# Analysis of Compiler Heuristics via Stochastic Scheduling and Hyper-Heuristics

## A Thesis Proposal

Department of Computing and Software

McMaster University

Curtis D'Alves

December 20, 2019

THESIS PROPOSAL .

Christopher Anand anandc@mcmaster.ca
Wolfram Kahl kahl@cas.mcmaster.ca

**Abstract**

In compiler optimization, Instruction Scheduling seeks to optimize the order of a sequence of instructions to maximize throughput while preserving semantics. As modern architectures become increasing more complex, with supported techniques/features such as super-scalar, out of order execution, register remapping, VLIW (Very Large Instruction Word) and more; opportunities to exploit Instruction Level Parallelism increase in turn. However as a known NP-Complete problem, finding the optimal schedule for a non-trivial program is too costly. Conventional compilers opt to utilize heuristics, many of which are developed on an ad-hoc basis.

We present a stochastic non-linear programming algorithm, capable of spanning all valid schedules by relaxing the problem to a continuous domain and optimizing with constraints. We combine the optimization problem with heuristics by encoding them as penalty functions that can be scaled to adjust priority. By stochastically generating scaling parameters, we can generate datasets of schedules that were obtained by optimizing a variety schedule spaces that model different combinations of priorities to sets of heuristics. So far, we have used this method to effectively find schedules for performance-critical code on IBM MASS libraries for the IBM Z15 architecture, gaining up to 20% speedup compared to previously scheduled code.

Although unsuitable for implementation in a conventional compiler, we believe the method may prove useful for hyper-heuristic based architecture analysis. By analyzing generated datasets together with the space of the model for a given architecture, we hope to gain insights into how different heuristics impact different architectures. We give an overview of current efforts for hyper-heuristic development in compilers and propose this model as a basis for developing hyper-heuristic methods for instruction scheduling.

—Source: `https://github.com/dalvescb/phd_thesis` —

# Contents

# Chapter 1

# Introduction (Background to Instruction Scheduling)

Modern processors are built with architectures capable of exploiting Instruction Level Parallelism (ILP) on a single core through pipelining and superscalar behavior. Superscalar processors of degree $n$ are architectures theoretically capable of issuing and executing $n$ instructions on $n$ parallel execution units. However maximizing throughput is not always achievable, in part due to instruction dependency (i.e the execution of an instruction cannot occur until it's operands are available). A sequence of dependent instructions of sufficient latency may achieve no benefit from a superscalar architecture. That is, unless they can be interleaved with a set of relatively independent instructions. Furthermore, resource constraints such as the number of available registers and functional units can also delay execution. This creates an interesting problem for hardware and/or compilers: selecting a schedule of instructions.

**Problem: Instruction Scheduling**

Given an a dependency graph (DAG) of instructions and resource limitations (number of registers, functional units, etc), designate an ordering (find a **schedule**) maximizing execution throughput

Even simple formulations of optimal instruction scheduling is an NP-Complete search problem [HG83], and problems attempting to integrate resource limitations on modern architectures are NP-Hard [Mot+95]. Thus practical solutions to instruction scheduling problems are given by either:

- ◇ **Heuristics** most commonly used by conventional schedulers
- ◇ **Approximation Algorithms** some experimental use done for near optimal schedules [Cos]

These solutions can implemented in hardware to execute at run-time (dynamically), or implemented at compiler time via the compiler. With the exception of certain niche architectures known as Very Long Instruction Word (VLIW) architectures that seek to move the burden of scheduling to the compiler entirely [Fis83], modern architectures approach scheduling from both the compiler and hardware. In general, comprehensive scheduling algorithms take a model of the program (typically represented as a dependency DAG [GM86]) as input and return an ordered model to be used by the assembler during compilation.

## 1.1 Types of Instruction Scheduling

There are two broad criteria that have historically influenced scheduling algorithms: the first being the control flow of the dependency graph, the second being the nature/constraints of the architecture being scheduled for [RF93]. The second criteria yields ad-hoc solutions that are difficult to categorize and often forgotten through the evolution of architectures. However an effective scheduling algorithm must consider both criteria.

Within the first criteria (scheduling based on control flow), the following categories are worth noting [RF93]:

- ◇ **Basic Block:** (local acyclic) break code into blocks within branches (most commonly performed scheduling)
- ◇ **Global Scheduling:** (global cyclic) schedule across basic block boundaries
- ◇ **Modulo Scheduling:** (local cyclic) schedules basic blocks inside of a loop, seeking to optimize by interleaving iterations
- ◇ **Trace Scheduling:** (global acyclic/cyclic) tries to optimize control flow by predicting routes taken on branches

Each of the above categories are distinguished by what consideration is given to different types of branching. Initial research into scheduling focused entirely on local scheduling (ignoring branching) [RF93] and culminated in the use of various list scheduling algorithms in most schedulers by the 80s [Fis83]. An intuitive approach to global scheduling is to first schedule basic blocks then attempt to move operations from one block to empty slots in neighboring blocks. However this approach would need to take into account/possibly reverse too many arbitrary decisions made in local scheduling in every possible neighboring block. To compensate for this, techniques for predicting more frequently occurring branch routes to improve global scheduling was invented known as trace scheduling [Fis81]. Cyclic scheduling deals with branching that conforms to a loop in the control graph, and could be dealt with in the same fashion as global/trace scheduling, however because so much performance-critical code is in looping it is important enough to have it's own class of algorithm known as modulo scheduling (discussed in a later section).

## 1.2 SuperScalar Architectures

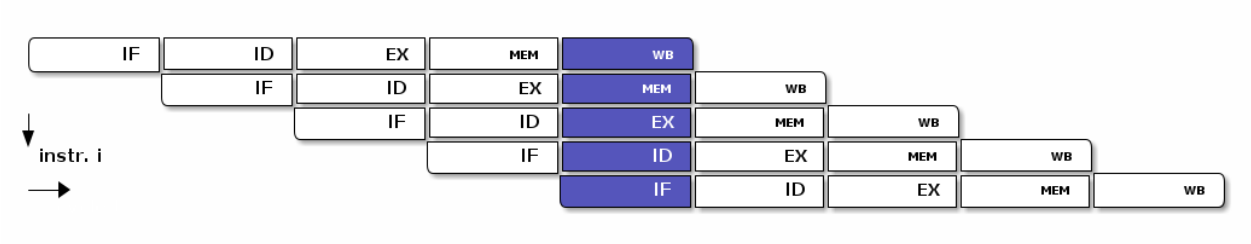


Figure 1.1: Simple Pipelined Architecture

Simple pipelined architectures issue a single instruction per cycle. Through pipeling, ILP is still exploitable, but limited by only having one of each type of functional unit. Figure 1.1 shows an example simple RISC architecture with a 5 stage pipeline (IF=Fetch, ID=Decode, EX=Execute,MEM=Memory Access,WB=Write Back) that exploits ILP while using only a single execution unit. Conversely, figure 1.2 shows a superscalar architecture that utilizes parallel execution units.

