An Approximation Algorithm Based Approach to Instruction Scheduling

AN APPROXIMATION ALGORITHM BASED APPROACH TO INSTRUCTION SCHEDULING

BY
KRISTON COSTA, B.Sc.

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Master of Science (2016)
(Computer Science)

McMaster University
Hamilton, Ontario, Canada

TITLE:	An Approximation Algorithm Based Approach to Instruction Scheduling
AUTHOR:	Kriston Costa B.Sc., (Integrated Science) McMaster University, Hamilton, Canada
SUPERVISOR:	Dr. Christopher Anand & Dr. Wolfram Kahl
NUMBER OF PAGES:	ix, 43

Abstract

Instruction scheduling is an NP-complete problem which allows for deciding the execution order of instructions in a function without altering the function's semantics. By modifying instruction execution order, CPU resource usage can be maximized resulting in increased instruction throughput and decreased overall execution times. In this thesis, a novel approximation-algorithm-based scheduler is introduced and a prototype is implemented for the IBM® z13™ processor. The algorithm is a modified version of Karger's minimum cut algorithm which is applied to a directed acyclic graph that defines instruction dependencies, also called the codegraph. Rather than finding the minimum cut of a codegraph, the algorithm produces multiple large groups of instructions that are independently scheduled and dispatched in software pipelined stages. Each iteration of this approximation algorithm produces a random feasible schedule. The likelihood that the performance of a schedule falls within a given range of the optimal solution improves as the total number of schedules produced increases. This allows for an explicit trade-off between schedule performance and scheduling time. The prototype was tested by scheduling functions from the IBM Mathematical Acceleration Subsystem (MASS) libraries for the z13 processor, and was found to be capable of producing production-quality schedules with minimal user involvement.

Acknowledgements

To begin, I would like to thank Dr. Christopher Anand for all his support and providing me guidance with both my thesis and my transition into the field of computer science. I wouid also like to thank Dr. Wolfram Kahl for his counsel and constructive feedback that was provided throughout my thesis. I am grateful to Robert Enenkel for all of his work in testing and timing the initial schedules that were produced.

Thank you to my parents for their consistent motivation and support throughout my life and education. Last, but not least, thank you to my friends and family for their encouragement throughout the past two years. My special thanks are extended to Emily Kramer, Darren Fernandes, and Yohan Yee for always being there if I needed someone to bounce ideas off of and their insightful questions.

Contents

List of Figures

Chapter 1

Introduction

NP-complete problems (NPCPs) have solutions which are easily verified, but are difficult to produce (Cormen *et al.*, 2009). Current methods of creating exact solutions to NPCPs require an impractically long amount of time. There are two methods which can reasonably solve NPCPs in polynomial time: heuristics, and approximation algorithms (Cormen *et al.*, 2009). Heuristics are algorithms which produce reasonable results in polynomial time, but may not always be fast and effective (Cormen *et al.*, 2009). Approximation algorithms search for approximately optimal solutions for NPCPs in polynomial time (Cormen *et al.*, 2009). An example of a NPCP is instruction scheduling (Hennessy and Gross, 1983). In this thesis, we will propose an approximation algorithm approach to instruction scheduling.

1.1 Thesis Organization

Chapter 1 will be devoted to giving the motivation behind the project and the basic background to hardware pipelining and the challenges surrounding it. Chapter 2 contains a basic introduction to scheduling algorithms and approximation algorithms in addition to the scheduling algorithm proposed by this thesis. Chapter 3 presents the numerical evaluation of the algorithm. Chapter 4 provides a final discussion.

1.2 Motivation

Mathematical functions have a high frequency of usage in a variety of different programs. This means that small improvements in function run time have the potential to significantly decrease the total runtime of a a given program. The IBM Mathematical Acceleration Subsystem (MASS) is a library of such math functions which is supplied with IBM XL compilers (IBM, 2015a)(IBM, 2015b)(IBM, 2015c). During

the development and release of IBM's z13 architecture a method of producing tuned z13 assembly versions of functions from MASS was investigated (IBM, 2015d). The z13 processor is an evolution of the IBM mainframe superscalar, out-of-order system architecture, which includes SIMD instructions (Lascu, 2015). Due to the complexity of MASS, a robust scheduler was required to produce schedules that were optimized for both resource usage and performance. Simple scheduling algorithms produced invalid schedules due to the complexity of both the functions and the complexity of the resource constraints. A scheduler was produced that allows for performance-optimized schedules to be created with the flexibility to be applied to complex architectures.

1.3 Instruction Scheduling

Instruction scheduling is the process of organizing the execution order of a set of instructions in order to optimize the performance of a function with complicated resource constraints (Hennessy and Gross, 1983). In particular, instruction scheduling allows for instruction-level parallelism to be exploited in order to improve performance on architectures which allow for out-of-order execution. To understand the reasoning behind this, hardware pipelining and the hazards associated with it must be understood.

1.3.1 Hazards

Hazards occur when the execution of an instruction is either not possible, or alters the result of the original source code (Page, 2009). For example, if two instructions depend on the same data value but one of these instructions overwrite the original data then the order of execution is important to prevent the data from being overwritten before it is utilized by both instructions. There are three primary hazards which must be considered when scheduling instructions (Page, 2009).

- Data Hazard — data is overwritten, or not assigned, before it is accessed by the instruction which depends upon it.
- Resource Hazard — the resources required by an instruction are not available due to usage by other instructions. Examples that show how this hazard can manifest include overloading functional units, and attempting to allocate more registers than the number available on the machine.
- Control Hazard — when a branch is not resolved before an instruction which depends on it.

1.3.2 Hardware Pipelining

Hardware pipelining is the process of executing instructions, or groups of instructions, in a concurrent manner. There are two methods of pipelining—software pipelining and hardware pipelining. Hardware pipelining involves dispatching an instruction for execution before the previous instruction has finished completing (Hennessy and Patterson, 2011). This is accomplished by allowing for different stages of instruction execution to be executed independently of one another. If an architecture supports hardware pipelining, the length of each cycle is shorter compared to non-pipelined hardware, however, it takes multiple cycles to fully execute an instruction (Page, 2009). Hardware pipelining maximizes the usage of the available hardware which results in an increased instruction throughput (Page, 2009).

Software pipelining is a method of hiding the latency of instruction execution by inserting non-dependent instructions in dependent instruction chains (Page, 2009). The scheduled code is broken into stages. The stages are then interleaved in such a way that high latency instructions and their dependent instructions are separated by other, independent instructions. This process improves throughput by preventing data hazards from stalling the dispatch of dependent instructions (Page, 2009). However, by separating dependent instructions the lifetimes of the associated registers is extended, so this process will result in an increased number of registers used (*register pressure*).

1	F	D	T	X	WB				
2		F	D	T	X	WB			
3			F	D	T	X	WB		
4				F	D	T	X	WB	
5					F	D	T	X	WB
# Instrs. / Iteration	1	2	3	4	5	6	7	8	9

Figure 1.1: Hardware Pipelining: An example of how hardware pipelining allows for resources to be maximized. The above chart shows how instructions propagate through a CPU. Each row is a new instruction, and the columns show the cycle number. Cycle 5 illustrates when the hardware is saturated. The above chart is based on the Intel 486 CPU. The order of instruction issuing is Fetch (F), Decode (D), Translate (T), Execute (X), Write Back (WB) (Johnson, 1990).

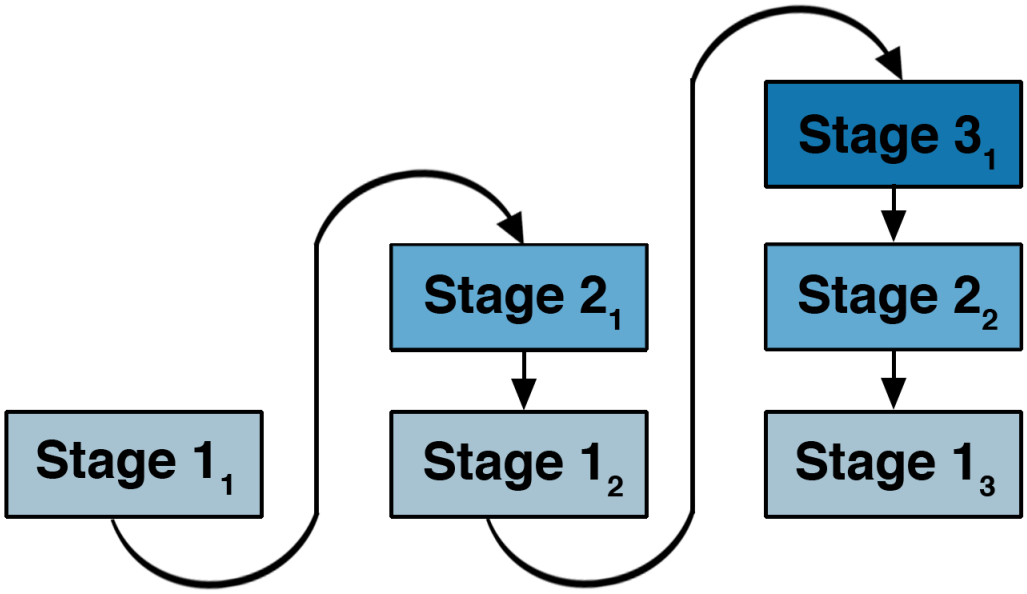


Figure 1.2: Software Pipelining: An example of software pipelining. In order to hide instruction latency stages are not executed in order. In the example above three stages are pipelined. Once the loop is saturated it results in each stage allowing for an additional stage to be called before the dependent stage. For example, stage one will feed into stage two, however the latency of the instructions is hidden by the execution of stage three. The superscript indicates the iteration number that the stage is currently executing.

1.3.3 Out-of-order Execution

Out-of-order execution depends on the idea of register renaming. A processor has *architectural registers which are* able to be accessed by a developer, and *physical registers* which are accessed by the hardware. *Register renaming* renames an architectural register to a physical register. There are always more physical registers than architectural registers, and it is possible to have an architectural register map to multiple different physical registers. Out-of-order execution allows for independent instructions issued to a processor to be executed once all data operands are available, not specifically in the order that they are issued. When an instruction is issued, it's operand architectural registers are renamed to hardware registers, which either contain the dependent data or placeholder values. The placeholder values are later overwritten by the data dependent value, and if all operands are available the

instruction is executed. This allows for instructions which are data independent but register dependent to be executed in parallel. If out-of-order execution is not permitted then independent instructions which access the same registers become serialized, and bottlenecks that affect one of those instructions will affect all subsequent instructions. Register renaming removes these false dependencies and allows schedules to ignore potential register dependencies causing serialization (Smith and Pleszkun, 1988). This greatly simplifies the register allocation step described in 2.3.3.

1.3.4 Motivation for Instruction Scheduling

Instruction scheduling is used in order to maximize CPU resource usage and throughput while preventing hazards from occurring. In order to maximize hardware pipelining, adjacent executed instructions should depend on different hardware resources, or be low latency instructions. Selecting appropriate groups for software pipelining is required to ensure that high latency instructions are being appropriately hidden by the execution of other stages. Care must be taken to prevent the hazard presented above. In particular, if the throughput relies too heavily on widely spaced instruction dependencies, register allocation may not be realizable if the machine does not have enough resources available. The complexity of this problem prevents an optimal solution from being determined in polynomial time, and is considered NP-Complete. It is possible to find near-optimal solutions in polynomial time.

