# DETECTING INSTRUCTION SCHEDULING CONSTRAINTS

**Julie Parent**

**May 2003**

Thesis submitted to

the Department of Computer Science

at Hamilton College

in partial fulfilment of the requirements

for departmental honors


**Department of Computer Science**
**Hamilton College**
**198 College Hill Rd**
**Clinton, NY 13323**

# Contents

**Abstract**

Instructions can execute differently on a machine based on the way that they are arranged. The software technique called instruction scheduling performs the reordering of instructions to increase efficiency. A drawback of current instruction scheduling approaches is that detailed knowledge of the machine is required in order to be effective. This research focuses on examining a way to automatically schedule instructions without the knowledge of the scheduling constraints. We hypothesize that scheduling constraints are observable via timing. Our approach is to find a good schedule by timing sequences to find the one that executes the fastest. However, the amount of time required to execute one instruction is so minute that in order to detect small differences in timings, a timing system with very good resolution is needed. We consider two timing mechanisms: the platform independent `getTimeOfDay` function and the hardware dependent `TICK` register on the UltraSPARC.

# 1 Introduction

Speed is always an important issue in computer science; the desire to have faster computers drives the market today. Increased speed can come from two sources: hardware or software improvements. Speedup is achieved in hardware by building faster processors, faster memory, and other methods. One process that can speed up program execution via software is instruction scheduling, a technique detailed in the next section.

## 1.1 Instruction Scheduling

Computer programs are made up of a series of instructions that are processor dependent. Instructions can execute differently on a machine based on the way that they are arranged, leading to different execution times. The software technique called instruction scheduling performs the reordering of instructions to increase efficiency. Due to the detailed knowledge of the machine required to write a scheduler, it must be written by hand to select the most efficient ordering of instructions. Further, whenever a new or revised computer architecture is introduced, a new instruction scheduler must be written. This process is painstaking, time consuming, and error prone. Complicating the process is the fact that hardware descriptions are not always available to the programmer, and even worse, hardware descriptions often do not reflect the current architecture of the machine.

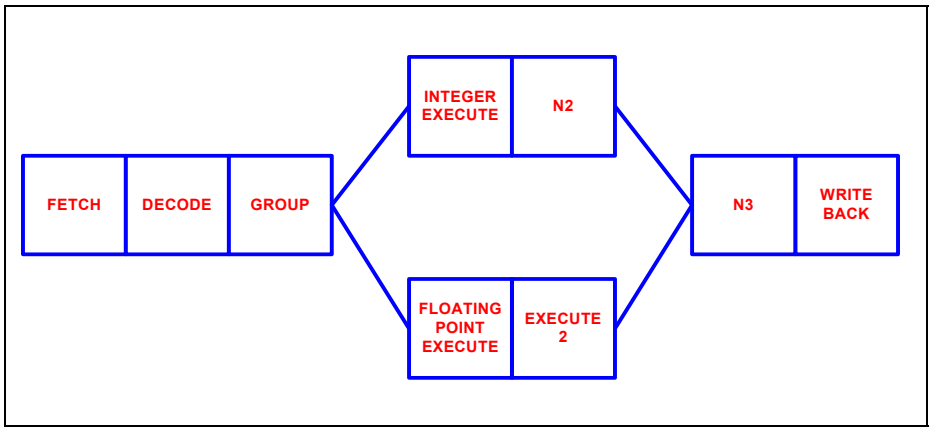


Figure 1. An Example Pipeline.

## 1.2 Pipelines

One architectural feature of modern processors is the pipeline. Pipelining speeds up execution of instructions by breaking instruction execution up into steps called stages. The stages are distinguished from each other because they involve different specialized hardware resources. An example pipeline is shown in Figure 1. Pipelines allow us to overlap execution. On non-pipelined machines, one instruction was required to completely finish before the next instruction could begin. With pipelining, the computer operates by starting the next instruction as soon as the first stage of the pipe is available. Figure 2 illustrates how instructions can be overlapped. In the non-pipelined example, each instruction takes four time units to execute. Thus, executing three instructions takes 12 units. When pipelined, each instruction still requires four time units to execute, but the instructions are overlapped, causing the total execution time to reduce to six units.

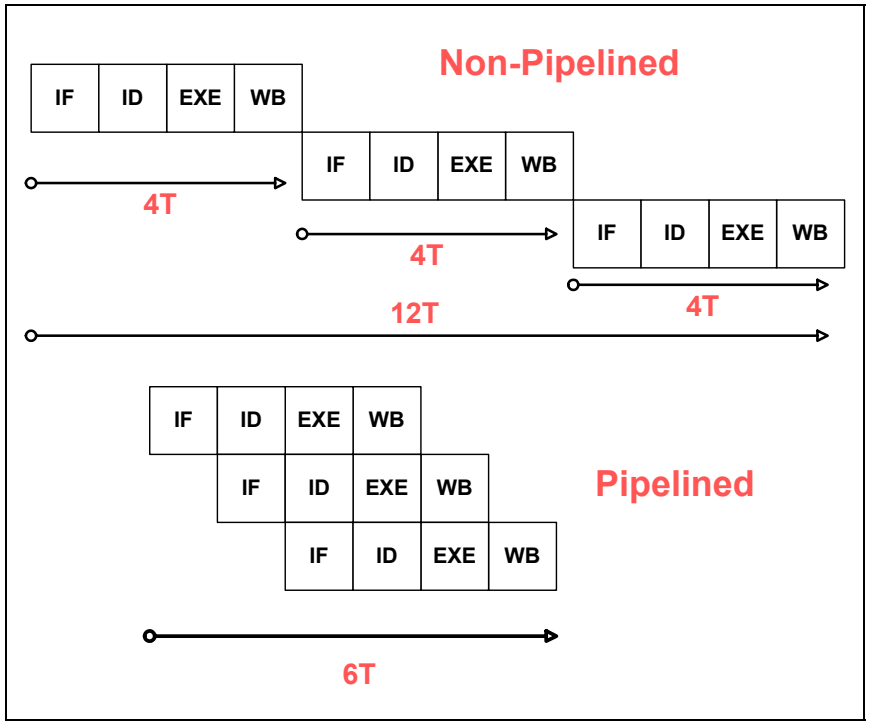


Figure 2. Overlapping Instruction Execution.