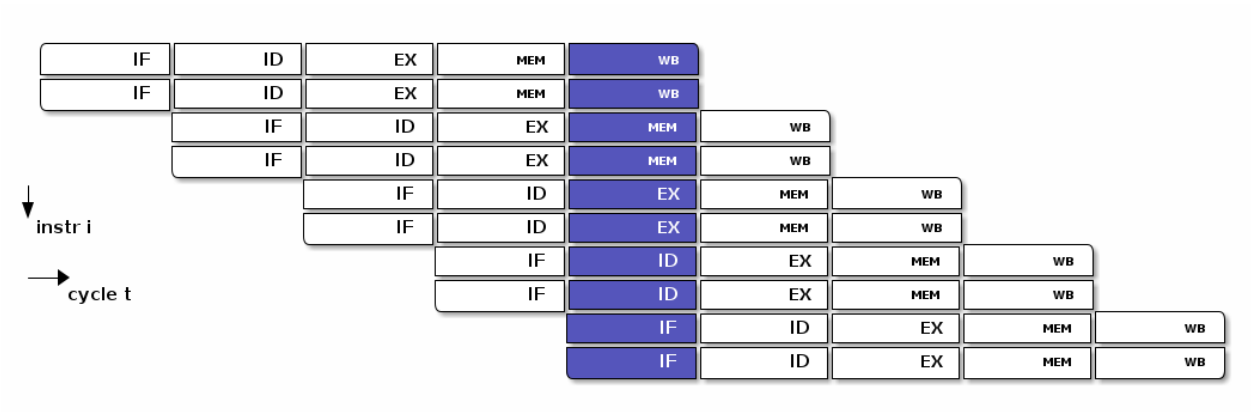


Figure 1.2: SuperScalar Architecture

Superscalar architectures are all uniprocessors that can execute two or more scalar operations in parallel, this encompasses a wide variety of architectures, but common to all these architectures is the existence of parallel and pipelined functional units, and the need to manage that parallelism [ZK01]. In particular, superscalar architectures put increased strain on resource management. This poses a more serious challenge for scheduling algorithms, since basic block scheduling is often not sufficient to allow full utilization of machine resources [BR91].

An ideal architecture to schedule for would be a RISC architecture with a collection of functional units of $m$ types, where the machine has $n_1, n_2, \ldots, n_m$ units of each type. One could view optimizing a schedule over such an architecture as maximizing the amount of live

functional units per cycle (i.e maximum throughput). This would generally be accomplished by interleaving different types of instructions, however if you stretch data dependent instructions too far apart doing this you risk running out of available registers (you increase register pressure).

## 1.3 Pipeline Stalls

Both of the previous figures 1.1 and 1.2 show an ideal schedule with **NO** stalls. A stall occurs when, because of various architecture **hazards** that can arise, full throughput cannot be achieved and a NOOP (No-Operation) instruction must be inserted. This is also known as inserting a bubble in the pipeline. Figure 1.3 gives an example of inserting a NOOP (bubble), because of a Read After Write (RAW) hazard.

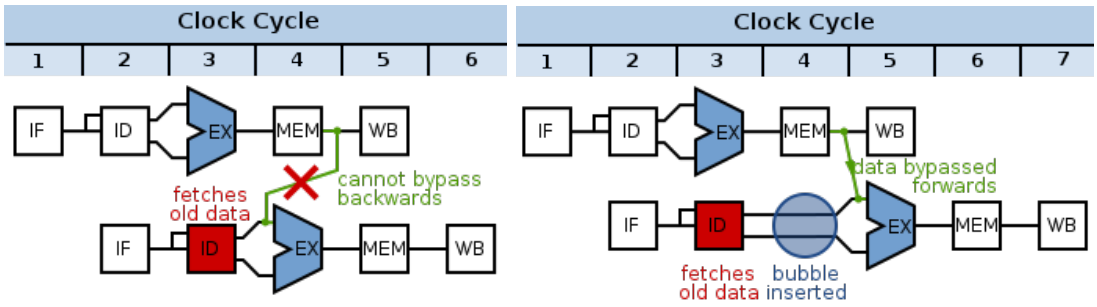


Figure 1.3: Example of a bubble (NOOP) being inserted to fix an unfullfilled data dependency

## 1.4 Hazards

Architecture hazards can be broken up broadly into three categories [**patterson2013computer**]

- **Data Hazards** occur when a data dependency is broken. There are three situations in which this can occur: read after write (RAW), write after read (WAR) and write after write (WAW)

| **RAW** | **WAR** | **WAW** |
|---|---|---|
| $\mathbf{R2} \leftarrow R5 + R3$ | $R4 \leftarrow R1 + \mathbf{R5}$ | $\mathbf{R2} \leftarrow R4 + R7$ |
| $R4 \leftarrow \mathbf{R2} + R3$ | $\mathbf{R5} \leftarrow R1 + R2$ | $\mathbf{R2} \leftarrow R1 + R3$ |

- **Structural Hazards** occurs when an aspect of hardware is accessed at the same time (such as a functional unit)

- ◇ **Control Hazards** also known as Branch Hazards, occur when a bad branch prediction is made causing instructions that were brought into the pipeline needing to be discarded

## 1.5 Register Allocation via Graph Coloring

The theory of graph coloring deals with algorithms that seek to partition a set of objects into classes, given simple rules associating objects that may not belong to the same class [**jensen2011graph**]. These algorithms operate on graphs, where the objects are vertices and the edges denote connected vertices that cannot be in the same class. Classes are represented via colors, where a **k-coloring** denotes a partitioning into k distinct classes. The problem of finding a **k-coloring** is a well-known NP-Complete problem [**jensen2011graph**].