1.3.5 Current Techniques

Heuristic implementations allow for solutions to be produced in polynomial time. Typically, the metric used to gauge the optimality of a solution is the throughput of the resulting schedule or the maximum register pressure of a viable schedule. Initial schedulers focused on improving the throughput through the use of list scheduling and critical path algorithms.

List Scheduling

List scheduling involves producing a list of potential instructions that could be inserted into a schedule. Instructions are schedulable if their preceding data dependent instructions have been scheduled. The listed instructions are then compared to one another to determine which instruction will not interlock with the previously scheduled instruction, and which instruction has the lowest probability of causing an interlock. This method is used to consider the priority of the instructions in a list. If the hardware required for the instruction's execution is currently saturated then the next highest priority instruction is considered. Once an instruction is selected,

additional instructions are added to the list of candidates if all their dependencies are already scheduled (Hennessy and Gross, 1983).

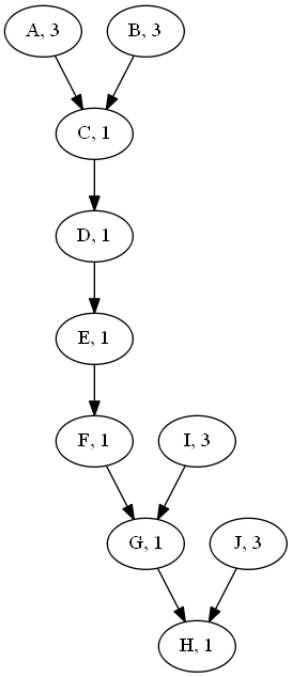
Improved scheduling algorithms based on the list scheduler utilized more robust methods of determining instruction priority (Gibbons and Muchnick, 1986). The most commonly used of these methods is the critical path algorithm.

Critical Path Algorithm

The critical path algorithm requires that all instructions have a total execution time. A back-flow algorithm is used to propagate the execution times backwards through the dependent instructions. The resulting graph contains the total length of time required to execute a particular path of the original graph and allows for critical long execution time paths to be discovered. The list scheduling algorithm can then be applied to this graph, selecting the instructions that have the highest backward propagation value (Landskov *et al.*, 1980) (Gonzalez, 1977).

List scheduling with the critical path algorithm is a greedy method of instruction scheduling. Due to little look-ahead, the scheduler will not choose a less optimal local path to produce a optimal global result. An example of a non-optimal schedule produced by the critical path algorithm is shown in Figure 1.3.

In addition to critical path there are other algorithms used in combination with list schedulers, including decisive path, smallest co-levels first, and high level first algorithms (Adam *et al.*, 1974) (Park *et al.*, 1997). These algorithms would allow for functions to be scheduled near optimally with regards to throughput. Register pressure is typically not considered in these schedulers.



(a) Example DAG

Instr. Num.	Critical	Optimal
1	A	A
2	B	B
3	NoOp	I
4	NoOp	J
5	C	C
6	D	D
7	I	E
8	J	F
9	E	G
10	F	H
11	G	
12	H	

(b) Possible Schedules

Figure 1.3: The figure to the left is an example of a DAG that does not produce an optimal result when scheduled using the critical path algorithm. The nodes are defined with a label and number of cycles for completion (nodes, cycles). A combination of critical path algorithms and hazard analysis would result in an optimal schedule.

1.3.6 Branch and Bound

An example of an algorithm that allows for optimal schedules to be produced is the branch and bound algorithm. By producing a tree of all possible schedules and evaluating each for optimality it is possible to find the optimal schedule, however the runtime is non-polynomial and therefore not suitable for general use (Clausen, 1997).

There are examples of schedulers that allow for register-sensitive scheduling, but they typically do not include software pipelining. A notable example of a scheduler which provides both register-sensitive scheduling with software pipelining is the scheduler proposed by Llosa et al, swing modulo scheduling (Llosa *et al.*, 1996).

1.3.7 Swing Modulo Scheduling

Swing modulo scheduling is a two phase scheduling method which attempts to minimize stage size and register lifetime. Utilizing a directed acyclic graph to represent the instructions to be scheduled each node is visited and added to a list to produce a

scheduling. To produce this list, the sinks of the DAG are added to the list based on their depth; the "deepest" sinks are added first. Predecessors of the nodes in the list are then visited in order of their height until there are no additional predecessors. The graph then starts from the source nodes and iterates through unvisited successors, in order of increasing depth. This initial list can be seen as the critical path of the DAG. The list is then iterated through, and the instructions are inserted into the schedule based on the ordering in the list. The position of an instruction in the schedule is based on the instructions predecessors or successors. If there are only successors in the schedule then the instruction will be scheduled as late as possible, if there are only predecessors it will be scheduled as early as possible. This minimizes the register lifetimes; however if there are both predecessors and successors present in the schedule, then inserting the instruction in the earliest or latest slot will result in extended register lifetime. Due to the initial ordering there will only ever be one instruction which will have both predecessors and successors present in the schedule at the time the instruction itself is being scheduled, preventing ambiguous placement issues with all other instructions. The placement of instructions limits slots available for future instructions to be inserted, increasing constraints over time and optimality. If there are no additional slots available to insert an instruction as the algorithm progresses, then the schedule is restarted with an increased stage size. The final schedule can then be split into stages (Llosa *et al.*, 1996).

Swing modulo scheduling is the current standard in software-based pipelined scheduling. Due to the rigid structure of the algorithm it is still not possible to reduce local optimality to improve the global solution. The approximation algorithm based scheduler was created with hopes that it will allow for these local minima to be overcome and increase the optimality of the final solution.

1.3.8 Principled Graph Transformations in Coconut

In pervious work at McMaster special functions were scheduled using principled code-graph transformations (Anand and Kahl, 2009). In particular, the mincut problem was used in explicitly staged software pipelining, a scheduling algorithm described in (Thaller, 2006).

Chapter 2

Approximation Algorithm Instruction Scheduling

2.1 Approximation Algorithms

Approximation algorithms take many forms, with perhaps the simplest being randomized enumeration or tree search. The local optimal solution from this solution set will have a probability of being the optimal (or near-optimal) solution. One of the most well-known examples of an application of approximation algorithms is Karger's solution to the minimum cut problem for a graph. Determining the minimum cut of a graph is an NPC problem, however Karger's algorithm allows for polynomial runtime with a lower bound on the probability of success of finding an optimal solution (Karger and Stein, 1996).

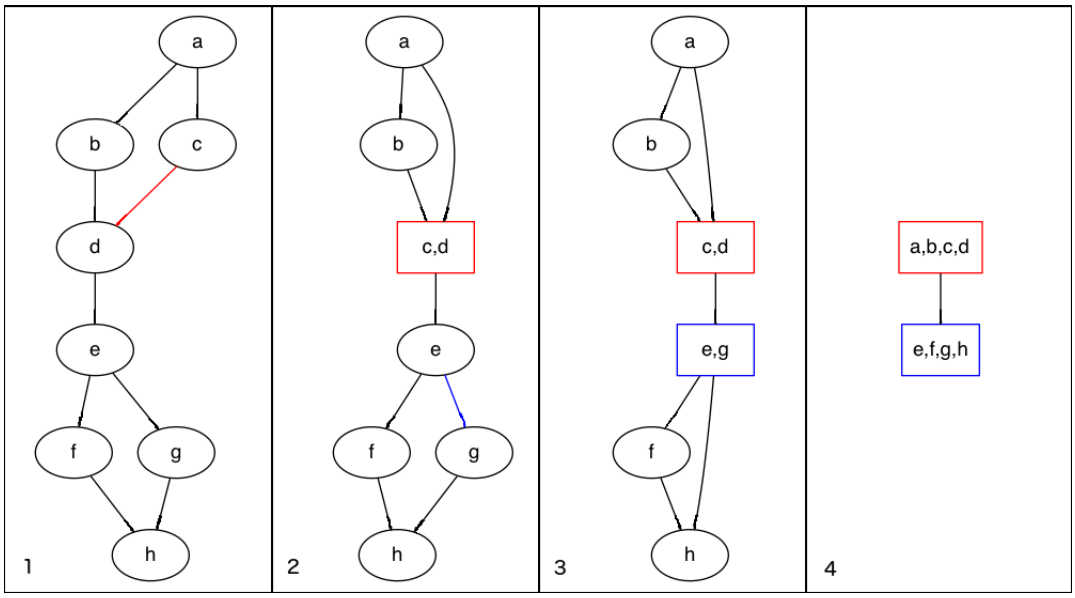


Figure 2.1: Example of Karger's Algorithm: Sections 1 to 3 show successive iterations, where colored edges are edges that are to be contracted and colored nodes are the resulting groups. Section 4 skips ahead a few iterations to the final step where there are two nodes remaining. The example above shows the optimal solution for the minimum cut, but Kargers algorithm will also find non-optimal solutions. This iterative process must be completed multiple times before the optimal solution can be guaranteed with some certainty.

2.1.1 Karger's Minimum Cut Algorithm

One iteration of Karger's algorithm works as follows:

1. Begin with an undirected graph $G = (V, E)$ with vertices V and edges E, where: $\{V_1, V_2\}$ are two independent nodes.
2. Choose a random node a and an adjacent node b
   - Replace nodes a and b with node ab
   - Replace edges $\{c, a\}$ or $\{c, b\}$ with $\{c, ab\}$, where c is any other node
   - Replace edges $\{a, c\}$ or $\{b, c\}$ with $\{ab, c\}$
3. If there are more than two elements in V then go to 2
4. Count the number of remaining edges E connecting the remaining two nodes.

The heuristic above is applied $n-1$ times; otherwise the graph would have fewer than 2 vertices remaining, and a cut graph is produced. With n nodes the probability that a given cut is the minimum cut is $\binom{n}{2}^{-1}$. If this procedure is performed $\binom{n}{2} \ln n$ times and the minimum cut is chosen from that subset, then the probability that the cut is the global solution is $\frac{n-1}{n}$.

2.1.2 Application to Scheduling

As Karger's algorithm progresses the graph is compressed into a number of subgroups and the number of external edges connecting the subgroups tends to be minimized. It was hypothesized that if the algorithm was terminated before only 2 vertices remain and the edges are weighted in a way that produces groups containing instructions that would benefit fro being dispatched together, then the subgroups generated could be used to produce the stages used during software pipelining.

2.2 Modification to Karger's Algorithm

If the instructions in a codegraph are scheduled in two groups according to the min-cut, the number of registers in use across the cut is the size of the cut, so the min-cut minimizes the register pressure at that point. For partially grouped codegraphs, the number of edges in the grouped graph, with grouped instructions scheduled together, the remaining edges represent register values used across groups. We will call this *external register pressure*. The minimization and grouping properties of this algorithm seem to make it a good fit for this scheduling problem.

Due to the nature of the algorithm, if a given vertex A has three edges feeding into vertex B and one into vertex C, the next iteration is more likely to group A and B together. If A and B are treated as stages it can be seen how, with each successive iteration, subgroups that have more connected edges will merge together, reducing external register pressure. By simply terminating the algorithm once n number of subgroups have formed it is possible to generate any number of stages. There are three properties of this algorithm that need to be corrected:

1. The algorithm will tend to produce one very large group with many small groups connected to the main large group by one or two edges.
    - To prevent this from occurring a maximum group size is set, where if the product of two groups merging results in a group that is larger than this size the algorithm prevents the merge.
2. Not all contractions are equally desirable.