Pipelines are very powerful because they can greatly increase the execution speeds of programs. However, they can introduce problems, called *hazards*. There are two types of hazards: structural and data dependencies. When two instructions require the same resource at the same

time a *structural hazard* occurs, causing the processor to stall and wait for the current instruction to finish using the resource before giving the subsequent instruction access. An example structural hazard is shown in Figure 3. Both instructions require the use of the floating point multiply unit at the same time. The second instruction must stall until the first instruction advances to the next stage. Structural hazards can be reduced in the hardware by adding an additional copy of a resource if instructions require multiple cycles to use it, as is the case with the floating point multiply unit. Scheduling instructions such that an instruction does not require a functional unit at the same time as the previous instruction also removes the hazard and incurs no hardware cost.

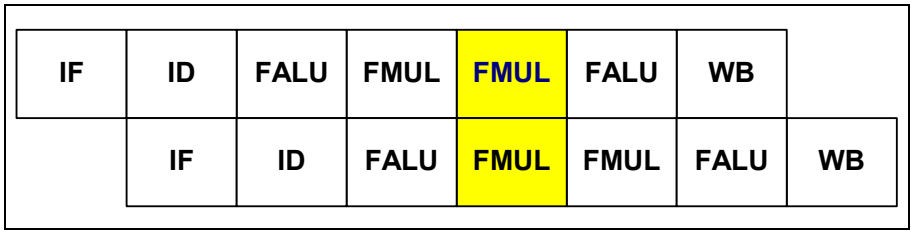


Figure 3. Structural Hazard using the FMUL resource.

Another type of hazard is a *data dependency.* When one instruction requires the result of the prior instruction, a data dependency may occur in which case the processor must stall until the result is ready for use. Figure 4 illustrates an example data dependency using register `o3`. In this example, the first instruction stores the result of an add into register `o3`. The second instruction requires the value in register `o3` to perform its computation. The second instruction must stall and wait for the updated value in register `o3` to be written before it can read from the register. This is known as a *read after write (RAW)* dependency. A *read after read (RAR)* dependency occurs when an instruction needs to read the same register as a prior instruction. A hazard does not occur in this case since the first instruction does not change the value of the register. Similarly, a *write after read (WAR)* dependency does not cause a hazard.

```
(1)  add %o1, %o2, %o3      // o3 = o1 + o2
(2)  add %o3, %o2, %o4      // o4 = o3 + o2
```

Figure 4. RAW Data Dependency.

Pipelined architectures have lead to dramatic increases in execution speed, but have added further complexity to instruction scheduling. Instruction scheduling produces faster executing programs by choosing sequences of instructions that avoid stalls in the pipeline and issue a new instruction every cycle (or in the case of superscalar machines, multiple instructions every cycle.)

### 1.3 The Problem

A drawback of instruction scheduling is that it requires detailed knowledge of the machine in order to be effective. This research focuses on examining a way to automatically schedule instructions without the knowledge of the scheduling constraints. The final goal is that when presented with a new computer, we will run a series of tests to determine the best way to organize the instructions. This information will be provided for a compiler, which will then automatically schedule the instructions efficiently, without the need for a hand written scheduler.

### 1.4 Organization

The rest of this paper is organized as follows: Chapter 2 examines related work; Chapter 3 presents issues with timing mechanisms and the size of the search space; Chapter 4 discusses the testing framework; Chapter 5 concludes; Chapter 6 discusses future work.

## 2 Related Work

This section is divided into four subsections: retargeting, general instruction scheduling, automated instruction scheduling, and timing.

### 2.1 Retargeting

An important aspect of compiler design is retargetability. When a new architecture is introduced, the ability to easily port a compiler to the new architecture is important. Similarly, when a new programming language is introduced, if the compiler can be easily ported across platforms, the language is more likely to succeed than one that took more effort to retarget. One problem with retargeting is that the machine description of the new architecture is needed. The machine description provides the details of the instruction set. The documentation provided by the vendors is often unreliable. There has been much work done on retargeting; several works stand out due to their unique approach.

In contrast to the traditional approach of doing optimization early, before code generation, Davidson instead performs optimizations on object code [Dav86]. The standard approach is to find the optimal sequence of instructions by choosing the solution with either the fewest lines of code or the fewest references to memory. The problem with this method is that although the algorithm used is machine independent, it does not produce good code because it does not have the machine dependent description required to do so. However, when optimization is performed on object code, as Davidson shows, the optimizer has complete knowledge of the machine so more thorough optimization can be done. In earlier work, two types of optimizations are done: targeting and evaluation order determination, using RTL-trees as the intermediate representation (IR). Targeting involves using the commutativity of arithmetic expressions to reduce the number of register loads and stores. Evaluation order determination finds the number of registers needed to evaluate a subtree and chooses the result with the highest number first. Performing these optimizations on object code greatly reduces the size of the object code.

Bradlee, Henry and Eggers developed the Marion system to be a retargetable code generator designed for RISC architectures [BHE91]. They focus their retargeting efforts on instruction scheduling, particularly for pipelined machines. The authors note that most scheduling properties of an instruction can be expressed in terms of operation latencies and pipeline requirements. The input to the Marion system is a machine description containing information about the registers, functional units, and pipeline stages. The Marion system uses a common structure in instruction scheduling to select the next instruction to be scheduled: a code directed acyclic graph (DAG). In the graphs used by the Marion system, nodes represent the instructions and edges represent dependencies. A root is an independent instruction. The `edge l = (x, y)` means that `y` cannot be scheduled fewer than `l` cycle after `x` or a hazard will be caused. The Marion system uses the *list scheduling* algorithm. A list stores all instructions that can be scheduled without causing a delay. The instruction chosen from the list is the one that is farthest from completion; in the graph, this is represented by the node with the maximum distance from the root. The authors conclude

that the Marion system is an example of how a good retargetable code generator can be built for a RISC architecture.