Architectures provide a finite set of registers that must be allocated after or during instruction scheduling. Finding an allocation for a given schedule (assuming one exists) has been shown to be equivalent to the Graph Coloring problem and hence NP-Complete [**chaitin1981register**]. Given a code schedule in Single Static Assignment (SSA) form, a unique interference graph can be constructed that denotes data dependency. On an architecture with **k** registers, a register allocation is found via a **k-coloring** of vertices of this interference graph. See Figure 1.4 as an example.

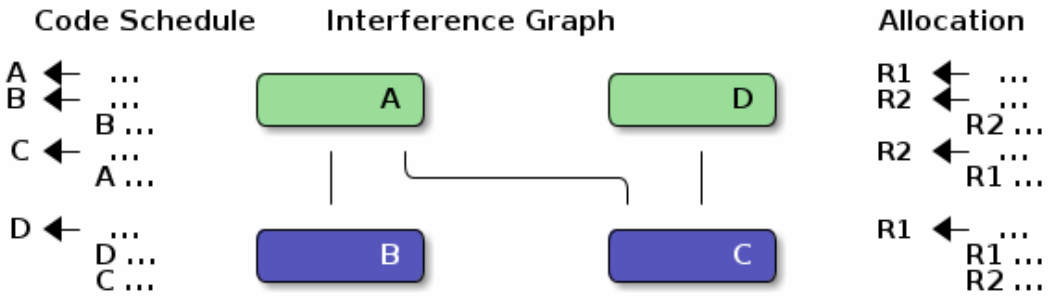


Figure 1.4: Register Allocation via Graph Coloring

## 1.6 Spilling

Finding the existence of a **k-coloring** for a given graph is itself an NP-Complete problem [**jensen2011graph**], and the absence of an existing coloring presents a difficult problem. When a schedule cannot be register allocated, variables must be *spilled* to memory (spilling is the action of storing variables into memory rather than registers [**bouchez2007complexity**]). Spilling requires the addition of new instructions to store/load from memory, which changes not only the interference graph (allowing different register allocations) but the dependency

graph as well (allowing different schedules). Therefore adding spills alters the space of valid schedules, and merits consideration when searching for a "truly optimal" schedule (although addition of spills unnecessarily is generally detrimental).

Graph coloring heuristics can be bolstered to include the addition of spilling when they fail to find a proper **k-coloring** [Cha82],[**briggs1989coloring**]. The choice of which node to spill is a cost/benefit estimation. Each edge in the interference graph can be assigned an estimated live range (sections of code which a value is defined and used but not re-defined). Eliminating longer live ranges alleviates more register pressure and creates a more flexible scheduling space.

## 1.7 Combining Register Allocation and Instruction Scheduling

Register allocation can be performed before, after, or combined with instruction scheduling, but is generally performed after [**brasier1995craig**]. Performing allocation before scheduling involves allocating on top of a "default" schedule and then manipulating the schedule while maintaining a fixed allocation. Having a fixed allocation creates new dependencies (known as *anti-dependencies*) that limit the space of valid schedules. Conversely, register allocation done after instruction scheduling is uninhibited by these anti-dependencies and may find more efficient schedules, but they may require post-hoc intervention via spilling.

Attempts to combine register allocation and scheduling are rare in conventional compilers, as even a simple instance of the problem (single register,no latency's, single functional unit) is *NP-hard* [Mot+95] [Pin93]. Heuristics developed for combining register allocation and scheduling generally involve estimating a tradeoff between controlling register pressure and instruction parallelism considerations [Mot+95].

## 1.8 Modulo Scheduling: Staging

The objective of modulo scheduling is to engineer a schedule for one iteration of the loop such that when this same schedule (known as the kernel) is repeated at regular intervals, no intra- or inter-iteration dependence is violated, and no resource usage conflicts arise between operations of either the same or distinct iterations [**rau1996iterative**]. This generally involves a sort of *loop pipelining*, where a basic block of a loop can be broken into stages and the loop can be *unrolled* to interleave stages between iterations (see Figure 1.5). Integral to this is the concept of an **Initiation Interval** or II, which is essentially the fixed delay between the start of successive iterations (see Figure 1.6). Each iteration of the loop can be divided into stages each consisting of II cycles. A smaller II corresponds to shorter execution time.