- Weightings are not all equal, instructions which favor being dispatched together have connecting edges with higher weighting. This provides a bias to favour combining these instructions, however it does not prevent the algorithm from selecting instructions that may generate a non-optimal local solution which may improve the global solution.

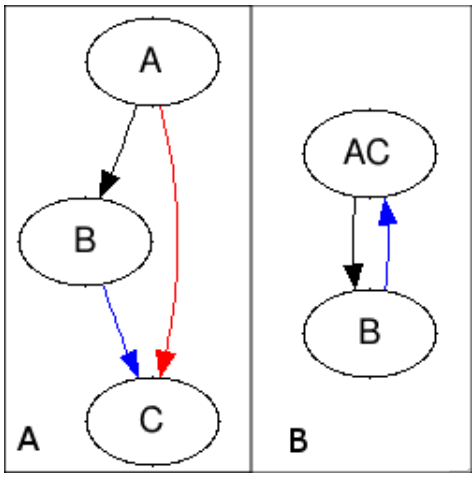


Figure 2.2: Possible Cycles When Producing Min-Cut: Part A shows the initial graph. If the edge connecting A and C is contracted and node AC is created then a cycle will be produced as seen in part B.

3. The algorithm could produce cyclic dependency graphs.
    - A dependency graph is produced at initialization of the algorithm. Vertices may only merge if there are no intermediate vertices that may join other groups. See Figure 2.2.

2.2.1 Modified Karger's Algorithm

The newly produced algorithm is as follows:

1. Begin with an directed graph $G = (V, E)$ with V vertices and E edges, where (v_1, v_2) is the edge with source v_1 and sink v_2.
2. Produce a $w : E \to \mathbb{N}$ function which returns the weighting of a given edge.
3. Choose a random vertex a:
    - Expand the edges connecting to a, duplicate an edge (a, c) or (c, a), where $c \epsilon V$, $w(a, c)$ or $w(c, a)$ times and add it to the multiset E_{temp}.
    - If there exists no edge, or the value of

$$\sum_{c \epsilon V} w(a, c) = 0$$

then return to 3.

- Select a random edge $(a, b), (a, b) \epsilon E_{temp}$.
- Replace nodes a and b with node ab.
- Replace edges (c, a) or (c, b) with (c, ab), where c is any other node.
- Replace edges (a, c) or (b, c) with (ab, c).
- Determine invalidated edges, and set their weight to 0.

4. If there exist any merged verticies which have not exceeded the maximum group size, or less than n iterations have been performed, go to 2.

5. Return the merged verticies and their elements (henceforth known as supergroups).

The supergroups returned by the modified algorithm will be used as the building blocks for software stages. There will be two stages of scheduling, internal and external. Internal scheduling will involve scheduling instructions contained within a supergroup; instructions are unable to move between supergroups. External scheduling will involve scheduling supergroups with respect to each other.

2.2.2 Size of Supergroups

The following should be considered when determining the maximum size of a supergroup. If the size of the supergroups is limited to 10 instructions it is possible to use the branch and bound algorithm to find the optimal internal schedule of the supergroup. In this case each stage would be made of a subset of the set of supergroups.

- Constant runtime for internal scheduling, due to small number of instructions.
- Internal schedule can be optimized for both grouping and minimized internal register pressure.
- Grouped instructions should also result in near-optimal throughput of each supergroup.
- The external register pressure will not be optimal, if there are a large number of supergroups then the external register pressure will be inflated.

It is also possible to set the maximum size of the supergroup to be approximately $\frac{n}{m}$ where m is the desired number of stages and n is the total number of instructions. In this case, each supergroup would represent a stage.

- The minimum external register pressure will be closer to optimal.
- The internal scheduling problem will take a prohibitive length of time; either a list scheduler or another heuristic scheduling algorithm should be used that can be run in polynomial time.

For functions with less than 300 instructions, testing showed that limiting the size of the supergroups to be 10 instructions allowed for additional instruction throughput with minimal increases in external register pressure.

2.2.3 Grouping Supergroups into Stages

The supergroups are grouped together into stages. In order to perform this grouping the same modified Karger's algorithm is applied to a directed acyclic graph (DAG) that represents the connected supergroups. The weighting of each edge is set to 1, the internal structure of each supergroup is not modified, only the external register pressure needs to be minimized. The maximum size of the merged vertices is set to be $\frac{\text{numberOfSupergroups}}{\text{numberOfStages}} + 1$ The resulting groups represent each stage and their associated supergroups.

The internal schedule of each supergroup is determined according to the methods described in 2.2. An estimate of the maximum internal register pressure is determined by detecting the maximum number of live edges at any given time.

If the number of supergroups per stage is less than 11, the topological sorts of each stage are enumerated to find all possible orderings of supergroups. If the number of supergroups is larger than 11, then the number of topological sorts is limited. Topological sorts are generated at random where the number of sorts generated is proportional to the number of super groups. The optimal sorting is the sort with minimized maximum total register pressure, where the total register pressure is the sum of the internal and external pressure of any given supergroup.

For scheduling MASS it was decided to have three stages, because this allowed for latency hiding in a representative set of test functions, and simplified the identification of data hazards described in 2.3.

2.3 Additional Schedule Modifications

2.3.1 FIFO insertion

Data hazards can occur for any pipelined schedules that have more than 2 stages. If the data generated by one stage is not used in the next loop iteration by the next stage then the data will be overwritten. Referring to Figure 1.2, it can be seen that if

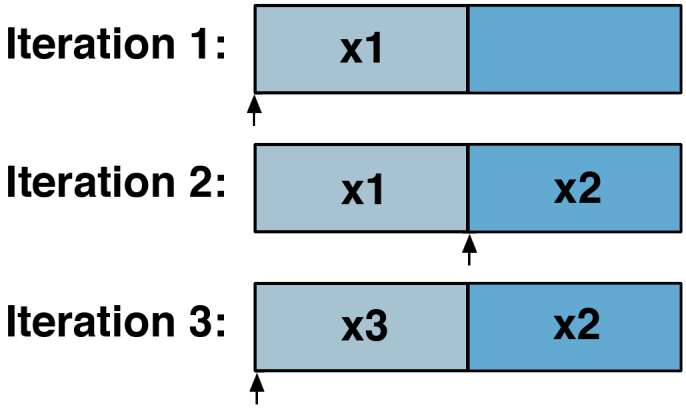


Figure 2.3: Basic FIFO Structure: With each iteration of the loop the data is pushed down the FIFO. When the data dependant stage will always access the end of the FIFO, where the appropriate iteration data is located. In the case of Figure 1.2 the first iteration of just stage 1 will write x1. The next iteration with stage 2 and stage 1 will write x2. Then in the last iteration stage 3 will access x1, then stage 1 will write x3. x1 is overwritten in the FIFO. The arrow indicates a pointer that is alternating between the two addresses with each iteration.

stage 3 depends on stage 1 there are two iterations of stage 1 before stage 3 is called, meaning the data supplied to stage 3 will be from a newer loop iteration than expected. Each iteration of stage 1 could write to different registers to prevent overwriting data, however this comes at a cost of significantly increased register pressure. Instead, it is possible to store the value in a FIFO rotating data structure in order to prevent data hazards.

The size of the FIFO is the length of the stage separation. If there is an edge from stage 1 to stage 3 the FIFO will only ever need to store two values. For stage 1 to stage 4 three values will be required, and so on. FIFOs are simplified if there are only three stages, there can only ever be edges of length 2 that will require a FIFO. An XOR operation on the FIFO pointer is all that will be required to switch between the two stored values. If there are more than three stages the modulo arithmetic to calculate the FIFO pointer will change depending on the length of the stage separation of each edge.

The z13 processor does not necessarily benefit from a large number of stages. Based on this and the simplified pointer operations it was decided that 3 stages would be used for scheduling.

The FIFO load and store instructions are inserted into the schedule by inserting

load operations at the beginning of the dependent supergroups internal schedule and the store instructions at the end. In the event that multiple supergroups depend on the same FIFO then the FIFO is loaded in the stage's first supergroup.

2.3.2 Interleaving Stages

Many MASS functions contain table lookups. Large tables need to be stored in memory, and accessed via loads with computed indices. Some tables are large enough not to fit in level-one cache, and have significantly higher latency when compared to the non-lookup instructions. If the schedule attempts to execute instructions that are dependent on the lookup instructions immediately following the lookup, throughput will be reduced as the lookup is performed. If instructions are inserted between the lookups and the instructions that depend on them the latency can be hidden. However, the grouping algorithm will tend to group the lookup instructions together in the same supergroups and stages.

In order to improve throughput, the stages are interleaved together. During the initial creation of the supergroups, a bias is set to ensure that lookup loads and the instructions that depend on them are in separate supergroups. Instead of dispatching the stages in reverse order, the supergroups of different stages are interleaved in such a way that lookup loads will always have supergroups from other stages called before the dependent lookup instructions are called.

The supergroups are topologically sorted, meaning the interleaving is trivial; however additional data hazards occur due to this new ordering. It is now possible to have negative stage separation, such as instances where a supergroup from stage n is called before a dependent supergroup from stage $n+1$. The stage n supergroup will overwrite its previous value before the following supergroup can access the value. For example, if there is an ordering where a supergroup from stage one is executed before stage two in the same iteration, then the length of separation is now negative one and stage one will always overwrite its previous value before stage two can access it, as seen in2.4.

To prevent this, a FIFO will be required to store the value. The XOR pointer arithmetic will not work in this case, the length of the edge is only one. The pointer is updated at the beginning of each loop, and the FIFO store is called before the FIFO load, meaning that the required FIFO value will always be overwritten before it is loaded. There are two ways to bypass this issue:

- Instead of loading the value in stage $n+1$, load the value in the last supergroup of stage n.
  - Requires no additional instructions or FIFO pointer updates.