Colberg addresses the issue of obtaining a machine description. He notes that creating a description by hand is often error prone and has instead bypassed the documentation and focused directly on the machine's hardware and software. Collberg describes a different method to retarget compilers for a new architecture that is fast and requires little user input [Col97]. The idea is to use the native C compiler and assembler to determine features of the new architecture, producing a formal machine description that is fed into a back-end generator to produce a new code generator. The system creates C code programs and compiles them into assembly language samples on the new architecture. Information about the machine such as the procedure calling conventions, the register set, and assembler syntax is then extracted from the assembly samples. Next, a data flow graph is built to model how information flows between instructions in the samples. The nodes of the data flow graph are operators and operands. An edge from `A` to `B` represents that `B` uses a value stored in or computed by `A`. The system uses these graphs to discover the signature of individual instructions. The final output is a machine description derived from data output by all the other phases. The quality of the code produced by the new code generator is based on the completeness of the samples and the C compiler's ability to generate all instructions in the instruction set. Clearly, there will be some instructions that the new code generator will never generate because the system did not discover them.

Creating a compiler that is easy to retarget has many advantages. However, building a retargetable compiler is difficult. One of the greatest difficulties is in maintaining machine independence without sacrificing good code optimizations. The need to discovering the machine description for retargeting is similar to the need of determining architectural features to effectively schedule instructions.

## 2.2 General Instruction Scheduling

Instructions can execute differently on a machine based on the way that they are arranged, leading to different execution times. The software technique called instruction scheduling performs the

reordering of instructions to increase efficiency. Instruction scheduling has become increasingly complex with the introduction of pipelined architectures. Instruction scheduling produces faster executing programs by avoiding stalls in the pipeline and trying to issue a new instruction every cycle (or in the case of superscalar machines, multiple instructions every cycle). The burden of doing this scheduling is placed on the compiler. Thus, an effective compiler must include an instruction scheduler.

An aspect of instruction scheduling under investigation is improving the algorithms used to implement the scheduling. Gibbons and Muchnic describe an algorithm that reduces the number of pipeline interlocks while having $O(n^2)$ runtime, an improvement over previous $O(n^4)$ algorithms [GM86]. The key to this algorithm is the use of a DAG to prevent deadlocks and to schedule without using lookaheads to avoid deadlock. The algorithm reorders instructions at compile time. The approach is to divide the code into basic blocks, construct a DAG representing the scheduling constraints, and then schedule the instructions from the block using the DAG. The nodes of the graph represent instructions and edges represent dependencies. An edge from `instruction a` to `instruction b` means that `instruction a` must be executed before `instruction b`. The instruction chosen from the candidate set is based on the following three heuristics: whether the instruction interlocks with any of its immediate successors in the DAG, the number of successors the instruction has, and the length of the longest path from the instruction to a leaf. After an instruction is selected, it is removed from the candidate set and any new candidates are added.

The use of graphs in the task of scheduling is widespread throughout the different algorithms or heuristics employed to do the scheduling. Smotherman, Krishnamurthy, Aravind, and Hunnicutt focus on constructing DAGs to use for instruction scheduling of basic blocks [SKAH91]. In the DAGs created, the nodes represent instructions and the arcs represent dependencies. The DAG is used to select an instruction based on a heuristic. The authors considered six major categories of such heuristics: stall behavior, instruction class, critical path, uncovering, structural, and register usage.

Bernstein and Rodeh propose a scheme for instruction scheduling that goes beyond the typical basic block scheme to include loops to better increase efficiency [BR91]. However, execution of instructions from different iterations of the loop are not overlapped. A unique aspect of their approach is that the data and control dependencies are both expressed at once, using a type of acyclic graph known as the Program Dependence Graph. In the graph, nodes represent basic blocks and an edge from `node X` to `node Y` means control can flow from `block X` to `block Y`, thus the control dependences are modeled by the edges. Next, data dependency edges are added to the graph with an edge between `instruction a` and `instruction B` if a data dependency exists. Data dependencies are determined by testing every pair of instructions. Instructions are chosen from a set of candidates to produce a list of ready instructions whose data dependencies are fulfilled. The heuristic used for scheduling is the delay time, which measures how many delay slots are found on the path between an instruction and the end of the block. As the authors expected, the technique of moving beyond the basic block did produce more efficient code than obtained by only scheduling within the block.

Proebsting and Fraser use finite state automata to aid in instruction scheduling [PF94]. Their work focuses on detecting hazards using a finite state automaton. The information used to produce the automaton is a specification of the pipeline. The FSA maintains a state that represents all possible structural hazards for the instructions in the pipe. The FSA's input is an instruction type and the state of the machine. The output of the automaton is either a reported hazard or an advance of the pipeline and change of state. A transition is not present in a particular state if any subsequent instruction execution would cause a hazard. Instructions that have the same resource vectors or generate the same collision matrices form instruction classes. These classes help to reduce the size of the tables and create a more efficient approach.

Bala and Rubin also use an approach to instruction scheduling that uses finite state automata built from the instruction resource vectors to model a processor implementation [BR95]. The automaton is designed to recognize all legal instruction schedules created from the language whose alphabet is the instruction set of the processor. This means that the scheduler is reasoning

about the schedule instead of about hardware resource usage. The states of the machine represent resource reservation tables. The final states in the automaton represent when the hardware pipeline clock advances to the next cycle. Because such an automaton can be very large, the authors use factoring to break the large automaton into several smaller automata based on categories, where the sum of all the states in the smaller automata is less than the number of states in the original automaton. A key to this approach is that these automata are built only once at compile-compile time and then used each time the compiler is run, thus reducing scheduling overhead.

Integral to instruction scheduling is modeling the instructions and resource constraints. These models provide the scheduler with the detailed knowledge needed to avoid stalls in the pipeline. The most common model is the resource vector. Ramanan and Govindarajan take a close look at several resource usage models, focusing on the level of abstraction, space, and time requirements [RG99]. They examine three standard approaches to managing resource usage: Conventional Reservation Tables, Reduced Reservation Tables, and the Automaton Approach. They introduce two new models: the Group Automaton and the Dynamic Collision Matrix. The Group Automaton is based on the finite state automaton approach but occupies less space by eliminating symmetry. The Dynamic Collision Matrix uses collision matrices instead of reservation tables. Ramanan and Govindarajan conclude that these new models do a better job balancing abstraction, space, and time requirements than previous methods.