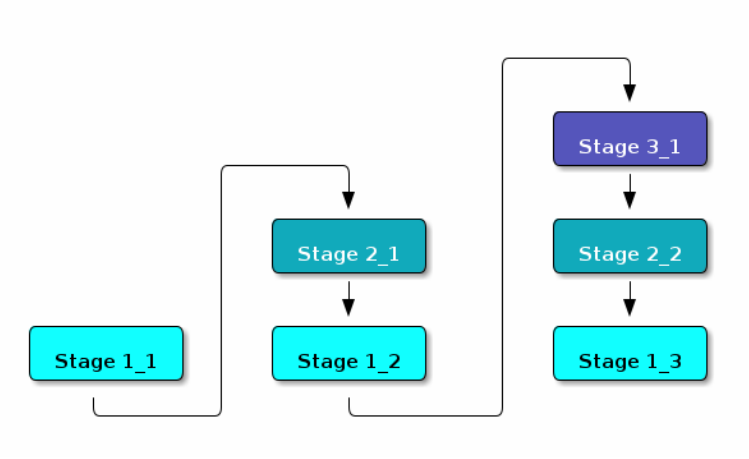


Figure 1.5: Example 3-Staged for Modulo Scheduling

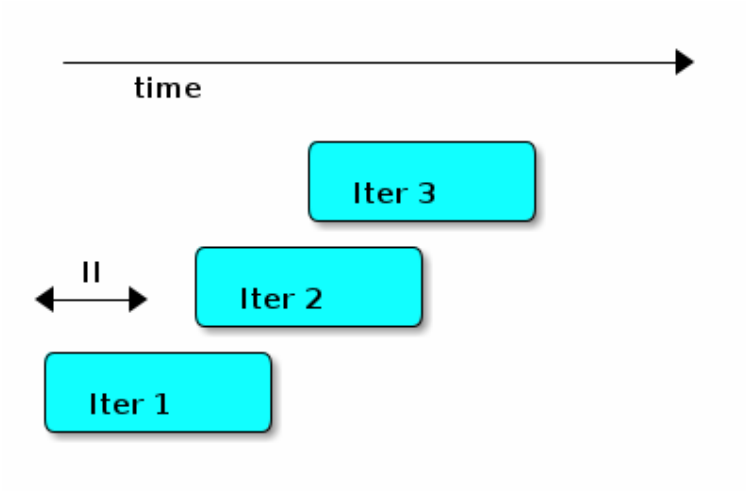


Figure 1.6: Initiation Interval

Modulo Scheduling algorithms require a candidate II be selected before scheduling is attempted. The **Minimum Initiation Interval** or MII is a lower bound on the possible value of any II for which a modulo schedule exists. The MII is constrained by both resource constraints (resMII) and recurrence constraints (recMII). The resource constraint simply holds that resources (such as functional units) must be sufficiently available and any extra latency created waiting for a resource to become available must be accounted for (the exact calculation of resMII is architecture specific and can become very complicated on Out-of-Order execution architectures, covered in a later section). The recurrence constraint lower bound is defined as the maximum, taken over all cycles C in the dependence graph, of the sum of latencies along C divided by the sum of distance along C:

$$\text{MII} = \max(\text{recMII}, \text{resMII}) \tag{1.1}$$

$$\text{recMII} = \max_{C \in DepGraph} \frac{\sum_{e \in C} latency(e)}{\sum_{e \in C} distance(e)} \tag{1.2}$$

The task of generating an optimal, resource-constrained schedule for loops with arbitrary recurrences is known to be NP-complete [Lam12]. A heuristic approach via Swing Modulo Scheduling has been implemented in the GNU C Compiler (GCC) [**hagog2004swing**].

## 1.9 Register Pressure In Staged Loops

Staging can increase throughput by enabling more instructions to be scheduled between high latency operations and subsequent use. However this also increases the number of live instances of loop variables and thus requires more registers to accommodate the schedule. Swing Modulo Scheduling (SMS) is a notable variation of modulo scheduling that utilizes a heuristic approach that aims to reduce register pressure [**gosling2000java**]. Some architectures also provide hardware mechanisms for **Register Queuing** that provide more efficient spilling.

Due to the nature of modulo scheduling, the lifetime of a variable can overlap with a previous definition of itself. To handle this, some form of register renaming needs to be provided. Some hardware provides support for this in the form of **Rotating Register Files** [**rau1989cydra**]. When no hardware solution is provided, the problem of overlapping lifetimes can be solved by a technique known as **Modulo Variable Expansion** (MVE), wherein the kernel is unrolled and multiple definitions of a variable are renamed at compile time [**valluri1998modulo**].

## 1.10 Register Remapping/Renaming

Not to be confused with renaming of registers at compile time, register renaming in hardware is a technique to remove false data dependencies — write after read (WAR) and write after write (WAW) — between register operands of subsequent instructions at runtime [**sima2000design**]. For example, a WAR hazard could be rewritten like so

| **Before** | | **After** |
|---|---|---|
| $R2 \leftarrow \mathbf{R4} + R3$ | | $R2 \leftarrow \mathbf{R4} + R3$ |
| $\mathbf{R4} \leftarrow R1 + R5$ | $\Longrightarrow$ | $\mathbf{R33} \leftarrow R1 + R5$ |

A register renaming scheme must not rename to a register that would introduce a new hazard, this would present a difficult problem were it not for the existence of **Logical** and **Physical** Registers. When executing machine code, hardware maps **Logical Registers** to **Physical Registers**

- ◇ **Logical Registers** are a set of registers usable directly when writing/generating assembly code (limited by system architecture)
- ◇ **Physical Registers** are a set of registers actually available in hardware

Having a larger number of Physical registers than Logical registers gives hardware extra flexibility when dispatching instructions for **Out of Order Execution**.

## 1.11 Out-of-Order Execution

The execution flow of an instruction on a CPU can be implemented in one of two paradigms: **in-order** or **out-of-order** (OoO, also known as dynamic) execution. Execution of an instruction cycle in each paradigm comprises different steps:

**In-Order Execution**

1. Instruction fetch
2. Stall until all operands are available
3. Dispatch to appropriate functional unit
4. Execute (on appropriate functional unit)
5. Write back to register file

**Out-Of-Order Execution**

1. Instruction fetch
2. Dispatch to a temporary queue known as **Reservation Station**
3. Wait in the reservation station until operands are available
4. Issue once operands are available (even if before an older instruction)
5. Execute (on appropriate functional unit)
6. Retire results to another temporary queue
7. Write results back to register files after all older instructions have their results written back

Although a more complicated design, OoO execution presents an opportunity to increase throughput by filling in time that would be wasted stalling in step 2 of in-order execution with instructions that are ready, then re-ordering the instructions back to appear they finished in order (known as retiring). This essentially decouples the fetch and decode stages of the pipeline from the execution stages. As the instruction pipeline deepens, there is therefore increased opportunity to dispatch out of order. When combined with Register Renaming, this is of particular benefit allowing instructions that are data independent but register dependent to be executed in parallel. OoO dispatching also provides benefits over in-order when cache misses (or high latency memory accesses in general) occur [**stark1997reducing**]. Figure 1.7 illustrates the general control flow in an OoO machine.

Out of Order execution requires dynamic scheduling (scheduling at runtime by hardware) performed via methods such as **Tomasulo's Algorithm** or a **Register Score-board**. Both methods seek to enable more efficient use of multiple execution units while preventing breaking of data dependencies. A register scoreboard checks resources for an instruction to see if the required resources are available for the instruction to execute, and if so allows dispatch. The scoreboard record (locks) the resources that would be modified by the instruction at issue time, and any subsequent instructions that want to access those resources cannot be issued until the instruction that initially locked them subsequently unlocks them [**popescu1997processor**]. Tomasulo's Algorithm innovated upon score-boarding by allowing improved parallel execution although requiring new hardware features such as register renaming in hardware, reservation stations for all execution units and a common data bus between all reservation stations [Tom67].