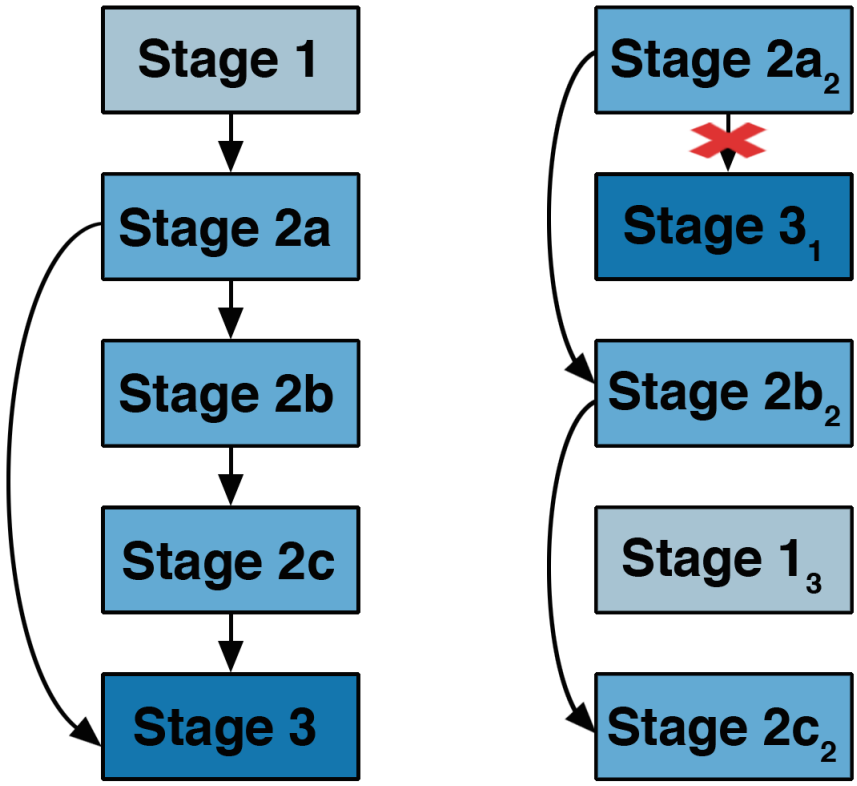


Figure 2.4: Data hazards when interleaving stages: The figure on the left shows the original topological sorting of a set of stages, with arrows indicating data dependencies. Stage 2 was broken into 3 stages, labelled with: a, b, and c. The figure on the right shows the software pipelined version of the codegraph on the left. The subscript indicates the data set that the stage is acting on. Due to the data dependence between stage $2a$ and stage 3, stage $2a_2$ overwrites the results from $2a_1$ which stage 3_1 depends upon. A FIFO is required to prevent this data hazard from occurring. The data from $2a_1$ is saved and reloaded in $2c_1$. The register is preserved across $2a_2$ and consumed by 3_1

  - Increases register pressure, as these edges are now carried across all supergroups after the load until the value is consumed.

- Increment the FIFO pointer after the FIFO store is called, but before the load. Then decrement once loaded.
  - No changes to register pressure.
  - Potential throughput decrease due to additional instructions required for pointer operations.

Due to uncertainty on the effect on throughput caused by the FIFO pointer increment/decrement instructions, the former method was used.

2.3.3 Constant Load Insertion and Register Allocation

Numerous instructions within MASS depend on constants which are used throughout the entire function. In a typical schedule the constant may be loaded into a register, and once all dependant instructions utilize the value during one loop iteration the value may be overwritten. To reduce the number of constant loads occurring during a given iteration a set number of constants are loaded permanently at the beginning of the function. The number of constants that can be loaded at the top of the function is the difference between the number of registers available on the processor and the maximum register pressure of the function. If the maximum register pressure of the function is larger than the number of registers available on the processor then constants cannot be permanently loaded, instead spills are inserted (refer to 2.3.4).

The remaining constant loads that are not permanently loaded are inserted after the initial schedule is created. Groups are generated based on the processors specifications, and if any groups are found to have slots available, constants are inserted. If there are no slots available before an instruction which depends on a constant is called then a group of constants is formed and inserted before the earliest dependent instruction.

The register allocation is then performed using a standard graph coloring algorithm. All instructions with data that is considered alive at any given point in time in the schedule are considered connected. The graph coloring algorithm will attempt to assign a register to each instruction such that there are no adjacent instructions that share the same register. Once this has been performed, the order of all instructions and their resident registers are defined.

2.3.4 Spills

If the maximum register pressure is still above the number of registers available on the processor, then spilling is required. Spilling involves moving data from a register into

the processor's cache. Loads have a much higher latency than register operations, resulting in a performance decrease when accessing data from memory. To reduce the impact on performance that spilling would incur, data which requires spilling should immediately be stored onto the stack and then reloaded several instructions before the dependent instruction. This would allow for the load data latency to be hidden by other instructions at the expense of slightly increased register pressure in the area local to the dependent instruction.

The registers which are most suited to be spilled are those with long register lifetimes which intersects with a region of high register pressure. The longer the edge the more opportunities to hide the load and store latencies. Spills can be handled using FIFOs, allowing for spills to occur between stages with a separation of 1. Otherwise, spills within the same stage only require an address on the stack with a single offset pointing to each spill location.

Chapter 3

Evaluation

3.1 Evaluation Platform

The modified Karger minimum cut (mKMC) algorithm was implemented in Haskell running ghc 7.6.3. All timings were produced on IBM z13 mainframes. During each mKMC cycle, $n \log n$ iterations were performed, producing one potential schedule. Approximately 400 such schedules were produced and compared with multiple different metrics, described below. If there was a case where the number of nodes being topologically sorted exceeded 11, such as when the supergroups were being interleaved, a randomized topological sort was used in order to avoid a non-polynomial increase in computation time; the number of randomized topological sorts taken in this case was set to an arbitrary number, 2000.

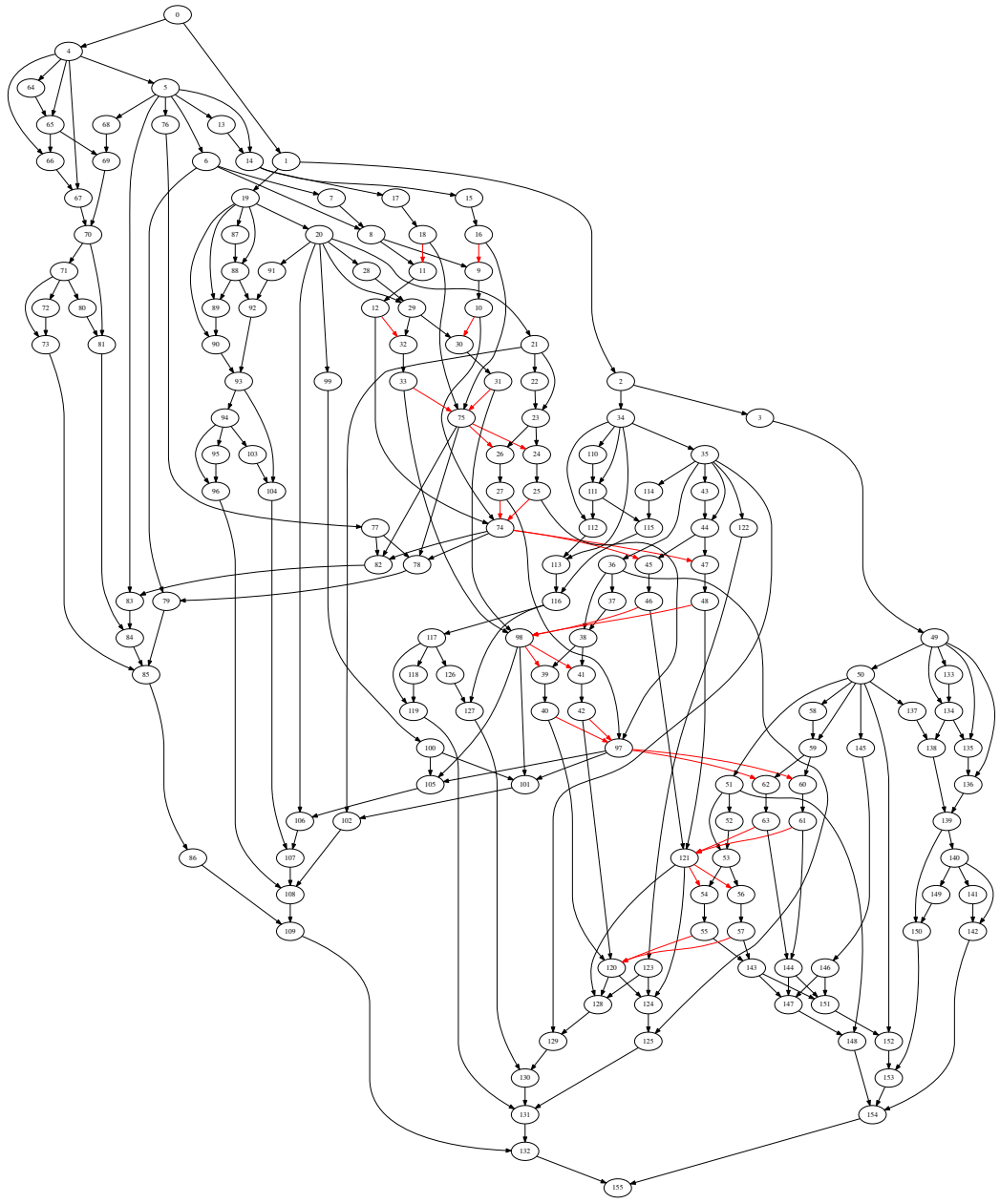


Figure 3.1: MASS Cosine Codegraph: An example of the dependency graph generated by the scheduler for a prototype MASS cosine function. Nodes represent individual instructions, the instruction names have been removed and replaced with index numbers in order to prevent describing platform proprietary details of instructions. The black edges represent data dependencies between instructions. The red edges represent false edges, edges which were placed in the graph in order to force a specific ordering of instructions.

Due to the limited number of general purpose registers (GPRs) available for the table lookup instructions false, dependency edges were inserted in order to limit the total number of GPRs active at once, as seen in Figure 3.1. The locations of the false edges were produced by Coconut prescheduling graph transformers and limits the maximum number of GPRs active at one time to 6.

Two metrics were used to gauge the desirability of a given schedule. The first being a schedule is seen as more desirable if it has a lower maximum register pressure. This would allow for additional constants to be loaded at the beginning of the loop and remain resident throughout the function, reducing the number of constant loads throughout the execution of the function as per 2.3.3.

The second metric involved weighing the likelihood of a hardware hazard occurring, the maximum register pressure, and the total number of FIFOs. This metric was chosen to attempt to reduce the total number of loads for both constants and FIFOs, while also preventing stalling which occurs when the functional units are saturated.

The functions chosen for analysis are arccosine, cosine, and cube root. The reasons are as follows: cosine was selected as a lookup-containing function which does not require spilling, cube root is a non-lookup-containing function which does not require spilling, lastly arcosine was chosen as a lookup-containing function which requires spilling. None of the non-lookup-containing functions we considered required spilling.

The distribution of the execution time against a given metric will indicate whether or not an approximation algorithm is a suitable fit for this instruction scheduling problem. A binomial distribution is the expected distribution due to the results being discrete and the randomized approximation algorithm likely generating a normal distribution. Multiple metrics were considered for this relation. The hope is that a metric is found that can be used in order to estimate the timing of a given schedule. This would allow a score to be generated that would allow schedules to be tested against one another without requiring hardware timing.

3.2 Metrics

3.2.1 Register Pressure

In this case only register pressure was considered when evaluating schedules. It was likely that functions that contain high latency instructions will have improved performance if the low latency instructions are serialized, allowing the high latency instructions to be more spaced out. This is especially true in the case of the lookup-containing functions as their expected latency is higher than both floating point and non floating point instructions that are present in other functions. The register pressure calculated only looks at vector register pressure, the false edges inserted for the lookup-containing functions results in fixed GPR register pressure.

3.2.2 Combined Metric

The likelihood that a data hazard would occur was evaluated by counting the number of instructions per supergroup which share the same functional unit and have a high execution latency. This combined with the number of FIFOs, the supergroups' internal register pressure, and total external register pressure were used to produce the combined metric score for a given schedule. Other permutations of this metric were tested, including simply data hazard likelihood, number of FIFOs, and simply internal pressures, but were not found to be predictive of performance and were not included in the results section.