### 2.2.1 Superoptimization

Massalin takes a different approach to scheduling with his Superoptimizer [Mas87]. The Superoptimizer accepts as input a machine language program and produces another, hopefully shorter, program as output. He uses an exhaustive method to find the shortest sequence of instructions that performs a given function. The approach is simple - use a subset of the machine's instruction set, and begin generating programs that start with length 1, then length 2, etc. until one is found that produces the correct results. The Superoptimizer is especially good at eliminating conditional jumps and replacing them with other instructions, resulting in significant improvement on pipelined machines. The Superoptimizer is also able to find interrelations between logical and arith-

metic instructions. The best use the author has found for it is as a tool for an assembly language programmer to come up with better sequences of instructions for small tasks. A shortcoming of the Superoptimizer is that it is limited to very short programs due to the exhaustive method.

Granlund and Kenner based their work off Massalin's Superoptimizer, and developed a system believed to be faster and more versatile [GK92]. Granlund and Kenner integrate the Superoptimizer into the GNU C compiler, forming the GNU Superoptimizer, GSO. GCC uses Davidson's Register Transfer List (RTL) to represent the actual instructions of the target machine. It then uses pattern matching with machine description files that contain information on how to generate code for the instructions. A key focus of the optimization done here is on eliminating jumps because they are costly, especially on pipelined machines. The best sequences are chosen using the following three criteria: using immediate operands, using fewest scratch registers, and putting output into the same register as input. GSO has goal functions such as equality tests, min, max and abs, among others. GSO works by searching for the sequences of instructions that implement a goal function.

Portability is accomplished by using generic operations and including all instructions supported by GSO on all supported machines. Support is added for a new machine by first defining any new generic operations present on this machine not previously supported. Then, the search routines are modified to reflect the operations present in the new machine. Finally, code must be written to output the operations in assembly language. This Superoptimizer is limited in a very similar way as Massalin's: the search space can be exponential. GSO tries to keep this number down by avoiding constants. Another possible shortcoming that the authors warn about is that neither their Superoptimizer nor Massalin's exhaustively test the code generated to make sure it is correct.

Many of the methods presented in this section rely on resource vectors or some other representation of the instruction scheduling constraints in order to function. However, the issue of obtaining the vectors is not examined. These vectors are often constructed by hand, but are commonly error prone because the information needed to construct them is inaccurate or difficult to

obtain. A better way to determine scheduling constraints is needed. The work done on the super-optimizers is fascinating in that it bypasses this need and but is impractical due to its exhaustive nature.

## 2.3 Automated Approaches to Instruction Scheduling

There have been several advancements made towards automating some steps of the instruction scheduling process. Some tasks have been basic, such as determining the range of accepted integer operands the machine will take as input, whereas others are more advanced, such as AI learning.

Mahler is an intermediate language designed by Wall and Powell to use with the Titan workstation [WP87]. The language is used as the machine description to hide details of the real machine from compilers. The Mahler source code is first translated into object code very simply, without trying to avoid stalls or fill branch delay slots. While this simple code is produced, the object module is annotated with information that the instruction scheduler later uses at link time. Registers are allocated based on how frequently variables are used. Variables local to procedures that are never used together can occupy the same register. The instruction scheduler tries to find places in the code where cycles are wasted and fills them with productive instructions. This implementation is different from many others in that no graph is used, rather conflicts are encoded by earliest and latest allowable time. Unconditional branches are easy to optimize; the instruction at the destination of the branch is filled into the delay slot and then the jump is set to point to one instruction ahead of that one. Conditional branches are optimized to execute a useful instruction two-thirds of the time. The Mahler process saves time by removing the need to program in assembly and removes the worry about delay branches from the programmer. Mahler has also been retargeted to several RISC processors.

Moss et al takes a very different approach to instruction scheduling by turning local instruction scheduling into a learning task in order to automatically create a scheduling algorithm [MUC+98]. The purpose is to remove the time and effort required to create a handwritten algorithm for instruction scheduling. The semantics for instructions are represented with a DAG. The

scheduler is trained to learn to select the best instruction from the list of instructions that could be executed next, using a process called supervised learning. This is done by comparing two instructions and determining which one is the best. This process is repeated, always maintaining the current best instruction found. The instruction pairs are compared using triples (P, Ik, Ij) where P is the partial schedule, and each I is the set of instructions to choose from. The triple belongs to the learning relation if the first instruction is preferable to the second, otherwise it does not belong to the relation. The worst shortcoming of this work was that a simulator determined all results so no actual timings were performed.

DERIVE is a tool developed by Engler and Hsieh that uses the existent system assembler to extract instruction encodings [EH00]. This method is simpler and less error prone than construction by hand that requires detailing the offsets, sizes, and values of instruction fields. DERIVE works in three steps. The first step is to take in the assembly language instruction as input and output a vector of C structures that specify the instructions whose encodings are desired. The second step is the DERIVE solver, which solves for each encoding and outputs a C structure that describes each encoding. The derive solver is made up of three solvers. The register solver determines the instruction's opcode mask and the location and size of each operand field. It begins by assuming all as for the opcode mask and then iterates over the instruction fields. Then it performs AND with the instruction, so that finally only 1s remain where they are set in all instructions. The registers are determined by watching the bits that change between 0 and 1. The immediate solver uses a similar method of cycling through the operands. The last solver is the jump solver that also uses a similar technique. The final step is an encoder generator that takes in the encoding specifications and outputs functions to generate the encodings.

Collberg's work on retargeting also aids in automatically determining a machine description [Col02]. His Architecture Discovery Tool (ADT) consists of five core components: the Generator, Lexer, Preprocessor, Extractor, and Synthesizer. A brief description of each follows. The Generator uses the native C compiler to generate a sample of C programs to compile into the native assembly language. Thus, a requirement of the ADT is a native C compiler. The Lexer then

analyzes the assembly code output by the compiler to determine the relevant instructions. The Lexer uses two methods to determine the type of syntax accepted by the assembler: Textual Analysis and Assembler Error Analysis. In the first the code produced by the compiler is scanned to extract information and in the second the assembler is used to determine information when things are accepted/rejected by the assembler. An example use of Assembler Error Analysis is to determine the accepted ranges of integer operands. The Preprocessor converts the information from the Lexer into data-flow graphs. The preprocessor also employs a redundant instruction elimination scheme and tries to detect implicit arguments. The Extractor takes the data flow graphs as input and produces a set of instruction semantics as output. One technique used is to exhaustively search combinations of operators and addressing modes and compare the results. Finally, the Synthesizer takes the instruction semantics as input and produces a machine description as output. A few shortcomings are that the ADT will fail if the assembly language of the machine is nonstandard, if the instruction set architecture has very complex semantics, or if the machine is heavily stack based.