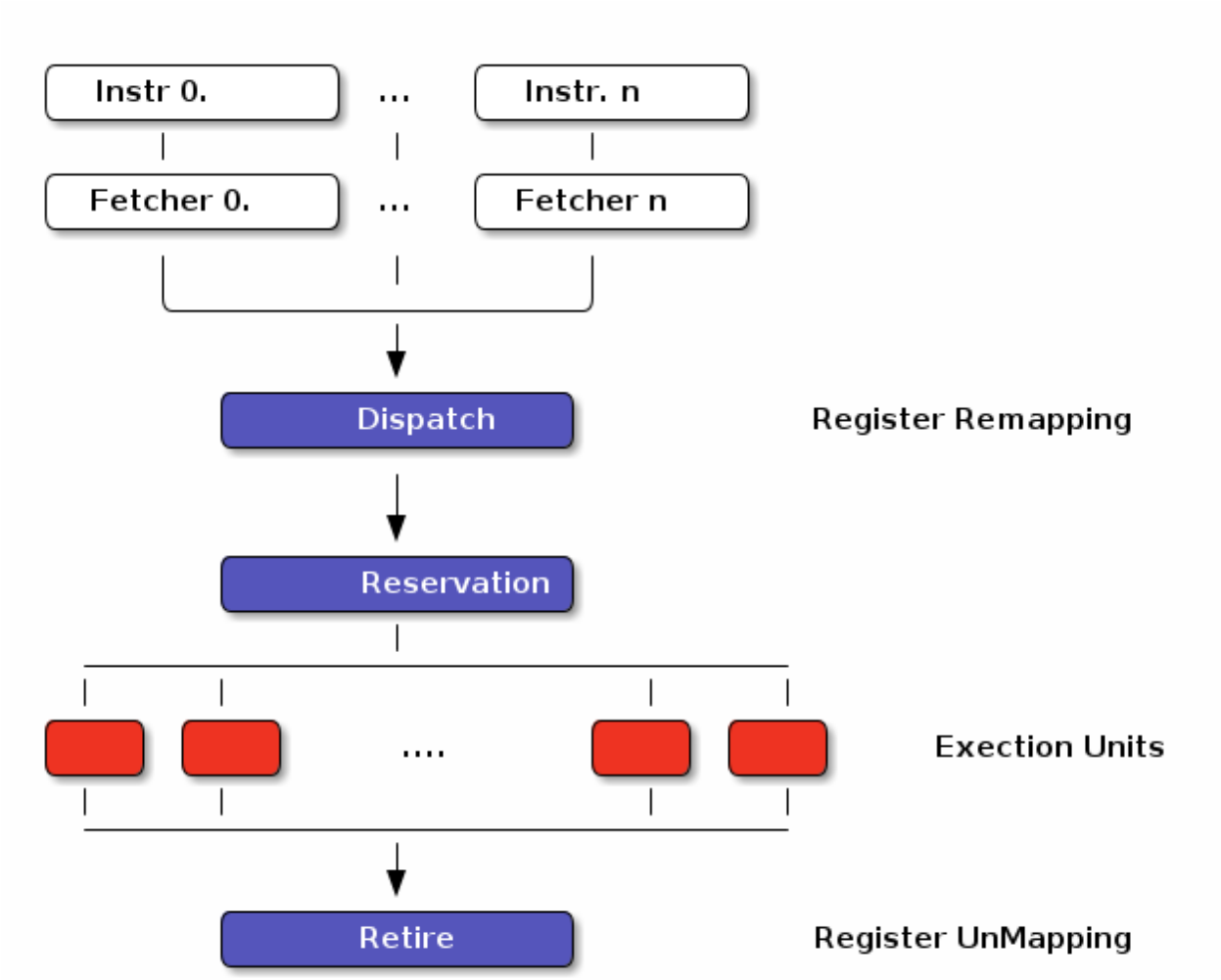


Figure 1.7: Out-Of-Order Execution Control Flow

# Chapter 2

# Current State of the Art and Notable/Relevant Works in Instruction Scheduling

## 2.1 List Scheduling (most commonly performed scheduling)

List scheduling is the most widely used technique for instruction scheduling [GM86]. List scheduling encompasses a class of algorithms for basic block scheduling via a chosen heuristic. It takes a list (of instructions), assigns priorities via a heuristic and schedules them in a topological order in decreasing priority [**wang2018list**]. The general structure of a list scheduling algorithm can be described as follows

Basic Structure of List Scheduling Algorithms

.

```
while there are instrs to be scheduled do
      Identify highest priority instr n
      Choose a processor p for n
      Schedule n on p at est(n,p)
end

est(n,p) = earliest start time of n on p
```

Priorities can be static (remain constant for the DAG) or dynamic (change through iteration of the algorithm). The complexity of list scheduling depends on the heuristic, but is generally polynomial time [**wang2018list**]. List scheduling can also be performed *forward* or *backward*, or performed successively, applying heuristic on top of heuristics. Examples of

common heuristics are given in Table 2.1 [**sarangal2018list**].

Table 2.1: Example List Scheduling Heuristics

| **Heuristic** | **Description** |
|---|---|
| HLFET (Highest level first with estimate time) | priority is chosen by the attributes of static levels |
| MCP (Modified Critical path) | priority by utilizing ALAP (As late as possible) attribute |
| ETF (Earliest time first) | finds the earliest start time for each task and later chooses the task having less initial time |
| DLS (Dynamic level scheduling) | finds the task priority on the tasks priority on dynamic basis |
| CNPT (Critical node parent tree) | prioritization of task is determined with CN (Critical node) attribute |

List scheduling is the chosen method for most conventional compilers because of its flexibility, efficiency and ability to find near-optimal schedules for most basic blocks. Although originally thought to yield near optimal schedules for almost all schedules, analysis of list scheduling techniques have shown that List Scheduling techniques have difficulty finding near-optimal schedules for codes with a moderate amount of available parallelism — the peak difficulty varying with both the number of processing elements and schedule length [**cooper1998experimental**]. Therefore as architectures become more sophisticated and provide more opportunity to exploit parallelism, list scheduling will in turn become increasingly inadequate.

Limitations of list scheduling most likely stem from the use of a single choice heuristic. There are many factors to consider when constructing a schedule, and it is difficult (or more accurately impossible) to condense this to assigning a priority via a polynomial time heuristic. As mentioned before, combinations of heuristics can be applied through successive iterations, but each iteration could undo previous iterations work.