3.2.3 Expected Result Distribution

According to the central limit theorem, the sum of an set of random variables which are independent and identically distributed will approximately fit a normal distribution. Under the simplifying assumption that edge weighting and super-group limiting can be ignored, our approximation algorithm merges all pairs of adjacent instructions with equal, independent probability, and each subsequent step will result in additional mergers occurring. The final result is some set of supergroups which are the result of identically distributed independent events. The hypothesis is that this merging can be modelled using a normal distribution and approximated using a binomial distribution. A binomial distribution is the likelihood that an event occurs some number of times over several trials, with the probability of this event occurring remaining the same across all trials. As the limit of the number of trials approaches infinity this distribution fits that of a normal distribution. To test the hypothesis the experimental data will be compared to a binomial distribution.

3.3 Results

Two primary methods were used to determine the value of a given metric. The first was that the metric compared to the timing was tested using a Pearson Correlation test to determine whether or not they were correlated. Secondly, both the timings and metrics were compared to a binomial distribution using a quantile-quantile plot (QQ plot). The quantiles of a set of data are compared to the quantiles of a binomial distribution, and if both lie on a linear slope then it can be said that a binomial distribution fits the data set.

3.3.1 Register Pressure Metric

Arccosine

Maximum register pressure follows a binomial distribution as seen in the QQ plots in Figure 3.2. No correlation was found between execution time and maximum register pressure, resulting Pearson correlation test returned an r^2 value of -0.0095855 with a two-tailed p value of 0.9151638. The timings do not fit well enough to the QQ plot to determine whether or not the distribution fits a binomial distribution.

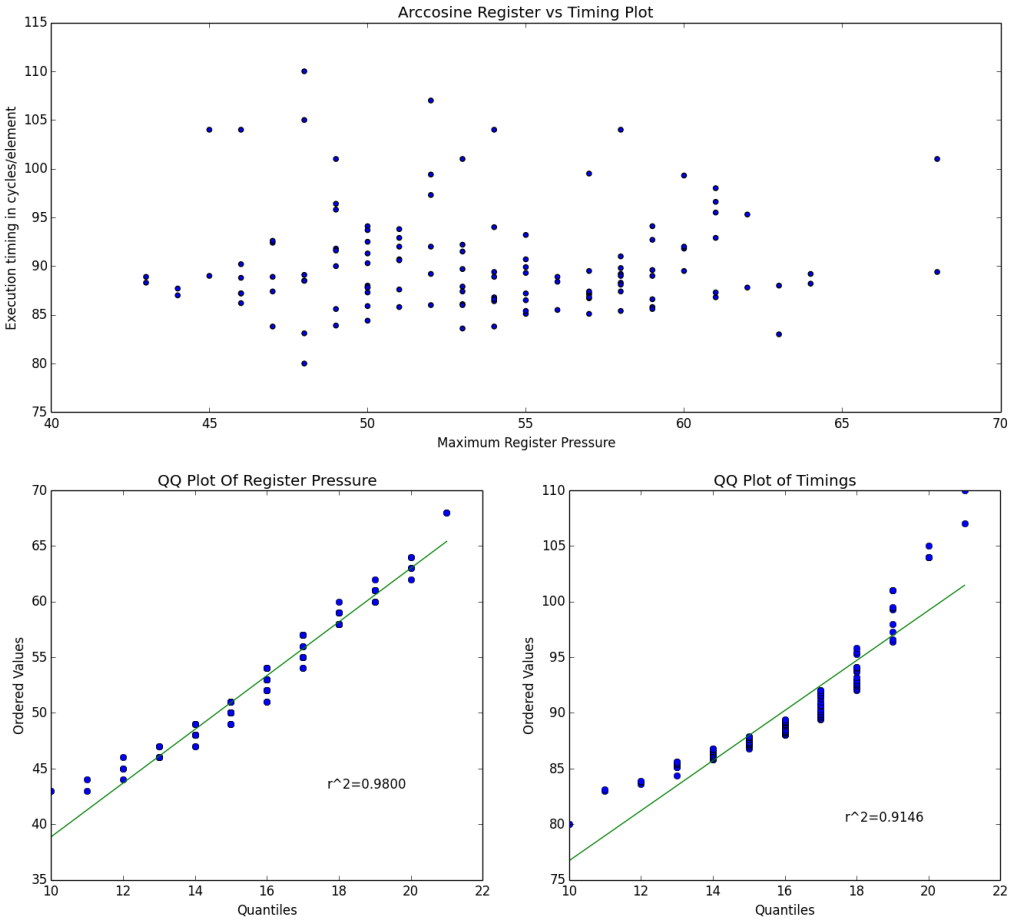


Figure 3.2: Analysis of Arccosine: Pearson Correlation r^2 value of -0.00958553 with a two-tailed p value of 0.91516383. The timings are likely not following a binomial distribution.

Cosine

Both execution time and maximum register pressure follow a binomial distribution as seen in the QQ plots in Figure 3.3. A positive correlation was found between execution time and maximum register pressure, resulting Pearson correlation test returned an r^2 value of 0.6806881 with a two-tailed p value of 1.0322e-53.

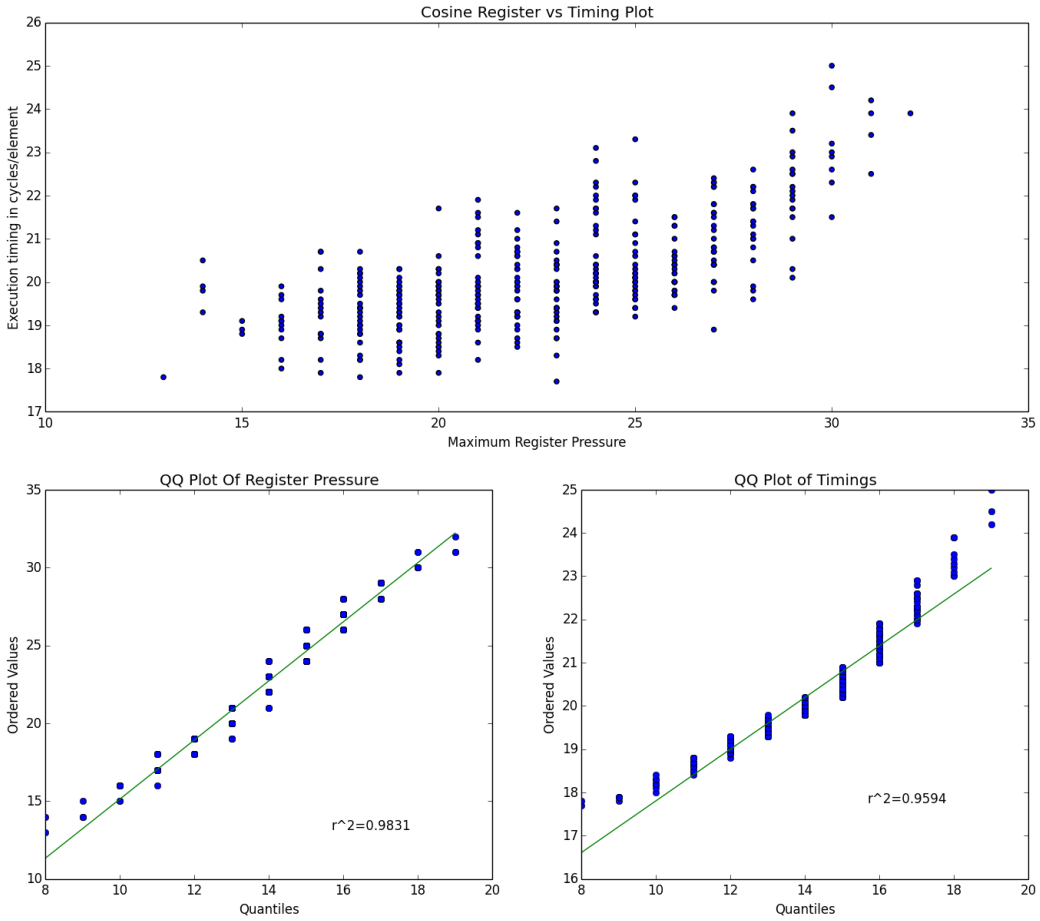


Figure 3.3: Analysis of Cosine: Pearson Correlation r^2 value of 0.68068807 with a two-tailed p value of 1.03227304e-53.

Cube Root

Both execution time and maximum register pressure follow a binomial distribution as seen in the QQ plots in Figure 3.4. A negative correlation was found between execution time and maximum register pressure, resulting Pearson correlation test returned an r^2 value of -0.346193979 with a two-tailed p value of 2.97621289e-12.

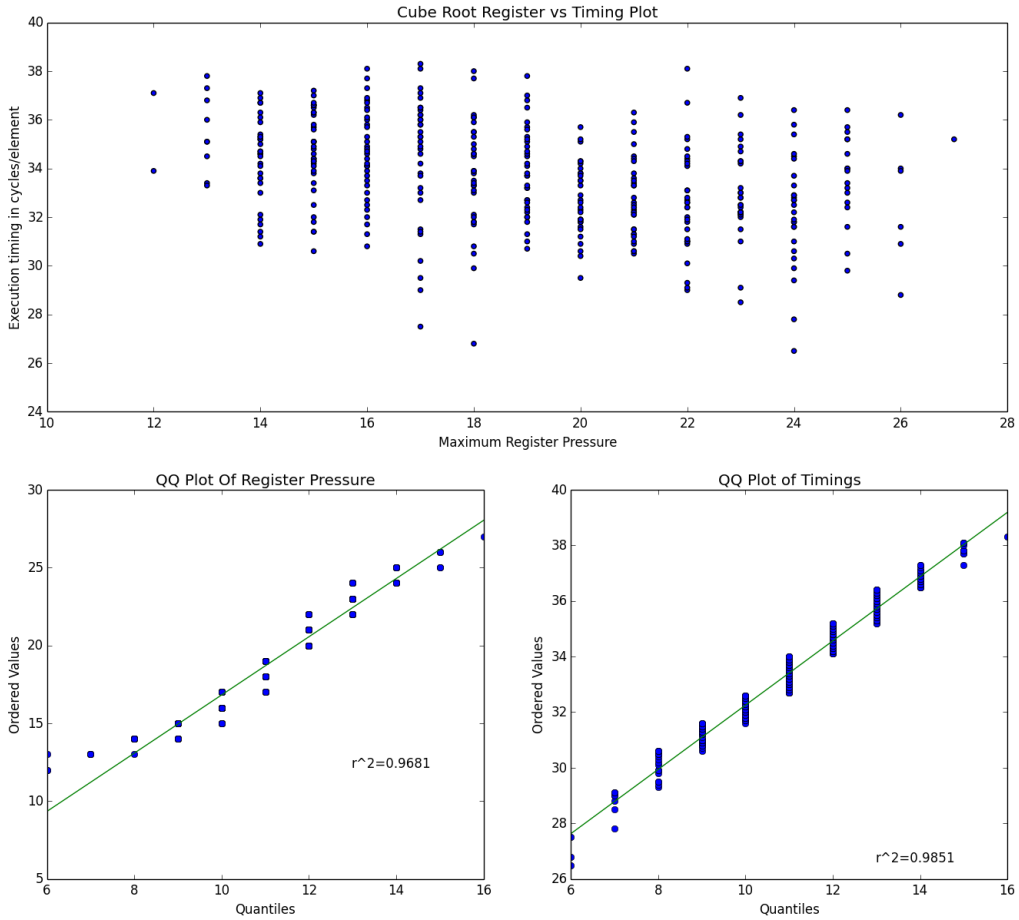


Figure 3.4: Analysis of Cube Root: Pearson Correlation r^2 value of -0.26167449 with a two-tailed p value of 7.40485540e-07.