Moss continues this learning approach in conjunction with McGovern and Barto [MMB02]. This work focuses on two methods to build instruction schedulers: rollouts and reinforcement learning (RL). The schedulers used only reorder instructions, they do not add or remove any instructions (i.e., nops are not added), and perform list scheduling. This work addresses the limitations found with supervised learning techniques, specifically the requirement of training the scheduler with already existing valid input/output pairs. The rollout scheduler tests each candidate to be added to a partial schedule and evaluates the running time. After each candidate is rolled out, the one with the best estimated running time is choose. The rollout scheduler examined has excellent performance and is able to outperform the commercial DEC compiler for the Compaq Alpha. However, the running time of $O(n^2m)$ where $n$ is the number of instructions and $m$ is the number of rollouts makes rollout techniques impractical for commercial compilers since their average running time is $O(n)$. The RL approach tries different actions and then learns from the consequences to determine which actions are best. The results were good; it performed

almost as well as the commercial DEC compiler. The authors experimented with combining reinforcement learning with rollout and found that for C benchmarks, they were able to beat the DEC compiler.

The work presented in this section demonstrate that partial automation of the instruction scheduling process is possible, but no complete system has yet been developed.

## 2.4 Timing Platforms

An additional issue in instruction scheduling is how to determine which schedule is the best. It could be the schedule with the fewest instructions, or more practically, the one that executes the fastest. To perform the timing necessary to determine which schedule is the fastest, a hardware specific timing platform is required. An examination of some platforms follows.

McVoy and Staelin describe lmbench, a benchmark suite that measures commonly found performance problems in system applications [MS96]. lmbench is extremely portable over UNIX systems. The package measures two factors: latency and bandwidth (the rate at which data is moved), because they strongly influence performance. lmbench measures the system's ability to transfer data, specifically between the processor, cache, memory, network, and disk. The timing mechanisms are implemented using the `gettimeofday()` method. Timing accuracy is gained by putting the operation to be measured in a loop and dividing the loop time by the number of times the loop executed. The benchmarks are run on a system in single user mode. The authors conclude that the memory subsystem is as important as processor speed, especially as processor speeds increase.

Brown and Seltzer make modifications to lmbench to create a new benchmark suite, hbench:OS [BS97]. Their modifications are designed to provide more consistency and rigor to the existing lmbench suite. A key issue they focus on revising is the timing methods. The authors note that if a system does not have or use a microsecond timer, the `gettimeofday()` function may be measured in units as large as 10ms. The authors made modifications including running the tests in internal loops and using the average time over all cases. These loops were made to run for at least one second so the timer resolution would not affect the results as much. They also use hooks into

the hardware cycle counters to do extremely accurate measuring when only one measurement can be taken. This method, however, is hardware-specific. The last change they made was to remove the overhead of the system call to `gettimeofday()` by measuring the time of this call and subtracting it from the total time of the measured. A few changes were made to the statistical output as well. The new system runs each test multiple times and then the best and worst 10% of the times are thrown out and the remaining are averaged. This eliminates the system overhead and the overly optimistic results that are not reflective of the system on average. They also have the benchmark perform an iteration before the timing mechanism is started to get all data needed loaded into the cache. Hbench:OS demonstrates how the timing mechanisms can be improved to produce even more accurate measurements.

McVoy and Staelin responded with new version of lmbench, version 2.0, to address all of the problems found by Brown and Seltzer with lmbench 1.0, including considering clock resolution, running the experiments multiple times in a loop, and now auto-sizing the duration of the benchmarks [MS98]. lmbench still uses the `gettimeofday()` function to compute the time intervals, but now includes a module to reduce the timing error to less than 1%. The timing and loop overhead are also removed from the final execution time. A new part of the suite is MHz, a portable ANSI/C program that measures the clock speed. MHz uses a GCD technique to find the clock speed by timing nine different expressions and assuming the result is an integral multiple of a single clock tick. Each instruction must be dependent on the one before it so that the machine does not use optimizations to overlap the instruction execution.

Baker proposes the idea of actually timing different instruction sequences to see which run the fastest on RISC architectures [Bak91]. He calls the technique used “scheduling through self-simulation.” A key insight to this is that the test program is timed on the same machine the program will be executing on, which leads to up-to-date information about the scheduling decisions for the specific processor. One shortcoming is that this method is not exhaustive, Baker proposes running only a small collection of instruction sequences to be timed. The timing is performed using a 16-bit counter attached to the processor oscillator, thus the architecture must support this.

Another architectural problem encountered is that the caches, registers, and pipeline must be initialized and loaded to the state they would be in when the instruction would execute.

The lmbench platform and its revisions note many of the difficulties encountered it timing instruction execution. Baker's work proposes the idea of actually timing instruction sequences to determine which sequence is the best. These works form the basis for my work in determining efficient instruction orderings.

# 3 Methods

A timing platform needs to be established to test the hypothesis that scheduling constraints can be derived from timing the instructions. Section 3.1 examines difficulties found in timing on the UltraSPARC and the solution needed. After a timing harness is in place, the next issue is to reduce the size of the search space consisiting of all instructions. Searching through all possible combinations of instructions is exponential, so a traversal mechanism needs to be implemented. Section 3.2 examines this problem.

## 3.1 Timing Issues

Timing is a sensitive issue for this project because the timing of individual instructions requires an extremely accurate timing system. The amount of time to execute one instruction is so minute that in order to detect small differences in timings, a timing system with very good resolution is needed. However, an important goal is to be architecture independent. The following subsections examine the techniques used and evaluate each in terms of accuracy and platform independence.