## 2.2 Constraint Programming

As an inherently discrete problem, a number of works have sought to formulate instruction scheduling as an Integer Linear Programming (ILP) problem ([WLH00], [**chang1997using**], [**kastner2001ilp**], [**nagarakatte2007register**], [**kastner1999code**]). These models optimize over integer objective variables (representing schedule dispatch times for instructions), and utilize constraints and penalties to achieve valid and desirable schedules. Solving an ILP problem is an NP-Complete problem, so formulating scheduling as an ILP only really serves to generalize solutions to heuristics/approximation algorithms used by ILP solvers.

Another approach which has been far less investigated would be to optimize the problem

focusing entirely on constraints (known as constraint programming). This type of optimization model would find a desirable schedule through constraints alone (converting penalties). Notable efforts into this approach are the works of McInnes/Beek in [MMV08],[**van2001fast**]. The constraint problem in [MMV08] found provably optimal schedules for basic blocks , with the following types of constraints:

- **Latency Constraints**, i.e
  - Given a labeled dependency DAG $G = (N, E)$
    - $\forall (i, j) \in E \cdot j \geq i + l(i, j)$
- **Resource Constraints** that ensured functional units were not exceeded
- **Distance Constraints**, i.e
  - Given a labeled dependency **DAG** $G = (N, E)$
    - $\forall (i, j) \in E \cdot j \geq i + d(i, j)$

The approach has some limitations on sophisticated enough architectures. The hard constraints on latency would not account for **Register Remapping** in **Out Of Order Execution** architectures that would be able to find more optimal schedules despite the fact that latencies in normal execution would create **pipeline stalls**.

Assembly Code Example Requiring Register Renaming for Optimal Scheduling

```
fma r3,r3,r4
fma r2,r2,r4
fma r4,r0,r3
fma r0,r1,r2
```

On a system with only 5 registers and an instruction fma of large enough latency, the scheduler would need to push these instructions apart to avoid breaking the anti-dependency on register r4. However a machine could use register remapping to execute these instructions efficiently Out-of-Order making that constraint unnecessary.

Despite limitations introduced by using hard constraints, this work is notable as it illustrates how scheduling can be modeled as constraints. Even with the limitations on OoO architectures cutting of the optimal schedule, it's reasonable to assume the model would find near-optimal schedules. And by converting hard constraints to soft constraints (penalties) its simple to assert this space would contain the optimal solution.

## 2.3 Stochastic Search

There is a notable work to explore the space of schedules through a stochastic algorithm by Schkufza, Sharma, Aiken at Stanford [SSA16]. The efforts are culminated in a piece of open source software known as STOKE that serves as a "stochastic optimizer and program synthesizer" for x86-64 instruction sets. Stoke is a super-optimizer with the following notable qualities:

- ◇ Suitable for **Short Basic Block** assembly code sequences (no modulo scheduling)
- ◇ Utilizes a multi-pass algorithm, applying possibly overlapping program transformations each pass
- ◇ Encodes constraints as a **Cost Function** and uses a **Model Checker** to ensure valid schedules (undo-ing transformations otherwise)
- ◇ Uses **Markov Chain Monte Carlo Sampler** to add a stochastic element to program transformation

Each pass of the optimization minimizes the cost function

$$\text{cost}(R;T) = w_e \times \text{eq}(R;T) + w_p \times \text{perf}(R;T)$$

| | |
|---|---|
| $\boldsymbol{R}$ | any rewrite of the program |
| $\boldsymbol{T}$ | the input program sequence |
| $\text{eq}(\cdot)$ | the equivalence function (0 if $R \equiv T$ ) |
| $\text{perf}(\cdot)$ | a metric for performance |
| $\boldsymbol{w_e}$ | weight for the equivalence term |
| $\boldsymbol{w_p}$ | weight for the performance term |

Limitations with the approach (as outlined in [SSA16]) include

- ◇ Only optimizes basic blocks (no loops)
- ◇ Extremely innefficent (only practical for very short scheduling)
- ◇ Cost function doesn't model the space of valid checking (hence model checking is required per each rewrite)

This approach highlights the ability to use stochastic optimization to find near-optimal schedules and the use of MCMC to explore the space of schedules. Although the cost function doesn't model the space of valid schedules, using it to generate schedules along with a model checker may still prove to be the most practical way to explore a variety of schedules.

## 2.4 Meta-Optimization / Hyper-Heuristics

Meta-optimization deals with the use of one optimization method to tune another one. Hyper-heuristics are an off-spring of meta-optimization, that search within the search space of just heuristics vs the entire problem solution space. Hyper-heuristics are a relatively new concept (first used in 2000 to describe heuristics to choose heuristics) the definition has more recently been extended to refer to a search method or learning mechanism for selecting or generating heuristics to solve computational search problems [**burke2013hyper**]. When speaking of hyper-heuristics, two main categories should be considered: heuristic *selection* and heuristic *generation*.

Previous research into meta-optimization of compilers has been attempted, using machine learning to improve compiler heuristics [**stephenson2003meta**]. The use of genetic programming has rarely been used to solve production scheduling instances directly, because of the difficulty of encoding solutions. However it has been found highly suitable for encoding scheduling heuristics [**jakobovic2007genetic**].

In the work of [**jakobovic2007genetic**], genetic programming is used to evolve dispatching rules, which are functions that assign a score to a job based on the state of the problem. When a machine becomes idle, the dispatching rule function is evaluated once for each job in the machine's queue, and each result is assigned to the job as its score. The job in the queue with the highest score is the next job to be assigned to the machine. This approach shows genetic programming combining and rearranging heuristic components to create heuristics superior to those which have been created by humans (heuristic generation).

Although more works have investigated usage of heuristic generation for scheduling (particularly through genetic programming [**beaty1991genetic**],[**kri2004genetic**],[**stephenson2003genetic**] no works have been found (at the time of writing for this proposal) on using heuristic selection / generation to analysis the effectiveness of existing heuristics on scheduling in different architectures architectures. Some work has been performed using ML in the form of Support-vector machines (SVM) to learn compilation strategies (although not specific to scheduling) in JiT compilers for the Java Virtual Machine [**sanchez2011using**].

# Chapter 3

# Proposed Approaches To Stochastic Scheduling and Heuristic Analysis via Hyper-Heuristics

In this section I will introduce a Stochastic Scheduling Algorithm that utilizes a continuous optimization model with stochastically generated parameters, and propose it's use as a training set generator for ML driven heuristics selection and possibly generation. The full goal of this approach would be to not only find a means to evaluate the effectiveness of scheduling heuristics on a given architecture by observing learned heuristic selection, but to also be able to evaluate the limitations of an architecture by analyzing the space of valid schedules via the continuous model used for optimization.