3.3.2 Combined Metric

The combined metric was not found to predict performance. One example of a permutation of a combination of these three values is shown below.

Reciprocal Square Root

The selected metric did not return a score which fit a binomial distribution. No correlation was found between execution time and the score of a schedule, resulting Pearson correlation test returned an r^2 value of 0.163938727 with a two-tailed p value of 4.9877904653e-7. Different permutations of this scoring system were used, but were not statistically significant. Refer to Figure 3.5 which contains exclusively the maximum register pressure to timing graph.

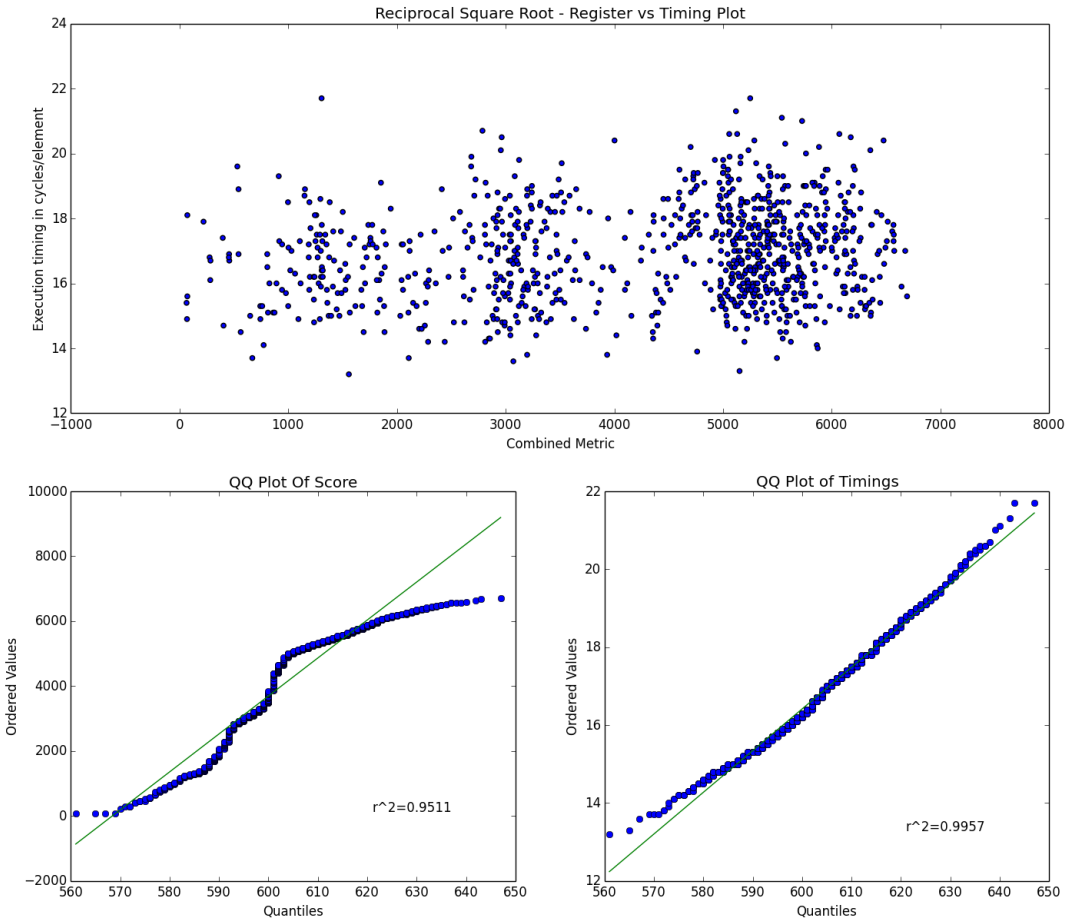


Figure 3.5: Analysis of Reciprocal Square Root: Pearson Correlation r^2 value of 0.16393872 with a two-tailed p value of 4.98779046e-7. The scoring does not fit a binomial distribution.

3.4 Discussion

3.4.1 Predicting Schedule Optimality

The lookup-containing functions typically have a higher number of constants to load compared to other functions. If register pressure is minimized, it introduces more opportunities to store constants permanently as described in 2.3.3. This reduces the number of instructions present in the loop body, and it was expected that this would allow for a decrease in execution time. Due to this, it was expected that there would be a positive correlation between register pressure and timing. This hypothesis was initially verified with cosine and later confirmed with exponent, base 2 exponent, and sine.

Initial expectations were that for non-lookup-containing functions without spilling, register pressure and execution time would be positively correlated. As register pressure was reduced more constants could be moved into permanent registers, in the same manner as the lookup instructions. The initial function tested, cube root, instead had a weakly negative correlation. Investigation into the function showed that it had a larger number of floating point instructions to total instructions. It was likely that in this case what was occurring was that as dependent instructions were moved closer together and reducing register pressure it introduced more serialization. If the instructions that were being serialized were high latency floating point instructions then it is likely that the reduction in execution time caused by the movement of constants out of the loop body was offset by the decrease in overall throughput of the new schedules. However, if a function had a relatively low ratio of floating point to total instructions then it is possible that non-floating point instructions would be able to still mask the latency of the floating point instructions. Combined with the decreased execution time of removing the additional constant loads it is likely that the low ratio case would likely result in a positive register pressure to timing correlation. The ratio of floating point instructions to total instructions was compared to the overall correlation of the function. Only 8 such functions exist in the MASS library and no correlation was found between the ratio of floating point instructions and the timing of a given function. In this case simply looking at register pressure is not enough, the scheduler needs additional information concerning the machines architecture in order to avoid hardware hazards more consistently.

Functions which include spills and lookup instructions did not have a well defined correlation with timing. It is likely that any correlation between register pressure and timing is reduced or removed with the introduction of spill load and store instructions. As the register pressure increases so does the number of spills required to schedule the function. This means that any increase in throughput caused by dependency edges becoming longer is likely hidden by the increase in execution time due to the

addition of extra instructions inserted to store spills. Hyperbolic tan was a function which required on the range of 30 registers to be scheduled. From maximum register pressure 22 to 30 the general positive correlation of a non-spill lookup-containing function can be seen, however once spilling is required beyond 30 the correlation is lost. This shows the effect of spilling on the resulting timings of a given function.

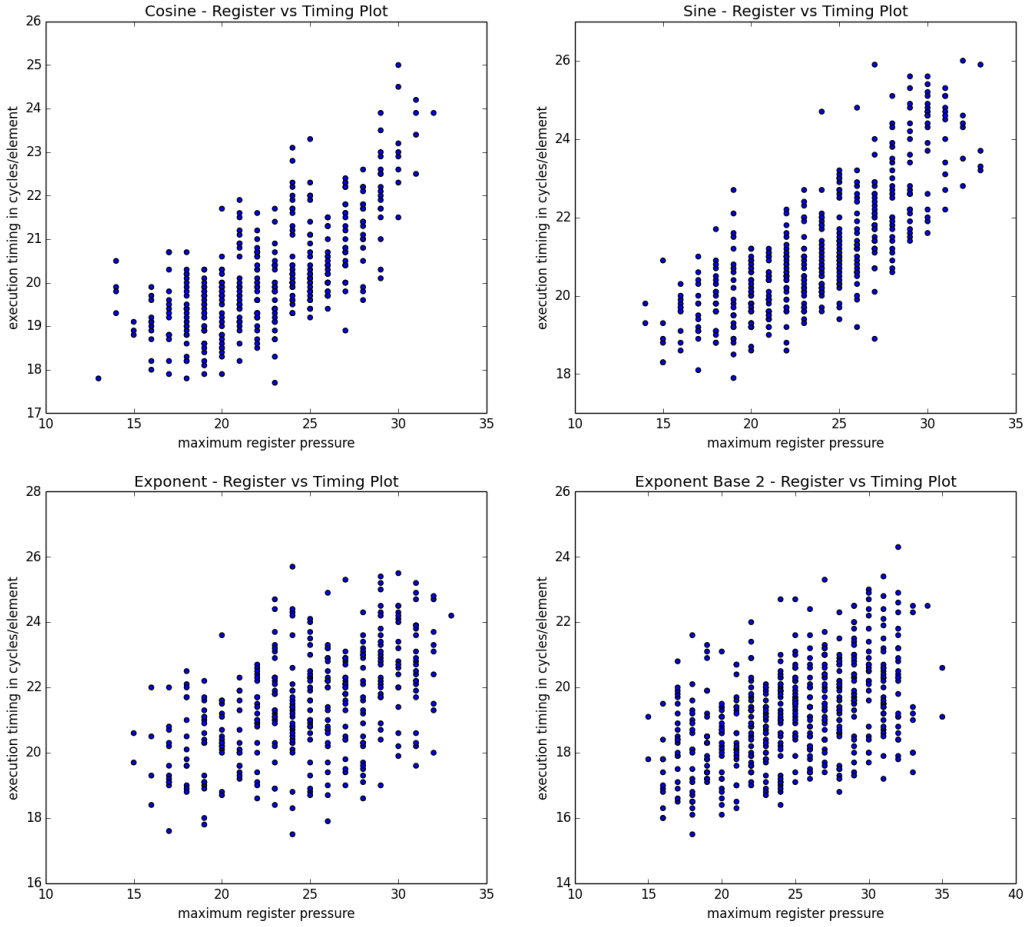


Figure 3.6: Additional lookup-containing functions: Top left: Cosine: r^2 value of 0.68068807. Top Right: Sine: r^2 value of 0.76704230. Bottom Left: Exponent: r^2 value of 0.47421500. Bottom Right: Exponent Base 2: r^2 value of 0.47251520.

