. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*. ISCA ’98. Barcelona, Spain: Association for Computing Machinery, 1998, pp. 34–36. ISBN: 1581130589. DOI: 10.1145/285930.285945. URL: https://doi.org/10.1145/285930.285945.

[3] Joseph A. Fisher. “Very Long Instruction Word architectures and the ELI-512”. In: *Proceedings of the 10th Annual International Symposium on Computer Architecture*. ISCA ’83. Stockholm, Sweden: Association for Computing Machinery, 1983, pp. 140–150. ISBN: 0897911016. DOI: 10.1145/800046.801649. URL: https://doi.org/10.1145/800046.801649.

[4] Groq. *Groq LPU Inference Engine*. 2024. URL: https://groq.com/blog/inside-the-lpu-deconstructing-groq-speed.

[5] Intel. *Intel Gaudi AI Accelerator*. 2024. URL: https://docs.habana.ai/en/latest/TPC/TPC_User_Guide/index.html.

[6] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. ISCA 2017. 2017. URL: https://arxiv.org/abs/1704.04760.

[7] M. Lam. “Software pipelining: an effective scheduling technique for VLIW machines”. In: *SIGPLAN Not.* 23.7 (June 1988), pp. 318–328. ISSN: 0362-1340. DOI: 10.1145/960116.54022. URL: https://doi.org/10.1145/960116.54022.

[8] Richard A. Lethin. “How VLIW almost disappeared - and then proliferated”. In: *IEEE Solid-State Circuits Magazine* 1.3 (2009), pp. 15–23. DOI: 10.1109/MSSC.2009.933431.

[9] Alexandru Nicolau. *Percolation scheduling: A parallel compilation technique*. Tech. rep. Cornell University, 1985.

[10] B. R. Rau and C. D. Glaeser. “Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing”. In: *SIGMICRO Newsl.* 12.4 (Dec. 1981), pp. 183–198. ISSN: 1050-916X. DOI: 10.1145/1014192.802449. URL: https://doi.org/10.1145/1014192.802449.

[11] John Ruttenberg et al. “Software pipelining showdown: optimal vs. heuristic methods in a production compiler”. In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 1–11. ISBN: 0897917952. DOI: 10.1145/231379.231385. URL: https://doi.org/10.1145/231379.231385.

[12] Jay Shah et al. *FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision*. 2024. arXiv: 2407.08608 [cs.LG]. URL: https://arxiv.org/abs/2407.08608.

[13] Rupanshu Soi et al. *Optimal Software Pipelining and Warp Specialization for Tensor Core GPUs*. 2025. arXiv: 2512.18134 [cs.PL]. URL: https://arxiv.org/abs/2512.18134.

[14] David W. Wall. "Limits of instruction-level parallelism". In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 176–188. ISBN: 0897913809. DOI: 10.1145/106972.106991. URL: https://doi.org/10.1145/106972.106991.

[15] Max Willsey et al. "egg: Fast and extensible equality saturation". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.