## 3.1 Optimization Model for Modulo Scheduling

| | |
|---|---|
| Objective Variables $t_i, b_i, f_i$ : | $\mathbb{R}$ |
| Constants II : | $\mathbb{R}$ |
| Indicator Function $\mathbb{1}_{\mathbb{N}}$ : | $\mathbb{R} \to \mathbb{R}$ |
| $t_i$ : | dispatch time |
| $b_i$ : | completion time |
| $f_i$ : | FIFO use $0 \leq f_i \leq 1$ |
| II : | iteration interval $= \frac{\#instructions}{dispatches/cycle}$ |

$$\text{Hard Constraints} \qquad \forall i, j \cdot i \to j \qquad t_i + \epsilon \leq t_j \tag{3.1}$$

$$0 \leq t_i \leq b_i \leq \#\text{stages} \cdot \text{II} \tag{3.2}$$

$$b_i + \epsilon \leq t_i + \text{II} \tag{3.3}$$

$$\text{Objective Function} \qquad \min \sum_i (b_i - t_i + f_i) + \text{Penalties} \tag{3.4}$$

The above model is a continuous optimization problem for scheduling, similar to the work detailed in Constraint Programming by [MMV08], but only maintaining the constraints necessary for ensuring a valid schedule, and relaxing the problem into a continuous domain as opposed to the Integer Linear Programming models mentioned. The model optimizes over a set of objective variables: $t_i$,$b_i$,$f_i$ for each instruction $i$ to be scheduled. Each variable $t_i$,$b_i$ represents the dispatch and completion times of the $i^{th}$ instruction respectively. As we wish to model an Out-of-Order execution architecture, completion times are not necessarily fixed to dispatch times. The third variable, $f_i$, which is constrained to be between 0 and 1, is the probability that the instruction $i$ should spill. The constant variable II is the pre-computed Initiation Interval.

Unlike the work of [MMV08], the proposed model makes little use of constraints. In fact the only constraints used are to enforce the resulting schedule is a valid modulo schedule. The notation $i \to j$ used in the Hard Constraints equations above denotes that instruction $j$ depends on instruction $i$. The first hard constraint enforces that any instruction must be dispatched only after it's dependencies. The second constraint sets a limit on the overall execution time that instructions must fall within. And the third constraint enforces that an instruction must complete within the same interval it was dispatched to avoid an anti-dependency hazard.

Without any **Penalties**, the above objective model would squash all dispatch and completion times down, moving dependent instructions apart only by a given $\epsilon$ (a small constant) and assigning 0 to all $f_i$. A **Key Idea** to this work: encode choice heuristics as penalties, and adjust preference between heuristics by scaling. Heuristics, such as the ones detailed for use in List Scheduling, can all be encoded as Penalty Functions (functions that return a large value to push down in the schedule, or a large negative value to push up). By picking a large or small number to scale a penalty function we can prioritize a schedule.

## 3.2 IO Penalty

Since we're not pushing instructions apart through latency constraints like in [MMV08], we need a penalty to compensate. We propose penalizing the dispatch time of instructions based on the quantity and latencies of it's dependencies. **Note**: unlike a hard constraint on

latencies, this won't cut off valid schedules that could be optimal on OoO architectures.

$$\text{Given} \qquad t_i, t_j \qquad \forall i, j \mid i \to j \tag{3.5}$$

$$\text{For each i} \qquad N_j = \sum_{i \to j} \text{latency}(j) \tag{3.6}$$

$$\tag{3.7}$$

$$\mathbb{IO}(i) = \sum_j \frac{1}{N_j} \mathbb{IN}(t_i - t_j) \tag{3.8}$$

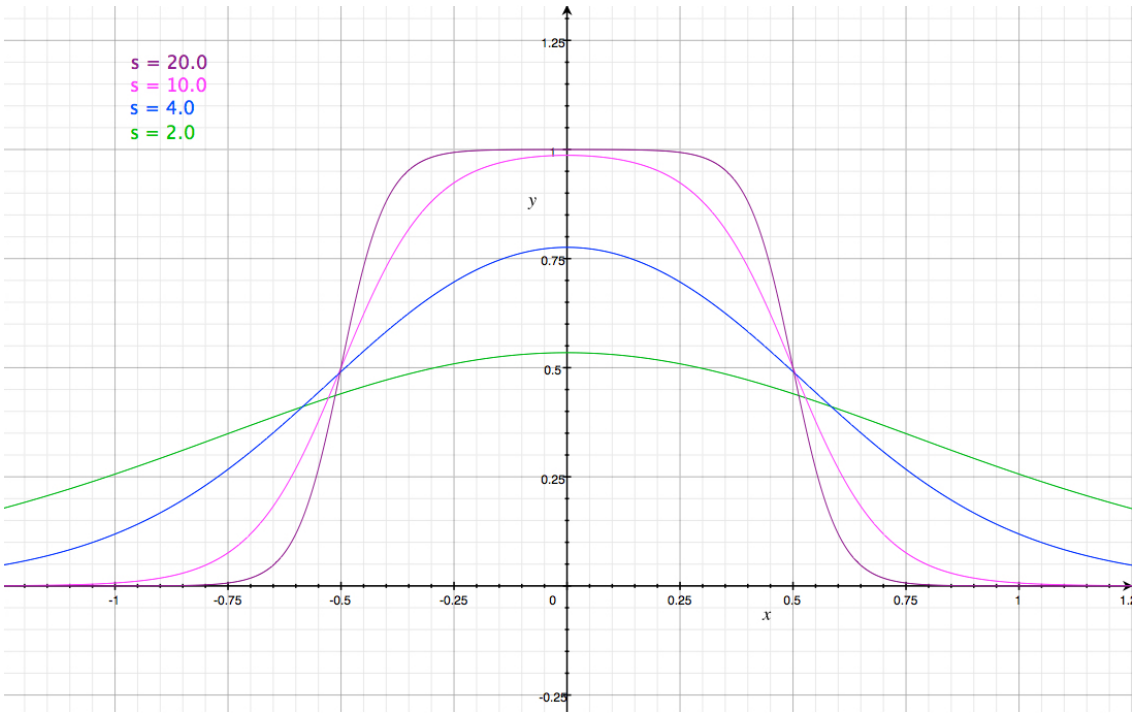


Figure 3.1: Custom Indicator Function

## 3.3 Indicator Function

The function $\mathbb{IN}$ specified in the model and used in the IO Penalty above is what's known as an Indicator Function, and is important for implementing heuristics that rely on testing the proximity of two instructions. Essentially, the function can be used to indicate (via a non-zero value) whether two instructions are scheduled within a certain range of each-other. For this purpose, we developed a custom indicator function (an alteration of the sigmoid function often used in neural networks) shown in Figure 3.1.