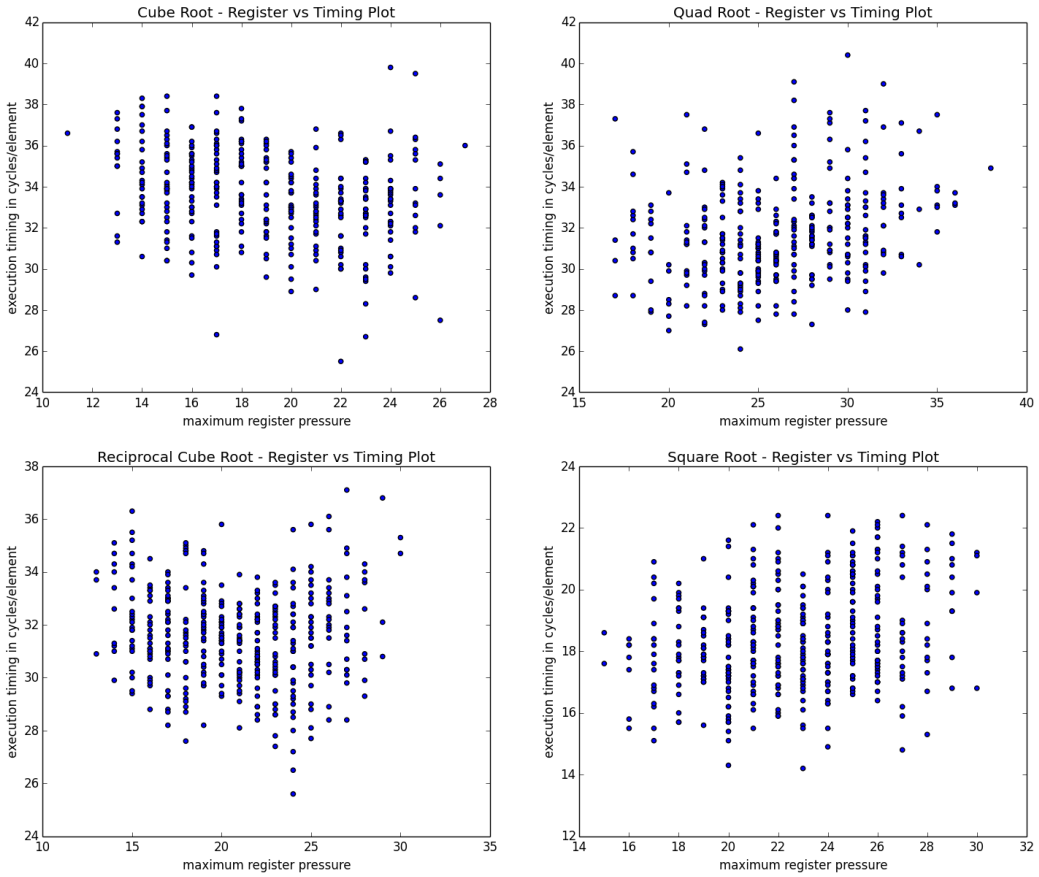


Figure 3.7: Additional non-lookup-containing functions: Top left: Cube Root r^2 value of -0.26167449 and a floating point to total instruction ratio of 0.65789473. Top Right: Quad Root r^2 value of 0.27568479 and a floating point to total instruction ratio of 0.47682119. Bottom Left: Reciprocal Square Root r^2 value of 0.08524685 and a floating point to total instruction ratio of 0.0.40816326. Bottom Right: Square Root r^2 value of 0.26716400 and a floating point to total instruction ratio of 0.44.

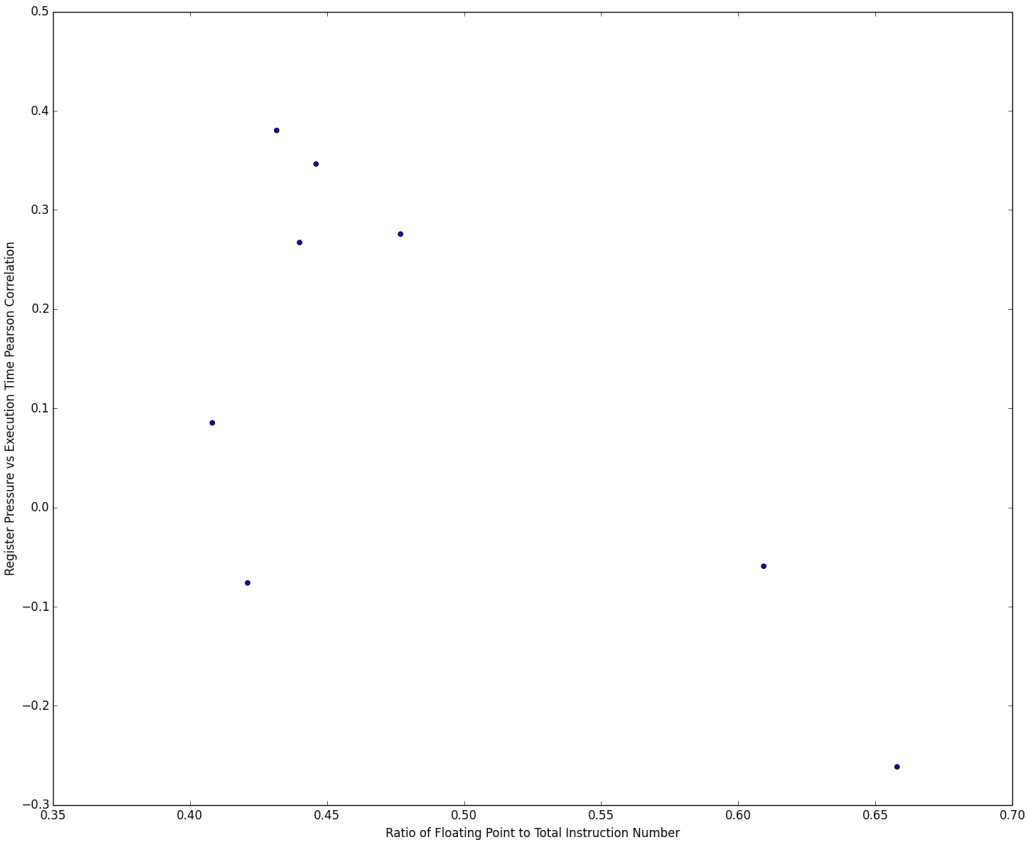


Figure 3.8: Effect of floating-point-instruction fraction on the correlation between execution time and maximum register pressure. The eight non-lookup-containing functions were tested for a correlation between register pressure and execution time. That correlation was then compared to the ratio of floating point instructions to total number of instructions. The resulting Pearson correlation returned an r^2 value of -0.67446110 and a two-tailed p value of 0.06656117. The eight functions tested were: cube root, reciprocal cube root, square root, reciprocal square root, quad root, reciprocal quad root, reciprocal, and hypotenuse.

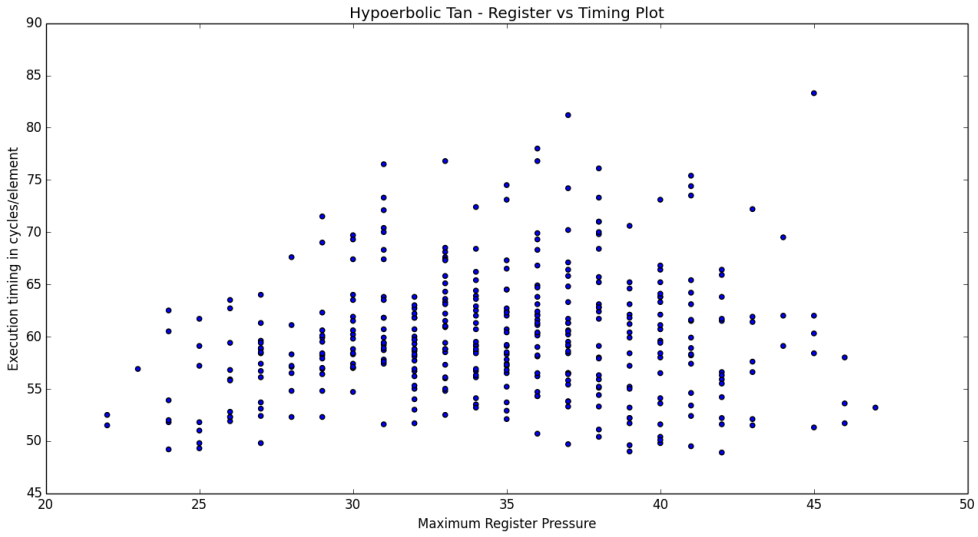


Figure 3.9: Hyperbolic Tan: Pearson Correlation r^2 value of 0.11364341 with a two-tailed p value of 0.02694837. The timings are likely not following a binomial distribution. Notice the shift from a positive correlation to noise at approximately maximum register pressure = 31.

3.4.2 Scheduler Performance

The scheduler was capable of producing valid schedules for all z13 MASS functions that were available at the time of this writing. The execution times of the resulting schedules were also within the expected acceleration compared to scalar functions written in C. Since many aspects of the z architecture are proprietary, we will not be providing timing information.

3.4.3 Benefits of an Approximation-Algorithm Approach

In other contexts, approximation algorithms provide expected performance defects, which is the next best thing to optimal solutions, and a big improvement over waiting for non-polynomial algorithms. Approximation algorithms for instruction scheduling are going to be more complicated, and harder to prove properties about, but as a first step, we asked, do performance metrics follow a known statistical distribution, which can be used to predict or gauge progress.

The schedules of the non-spilling MASS functions were found to produce a distribution of timings that approximately fit a binomial distribution, as seen by the QQ plots. This fact allows us to make assumptions about the expected performance of

a given schedule produced by this algorithm. Constraints on the algorithm, such as the insertion of false edges, seemed to have minimal effect on the final distribution. Only the tail cases seem to deviate from that of a binomial distribution, however it is uncertain as to whether or not this is due to the constrained system or due to insufficient data points.

To start, if for a given performance requirement, Q, there is a probability, s, of a schedule meeting this requirement at each iteration and the total number of schedules produced is i then the likelihood that at least one schedule meets this requirement is p, which is defined by: $(1-p) = (1-s)^i$. It is possible to rearrange this to solve for the number of cycles required to produce a schedule with the required likelihood:

$$\frac{\log(1-p)}{\log(1-s)} = i \tag{3.1}$$

Refer to Figure 3.10 for examples.

Probability of Best Schedule Meeting Requirement	Percentile of Schedules Meeting Requirement	Number of Schedules Required
95%	10%	28 schedules
95%	5%	58 schedules
99%	1%	458 schedules
99.9%	1%	687 schedules

Figure 3.10: Examples of the number of schedules required to generate schedules with a set likelihood.

Defining a schedule as being within the top n% is difficult to work with. If a boundary is set on the normal distribution that defines a near-optimal schedule then it is possible to convert the performance percentile variable to cycles per element away from near-optimal. For example, rather than defining a schedule in the top 5% of all schedules it is possible to find a schedule within 1 cycle per element of the estimated optimal schedule. To investigate an approximate boundary for the near-optimal schedule the standard deviation of multiple timing distributions were tested and compared with the best schedule found by the algorithm.

The ratio of difference between the mean and the minimum execution time to the standard deviation (σ) will be used in order to define the boundary for the most likely optimal schedule. An analytical solution would be the preferred method of determining this boundary, however a reasonable approximation can be made by looking at the experimental values in Figure 3.11. A boundary of 3σ was chosen based on the data in the figure as the boundary in which the optimal solution likely

Function Name	Mean	Minimum	Standard Deviation	Ratio
Exponent Base 2	19.2076335878	15.5	1.50561890704	2.4625312358
Exponent	21.481920904	17.5	1.66994391951	2.38446384782
Cosine	20.1968831169	17.7	1.26418831794	1.97508795284
Sine	21.1913551402	17.9	1.65833917454	1.98472977707
Cube Root	33.5968390805	25.5	2.20129698119	3.67821295793
Reciprocal Cube Root	31.5793733681	25.6	1.83357648886	3.26104386944
Hypotenuse	22.9673366834	19.8	1.22551945389	2.5844850307
Square Root	18.4198019802	14.2	1.6721226092	2.52361995286

Figure 3.11: Examples of standard deviations compared with the best-observed schedule. The ratio is determined by finding the difference in the mean and the minimum and dividing the value by the standard deviation.

exists for non-lookup functions, and boundary of 2σ was chosen for lookup based functions. The ordering of the lookup instructions result in tighter constraints which limit the total number of schedules available in the tail-cases. A normal distribution is defined as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.2}$$

The integral can be taken from $-\infty$ to x' in order to determine the probability that a sample is less than or equal to some value x'.