### 3.1.1 getTimeOfDay()

The first technique we use is the `getTimeOfDay` function provided by `UNIX` in sys/time.h. Documentation is available in the `Linux` man pages under `getTimeOfDay`(2). The function has the desirable property that it is extremely portable because it is available on most, if not all, `UNIX` machines, however it suffers from lack of precision. It records the time of day in microseconds since 00:00 Universal Coordinated Time (UTC), January 1, 1970 and stores the value into a `timeval struct`. Figure 5 shows code required for this instrumentation. The resolution of this timer is dependent on the architecture. Time can be measured in different units depending on the

architecture and is often not accurate to the microsecond level. Since the time for a single instruction to execute is properly measured in nanoseconds, this lack of precision will mask differences between instruction execution times. Another shortcoming of `getTimeOfDay` is that it is a system call, and so it introduces unwanted overhead. This overhead is added into the overall timing and is larger than the execution time for a single instruction. As a result, the overhead dominates the time making it difficult to differentiate between the execution times of particular instructions.

```
struct timeval startTV;
struct timeval stopTV;

void startTimer(){
  gettimeofday(&startTV, NULL);
}

void stopTimer(){
  gettimeofday(&stopTV, NULL);
}

long timeDiff(){
  long start_ms;
  long stop_ms;
  start_ms = startTV.tv_usec + startTV.tv_sec*1000000;
  stop_ms = stopTV.tv_usec + stopTV.tv_sec*1000000;
  stop_ms -= start_ms;
  return stop_ms;
}
```

Figure 5. Timing Harness for getTimeOfDay().

The most simple instruction to issue is *no operation*, called a *nop*. Issuing a nop causes a delay but does not change the state of the program. An initial test is run that times a variable number of nops and displays the results to standard output. Sample code for one nop is shown in Figure 6. The code is written in C++ with inline assembly, a technique described in [Kip02]. The volatile attribute is used to ensure that the compiler does not optimize or change the code in the asm statement. Table I shows the results of running this test using `getTimeOfDay`. The problems with inaccuracy that result from using the `getTimeOfDay` call are seen by the fact that two, three, and four nops seem to take the same amount of time to execute. Further, the results show one

microsecond variations when timing two and three nops. A one microsecond difference is quite significant at this level. The times for five nops are disturbing because they are much greater than the other times and exhibit even more variation. Upon more careful examination, we determine that output buffering is responsible for these unexpected results. A second test is run that removes the output from the code and stores the time values into an array. Table II shows these improved results. The output problem effecting the five nops has been fixe. We can now see that the results for timing five nops are now closer to the other values in the table as is expected. However, this timing mechanism still suffers from the coarse grained resolution of the timer. The times for one, three, four, and five nops are still varied by one microsecond.

```
startTimer();
asm volatile("nop");
stopTimer();
cout << timeDiff();
```

Figure 6. Timing one NOP using getTimeOfDay().

| Time for 1 nop (milliseconds) | Time for 2 nops (milliseconds) | Time for 3 nops (milliseconds) | Time for 4 nops (milliseconds) | Time for 5 nops (milliseconds) |
|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 84 |
| 2 | 1 | 1 | 1 | 87 |
| 2 | 1 | 2 | 1 | 83 |
| 2 | 1 | 1 | 1 | 296 |
| 2 | 1 | 1 | 1 | 86 |

Table I. Test 1 Implemented using getTimeOfDay.

| Time for 1 nop (milliseconds) | Time for 2 nops (milliseconds) | Time for 3 nops (milliseconds) | Time for 4 nops (milliseconds) | Time for 5 nops (milliseconds) |
|---|---|---|---|---|
| 2 | 1 | 0 | 2 | 3 |
| 2 | 1 | 1 | 1 | 2 |
| 3 | 1 | 1 | 1 | 2 |
| 2 | 1 | 1 | 2 | 3 |
| 2 | 1 | 1 | 1 | 3 |

Table II. Test 2 - Removing Output - Implemented using getTimeOfDay.

### 3.1.2 The UltraSPARC Tick Register

As hypothesized a hardware dependent technique is more accurate than a software based technique. Next we examine the UltraSPARC `TICK` register: a hardware based method for timing. The UltraSPARC is equipped with a 63-bit tick register that measures clock cycles [WG94]. The timing code that uses the `TICK` register is shown in Figure 7. The values of `begin` and `end` are held in registers `g1` and `g2`, respectively. This allows us to read the registers directly and access them from C code using the variable names. The `asm` statement directs the assembler to use the code passed as a parameter directly without modification. The `TICK` register is read, the nops are executed, and the `TICK` register is read again. The time to execute the code between the two register reads is determined by subtracting the values from the registers. Table III shows the results of the first test instrumented using the `TICK` register. The times shown are in clock cycles. Here we can see that there is a clear difference in the times required to execute each number of nops. Again, we can see the effects of output on the timing mechanism. Table IV shows the results obtained by removing the output. With the exception of one nop, the times seem to be steady and the variation observed using `getTimeOfDay` is eliminated. The times recorded for one nop show the effects of loading the i-cache. Elimination of this effect is examined in the next section.

```
register int begin asm("%g1");
register int end asm("%g2");

asm volatile("rd %tick, %g1");
asm volatile("nop");
asm volatile("rd %tick, %g2");
cout << end-begin << endl;
```

Figure 7. Timing one NOP using TICK register.

| Time for 1 nop (cycles) | Time for 2 nops (cycles) | Time for 3 nops (cycles) | Time for 4 nops (cycles) | Time for 5 nops (cycles) |
|---|---|---|---|---|
| 191 | 6 | 16 | 15 | 46 |
| 89 | 6 | 16 | 15 | 46 |
| 89 | 6 | 16 | 15 | 15 |
| 191 | 6 | 16 | 15 | 15 |
| 89 | 6 | 16 | 15 | 15 |

Table III. Test 1 Implemented using TICK.

| Time for 1 nop (cycles) | Time for 2 nops (cycles) | Time for 3 nops (cycles) | Time for 4 nops (cycles) | Time for 5 nops (cycles) |
|---|---|---|---|---|
| 191 | 7 | 7 | 13 | 8 |
| 89 | 7 | 7 | 13 | 8 |
| 191 | 7 | 7 | 13 | 8 |
| 89 | 7 | 7 | 13 | 8 |
| 89 | 7 | 7 | 13 | 8 |

Table IV. Test 2 - Removing Output - Implemented using TICK.