$$S(x) = \frac{1}{(1 + e^{s(-0.5+v)})(1 + e^{s(-0.5-v)})} \tag{3.9}$$

## 3.4 Stochastic Scaling

The scaling $\frac{1}{N_j}$ may be a good guess, but not necessarily effective in practice

- ◇ **IDEA** scale the **IO penalty** stochastically by a multiple of the II

$$\text{Define a Grouping} \qquad \mathbb{C} = \text{Group}(\forall i \mid i \to j) \tag{3.10}$$

$$\text{For each Group i} \qquad c_i \in \mathbb{RAND}(\mathbb{R}) \tag{3.11}$$

$$\text{Stochastic Penalty} \qquad \sum_i c_i II \cdot \mathbb{IO}(i) \tag{3.12}$$

As previously mentioned, various other heuristics can be encoded and also scaled stochastically, in basically the same manner as the IO penalty. The nature of many heuristics can be encapsulated by simply choosing the same scaling based on the type of instruction (i.e push all loads up or down).

## 3.5 Opportunity for Hyper-Heuristic Development

By representing the space of schedules in a continuous model (i.e the space of $\mathbb{R}^n$) and encoding heuristics as penalties, we can evaluate the merits of various heuristics in combination with each other. The continuous nature of the model provides more degrees of freedom when evaluating overlapping heuristics. By scaling these heuristics stochastically, we already have a method to analyze heuristic selection through analyzing which scalings provide better performing schedules.

By generating a variety of schedules for different types of basic blocks using the stochastic method we can also obtain a "training set" which can subsequently be used with various supervised Machine Learning (ML) algorithms, most notably Support Vector Machines (SVM), supervised learning models that analyze data used for classification and regression analysis. The use of neural nets, ensemble methods or genetic algorithms may also be worth exploring.

Principal Component Analysis is a dimension reduction tool that can be used to reduce a large set of variables to a small set that still contains most of the information in the large set. Principal component analysis can possibly be performed over the scaling parameters in conjunction with the training set results in order to judge the influence of penalties on a given architecture.

Clustering methods (unsupervised learning) can possibly be used for heuristic generation, by finding new groupings to scale. Given a clustering with scaling $c_i$ we can check the following assertion: For each scaling $c_i \in \mathbb{RAND}(\mathbb{R})$, there exists an $\epsilon \in (R)$ such that $c_i + \epsilon$ produces

a distinct schedule from $c_i$. If the assertion fails, the clustering is useless. A prominent research question would be for this work would be whether or not it's possible to avoid such clusterings.

# Bibliography


[BR91] David Bernstein and Michael Rodeh. "Global instruction scheduling for superscalar machines". In: *ACM SIGPLAN Notices*. Vol. 26. 6. ACM. 1991, pp. 241–255.

[Bre13] Glen E Bredon. *Topology and geometry*. Vol. 139. Springer Science & Business Media, 2013.

[Cha82] G. J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *SIGPLAN Not.* 17.6 (June 1982), pp. 98–101.

[Cos] Kriston Costa. "Approximation Algorithm based Approach Instruction Scheduling". In: ().

[Fis81] Joseph A. Fisher. "Trace scheduling: A technique for global microcode compaction". In: *IEEE transactions on computers* 7 (1981), pp. 478–490.

[Fis83] Joseph A Fisher. *Very long instruction word architectures and the ELI-512*. Vol. 11. 3. ACM, 1983.

[GM86] Philip B Gibbons and Steven S Muchnick. "Efficient instruction scheduling for a pipelined architecture". In: *Acm sigplan notices*. Vol. 21. 7. ACM. 1986, pp. 11–16.

[HG83] John Hennessy and Thomas Gross. "Postpass code optimization of pipeline constraints". In: *ACM Trans. Program. Lang. Syst.;(United States)* 3 (1983).

[Hwu+93] Wen-Mei W Hwu et al. "The superblock: an effective technique for VLIW and superscalar compilation". In: *Instruction-Level Parallelism*. Springer, 1993, pp. 229–248.

[Lam12] Monica S Lam. *A systolic array optimizing compiler*. Vol. 64. Springer Science & Business Media, 2012.

[MMV08] Abid M Malik, Jim McInnes, and Peter Van Beek. "Optimal basic block instruction scheduling for multiple-issue processors using constraint programming". In: *International Journal on Artificial Intelligence Tools* 17.01 (2008), pp. 37–54.

[Mot+95] Rajeev Motwani et al. "Combining register allocation and instruction scheduling". In: *Courant Institute, New York University* (1995).

[Pin93] Shlomit S. Pinter. "Register Allocation with Instruction Scheduling". In: *SIGPLAN Not.* 28.6 (June 1993), pp. 248–257.

[RF93] B Ramakrishna Rau and Joseph A Fisher. "Instruction-level parallel processing: history, overview, and perspective". In: *Instruction-Level Parallelism*. Springer, 1993, pp. 9–50.

[SSA16] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Program Optimization". In: *Commun. ACM* 59.2 (Jan. 2016), pp. 114–122.

[Tom67] Robert M Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[TSD01] Gary S Tyson, Mikhail Smelyanskiy, and Edward S Davidson. "Evaluating the use of register queues in software pipelined loops". In: *IEEE Transactions on Computers* 50.8 (2001), pp. 769–783.

[WLH00] Kent Wilken, Jack Liu, and Mark Heffernan. "Optimal instruction scheduling using integer programming". In: *Acm sigplan notices*. Vol. 35. 5. ACM. 2000, pp. 121–133.

[ZK01] Victor V Zyuban and Peter M Kogge. "Inherently lower-power high-performance superscalar architectures". In: *IEEE Transactions on Computers* 50.3 (2001), pp. 268–285.