$$\begin{aligned} f(z) =& \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{\frac{-z'^2}{2}} dz' \quad \text{with} \quad z = \frac{x-\mu}{\sigma} \\ F(z) =& \frac{1}{\sqrt{2\pi}} \frac{\sqrt{\pi}}{\sqrt{2}} \left[\text{erf}\left(\frac{z}{\sqrt{2}}\right) - \text{erf}\left(\frac{-\infty}{\sqrt{2}}\right) \right] \\ F(z) =& \frac{1}{2} \left[\text{erf}\left(\frac{z}{\sqrt{2}}\right) + 1 \right] \end{aligned}$$

$$F(x) = \frac{1}{2} \left[\text{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right) + 1 \right] \tag{3.3}$$

Using this equation it is possible to define a radius around the approximated schedule at 2σ or 3σ and determine the schedule percentile required in order to produce a desired schedule. For example, if a schedule was desired to be within 1 cycle per element of the optimal for cosine:

Determine the desired schedule's distance away from the mean:

$$\begin{aligned} x_1 =& 2\sigma - 1 \\ x_1 =& 2 * (1.26418831794) - 1 \\ x_1 =& 1.52837663588 \end{aligned}$$

Determine the probability of selecting a worse schedule:

$$\begin{aligned} F(x_1) =& \frac{1}{2}\left[\text{erf}\left(\frac{x_1}{\sqrt{2}\sigma}\right) + 1\right] \\ F(x_1) =& 0.88666446848 \end{aligned}$$

Therefore, a schedule in the top 11.3% is required to satisfy those constraints. Following equation 3.1, in order to produce a schedule with this level of performance with a guarantee of 99.9% it would require approximately 58 schedules to be produced. The time required to generate each schedule is low. The approximate time per cycle for a number of functions is shown below. As the number of instructions in the function increases so does the execution time. The complexity of the algorithm was also tested using the table and graph below. The schedules were produced on a personal desktop running with an Intel 4690k processor on a single thread. A second order polynomial was fit to the graph, giving a likely complexity of approximately n^2, which is the expected complexity considering it is simply a modified version of Karger's minimum cut algorithm. However, as stated before the scheduler is at a prototype state, and it is possible that other aspects such as the topological sorting may be masking the true complexity of the combined approximation algorithm and heuristics. Additional functions will also need to be tested in order to allow for intermediate values to be filled in.

Function Name	Number of Instructions	Seconds Per Schedule
Arccosine	283	3.9
Hyperbolic Arccosine	263	5.0
Arcsine	283	3.6
Hyperbolic Arcsine	335	9.1
Two-Argument Arctan	324	8.4
Arctan	301	7.5
Hyperbolic Arctan	255	3.3
Cube Root	159	1.6
Cosine	155	1.5
Hyperbolic Cosine	194	2.2
Exponent Base 2	136	1.3
Exponent	140	1.3
Hypotenuse	135	1.4
Logarithm	125	1.7
Logarithm Base 2	117	1.7
Power v1	688	38.8
Power v2	368	7.4
Quad Root	158	1.6
Sine	163	1.9
Square Root	111	1.0
Hyperbolic Tan	259	3.6
Tan	263	3.9

Figure 3.12: Examples of scheduler timing per function: these times were based on the total execution time for a 50 schedule run.

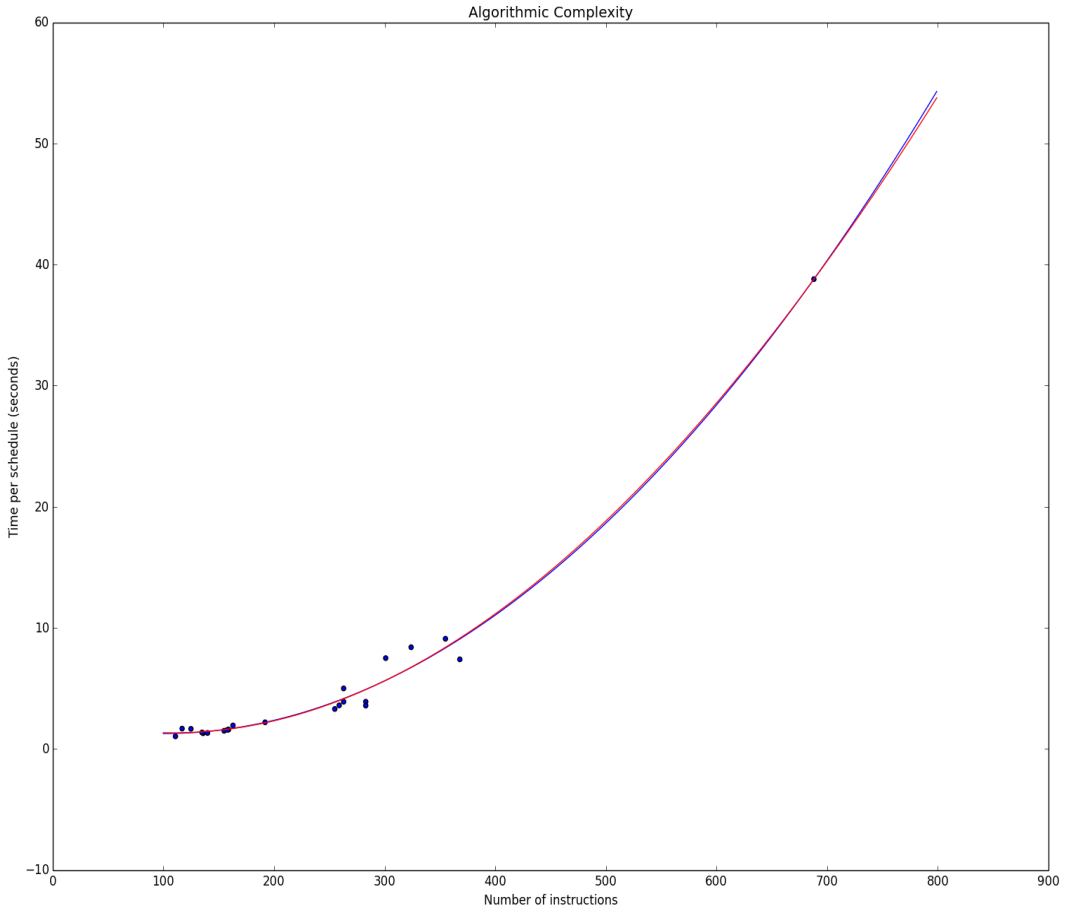


Figure 3.13: Algorithmic complexity graph generated from Figure 3.12. The scheduler was run on an Intel 4690k processor to determine the number of seconds per schedule. The data was fit to a degree-2 polynomial giving an estimated complexity of n^2 per schedule. However, the approximation algorithms and heuristics were not isolated from other aspects of the scheduler which may have skewed the complexity. Also, additional larger functions are required in order to properly observe the increase in execution time at high instruction numbers. The blue line is the second order polynomial fit with the following constants [1.08439266e-04, -2.16298058e-02, 2.35041415e+00]. The red line is a third order polynomial fit with the following constants [-1.37813359e-08 1.23925733e-04 -2.63618794e-02 2.74855162e+00]. The constants are in aranged of increasing order (c_1+$c_2 * x$+$c_3 * x^2$...)

This can all be tied together in order to allow for a scheduler with easily modifiable parameters. For any function to be scheduled, trial schedules can be sampled until the standard deviation of the distribution is determined within a reasonable degree. This standard deviation can then be combined with the estimation of the near-optimal boundary to allow a user to define the desired performance. Finally, the number of instructions to be scheduled combined with the number of trial schedules that would be required can be used to provide an estimate for total CPU time for scheduling.

Chapter 4

Conclusion

The initial investigation into the approximation-algorithm based scheduler proposed by this thesis establishes expected properties of the resulting schedules and areas for improvement. One advantage of this approach is that it allows for an explicit tradeoff between performance and scheduling time in a way that is easily understood by a user. Rather than turning optimizations on or off, an expected final execution time is set relative to the expected optimal schedule. This prototype, which only supports modulo-scheduling of loops for the z13 processor, is capable of producing production-quality schedules with minimal user involvement.

The timing results show multiple areas for possible improvement. Execution-unit information needs to be added to the initial codegraph in order to minimize the likelihood that one supergroup is filled with instructions that depend on the same functional unit, resulting in a hardware hazard. Additional methods also need to be investigated for combining supergroups into schedules. Specifically, methods for finding long dependency chains and interleaving them with non-adjacent instructions. The internal scheduling heuristic also needs to be improved. The investigation into the register pressure metric shows that a list scheduler based on register pressure is not the optimal way to produce an optimal internal schedule. A metric that involves the additional information provided by an updated codegraph would likely allow for an improved estimate of final schedule timing without explicitly having to test the execution time of each schedule. Finally, when the algorithm variations are better understood, the scheduler should be remade to make it embeddable in multiple toolchains. For toolchains which support multiple processors, this algorithm is easily parallelized.

Bibliography

Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). A comparison of list schedules for parallel processing systems. *Commun. ACM*, **17**(12), 685–690.

Anand, C. K. and Kahl, W. (2009). An optimized Cell BE special function library generated by Coconut. *IEEE Transactions on Computers*.

Clausen, J. (1997). Branch and bound algorithms — principles and examples. *Parallel Computing in Optimization*, pages 239–267.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Gibbons, P. B. and Muchnick, S. S. (1986). Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.*, **21**(7), 11–16.

Gonzalez, Jr., M. J. (1977). Deterministic processor scheduling. *ACM Comput. Surv.*, **9**(3), 173–204.

Hennessy, J. L. and Gross, T. (1983). Postpass code optimization of pipeline constraints. *ACM Trans. Program. Lang. Syst.*, **5**(3), 422–448.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

IBM (2015a). Mathematical acceleration subsystem family. `http://www-03.ibm.com/software/products/en/mathaccesubsfami`. Accessed: 2015-01-17.

IBM (2015b). Optimization and programming guide. `http://publibz.boulder.ibm.com/epubs/pdf/cbc1p210.pdf`. Accessed: 2015-01-26.

IBM (2015c). Optimization and programming guide. `http://www-01.ibm.com/support/docview.wss?uid=swg27046908&aid=11`. Accessed: 2015-01-26.

IBM (2015d). IBM z13 (z13). `http://www-03.ibm.com/systems/z/hardware/z13.html`. Accessed: 2015-01-17.

Johnson, S. C. (1990). The i486 CPU: Executing instructions in one clock cycle. *IEEE Micro*, **10**(1), 27–36.

Karger, D. R. and Stein, C. (1996). A new approach to the minimum cut problem. *J. ACM*, **43**(4), 601–640.

Landskov, D., Davidson, S., Shriver, B., and Mallett, P. W. (1980). Local microcode compaction techniques. *ACM Comput. Surv.*, **12**(3), 261–294.

Lascu, O. e. a. (2015). *IBM z13 Technical Guide*. IBM REDBOOKS.

Llosa, J., González, A., Ayguadé, E., and Valero, M. (1996). Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 80–86. IEEE.

Page, D. (2009). *A Practical Introduction to Computer Architecture*. Springer Publishing Company, Incorporated, 1st edition.

Park, G.-L., Shirazi, B., Marquis, J., and Choo, H. (1997). Decisive path scheduling: A new list scheduling method. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 472–480.

Smith, J. and Pleszkun, A. (1988). Implementing precise interrupts in pipelined processors. *Computers, IEEE Transactions on*, **37**(5), 562–573.

Thaller, W. (2006). *Explicitly Staged Software Pipelining*. Master's thesis, McMaster University, Department of Computing and Software. http://sqrl.mcmaster.ca/~anand/papers/ThallerMScExSSP.pdf.