### 3.1.3 Eliminating The Effects Of Caching

The results shown for timing one nop in Table III and Table IV show the effects of a cache misses influencing timing. In the test code, the first test run is the one nop case. The results show one nop requiring more time than two, three, four, or five nops. The reason behind this is that the first time the code is run, a cache miss occurs, and the time it takes to load the cache is reflected in the timing code. Had the case of two nops been before the case for one nop, the effect of the cache would have been seen there instead. One way to remove the cache effects is to execute the code inside a loop and not time the first iteration. However, this introduces loop overhead into our timing, and it is overhead that we want to avoid. The best mechanism found to solve this problem is to unroll the loop and repeat the code, as shown in Figure 8. This loads the cache and avoids unnecessary compares and branches due to looping. However, we now are recording a series of results for each code fragment to be timed. The next question that arises is how to interpret these results.

```
for(int i = 0; i < 5; i++){
   asm volatile("rd %tick,
   %g1");
   asm volatile("nop");
   asm volatile("rd %tick,
   %g2");
   cout << end-begin << endl;
}
```

```
asm volatile("rd %tick, %g1");
asm volatile("nop");
asm volatile("rd %tick, %g2");
cout << end-begin << endl;
asm volatile("rd %tick, %g1");
asm volatile("nop");
asm volatile("rd %tick, %g2");
cout << end-begin << endl;
...
```

Figure 8. Loop Unrolling.

#### 3.1.4 Interpreting The Results

We know that if tests are run multiple times, the first iteration of each test should be discarded due to the effects of caching. The question that arises is, when a fragment of code is timed 10 times in a row after an initial set up execution, what time should be reported? One possibility is to average the results. Given a series of timings, several factors can skew the results. One that we have seen is the effects of caching. Another is a context switch when our process is switched out of the CPU. On the SPARC, alignment can cause increases in execution time due to the multiple dispatch nature of the architecture. Clearly, we do not want these effects included in our results. Therefore simply taking an average could be skewed. A better solution is to calculate the minimum value of all the runs. The key here is that it is not possible to take less time than the actual time for the code fragment to execute. All values greater than this minimum represent the cases where something additional has affected the timing.

#### 3.1.5 Implementing the Timing Platform

For all results shown in Section 5, timing is performed using the `TICK` register and the minimum value technique discussed in the previous section. These timings have been shown to be stable, accurate, and do not incur any overhead due to system calls.

### 3.2 The Search Space

Given an instruction set of $n$ instructions and a sequence of $k$ instructions to be scheduled, there are $n^k$ different ways to arrange the instructions. Thus, there are $n^k$ unique ways to schedule the instructions. Our approach to find the optimal schedule is to time the sequences to find the one that executes the fastest. Therefore the exhaustive search time will always be $O(n^k)$. Given a short sequence length, such as a pair of instructions to schedule, this is practical. However, as the length of the sequence and the size of the instruction set increase, the search space increases with exponential complexity and becomes intractable. One of these two factors needs to be reduced. However, to perform a task a certain number of instructions are needed. Thus, decreasing the size of the sequence is not the solution. Instead, a method is needed to decrease the size of $n$.

3.2.1 Categories

A natural way to reduce the number of instruction sequences that need to be considered is to break the instruction set up into groups called categories. An instruction is a member of a category if it exhibits similar scheduling characteristics. That is, given a partial schedule of instructions *S* and a category $C = \{i_1\ i_2\ i_3\ ...\ i_j\}$ where each $i_j$ is an instruction, the execution time for a schedule consisting of the original *S* followed by any element of *C* will be identical. Hence, if it can be determined that *j* instructions all belong to category C, then the number of timings is reduced from *j* to 1.

Two instructions will exhibit the same scheduling characteristics if they use the same resources during the same stages of execution. A example break down of categories is as follows: integer add/subtract, integer multiply/divide, floating point add/subtract, floating point multiply/divide, control transfer (branches, call, jump), and memory access (loads,stores). Figure 4 provides an example of three different instruction sequences where $C = \{add, sub\}$, $S = i_1 i_2 i_3$, and $i_1$ = `and %o6, %o2, %o3` $i_2$ = `rd %ccr, %o2` and $i_3$ = `mov 2, %o2`**.** Table V presents the timing for each sequence shown in Figure 9. Since `add` and `sub` both belong to the same category, when they are each added to the schedule *S*, the execution time of the schedule is the same. However, since `umul` does not belong to this category, adding it to *S* yields a different execution time.

# 4 Discussion

In this section, we examine the test environment and the impact of our study. All testing was done on a Sun Blade 100 workstation. The system features a 64-bit 500 MHz UltraSPARC IIe CPU running Solaris 8.

## 4.1 The Timing Harness

We developed a timing harness using the `TICK` register on the SPARC, as discussed in Section 3.1. The timing system takes as input an array of instructions are returns the execution time. It first generates a C++ file that contains the assembly instructions inlined, along with the code

```
Sequence 1:
      and %o6, %o2, %o3
      rd %ccr, %o2
      mov 2, %o2
      add %o4, %o5, %o1


Sequence 2:
      and %o6, %o2, %o3
      rd %ccr, %o2
      mov 2, %o2
      sub %o4, %o5, %o1


Sequence 3:
      and %o6, %o2, %o3
      rd %ccr, %o2
      mov 2, %o2
      umul %o4, %o5, %o1
```

Figure 9. Example Use of Categories.

| | |
|---|---|
| Sequence 1 | 12 Cycles |
| Sequence 2 | 12 Cycles |
| Sequence 3 | 25 Cycles |

Table V. Timings for Sequences in Figure 4.

required to perform the timing. The timing is done by executing the sequence of instructions multiple times and returning the minimum time, for reasons discussed in Section 3.1.4. The generated file is compiled, run, and its output is written to a log file. Finally, the log file is then read back into the timing system to output the result. Figure 10 shows an overview of the system. The code for the timing harness is given in Appendix A.

### 4.2 Timing in Instruction Scheduling

We wish to use the timing platform above to aid in instruction scheduling. Such a system would choose the optimal sequence of instructions by timing different sequences and finding the one that executes the fastest. Instruction scheduling is described in more detail in the next section. Finally, we describe how to integrate timing into instruction scheduling and discuss problems of efficiency and effectiveness.

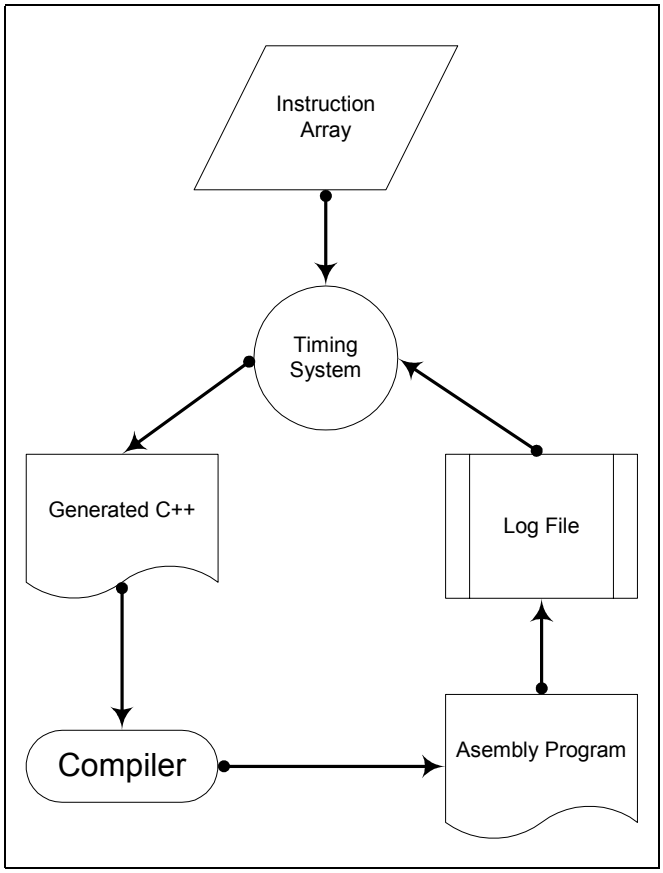


Figure 10. The Timing System.

4.2.1 Instruction Scheduling Background

The instruction scheduler in a compiler must rearrange the instructions it is given to produce a schedule with good execution speed that maintains program correctness. A RAW data dependency is formed when an instruction requires the result of the previous instruction. Consider the following code excerpt and its assembly counterpart.

```
(1)   i++;       add %r1, 1, %r1  // r1 = i
(2)   j = i;     mov %r1, %r2     // r2 = j
(3)   k = 2;     mov 2, %r3       // r3 = k
```

Here, (2) requires the result of (1). If *i* had the initial value 0, then after this code is executed *i*=*j*=1 and *k* = 2. To preserve program correctness the scheduler must ensure that (2) always exe-

cutes after (1) and that no instruction that modifies *i* is inserted between (1) and (2). Consider the following permutation of the instructions:

```
(2)   j = i;      mov %r1, %r2     // r2 = j
(1)   i++;        add %r1, 1, %r1  // r1 = i
(3)   k = 2;      mov 2, %r3       // r3 = k
```

Assuming the same initial values as above, after running this code $j = 0$, $i = 1$, $k = 2$. Clearly, program correctness was not preserved. A scheduler should not produce this schedule.

However, it is often more efficient to schedule another instruction in between (1) and (2) since (2) must stall and wait for the result of (1). Since (3) does not modify *i* it is a good candidate. The proper schedule would be as follows:

```
(1)   i++;        add %r1, 1, %r1  // r1 = i
(3)   k = 2;      mov 2, %r3       // r3 = k
(2)   j = i;      mov %r1, %r2     // r2 = j
```

In this case, after execution $i = j = 1$ and $k = 2$, as in the original schedule. Therefore, this program is correct. It wins over the original schedule since instead of stalling while (1) finishes, (3) enters the pipe and begins executing.

To perform scheduling, a scheduler uses three sets of instructions: the *partial schedule*, the *ready set*, and the *not ready set*. The partial schedule consists of the instructions already scheduled. The ready set contains all instructions that, if executed next, would preserve program correctness. The not ready set contains all instructions not scheduled and not in the ready set. Initially, the partial schedule is empty. The scheduler picks an instruction from the ready set, adds it onto the partial schedule, and updates the ready and not ready sets. This process continues until all instructions have been scheduled. If the ready set is empty and all instructions have not yet been scheduled (i.e. the not ready set contains instructions), the scheduler schedules nops until the ready set is non-empty.

### 4.2.2 Using Timing

The timing harness discussed above can be used in a scheduler. A brief overview of such a system follows. Given a category listing and a ready set, the ready set can be partitioned into categories. Then the last instruction (or last few instructions) from the partial schedule, followed by one instruction representative of each category in the ready set, is timed. As more instructions from the schedule are timed along with the candidate instruction, the more accurate the schedule becomes, due to *instruction grouping*, as discussed next. However, including more instructions will also cause the timing to take more time, which is addressed in the next section. It is not necessary to time the schedule followed by each instructions in the ready set, since multiple instructions from the same category will have the same timing results. From the resulting execution times, the best category (or categories) to be scheduled next is determined based on the minimum execution time. In the ideal case, there will be a category from which instructions can be added to the schedule without causing a pipeline stall, resulting in minimum execution time. Otherwise the category that causes the least amount of stall will be selected. Finally, the ready set and the category are compared. If there is only one instruction in the ready set from the optimal category, it should be scheduled. Otherwise, any instruction in the ready set that is part of the optimal category can be chosen.

### 4.2.3 Efficiency

The instruction scheduling timing method could be effective, although very slow, if used at compile time. For every instruction that needs to be scheduled, the system has to time a representative from all categories in the ready set following the instruction or instructions from the partial schedule. Even if the number of categories represented by the ready set is frequently small and only one instruction is used from the partial schedule, scheduling a full size program would require a large amount of time. A system needs to be devised that can perform the timings offline once and produce an output that can be integrated into the compiler to use every time it schedules. Such a system is examined in the future work section of this paper.

#### 4.2.4 Effectiveness

An aspect of the instruction scheduling timing technique that needs to be analyzed is how effective it is in choosing a good schedule. Scheduling is done without lookahead and with minimal information about what preceded the instruction, which is required in order to choose the optimal schedule. Without knowing what lies ahead, the scheduler can choose the best instruction to schedule at that time, but it may not be the best to schedule considering what comes next. This is further complicated by *instruction grouping.*

#### 4.2.5 Instruction Grouping

The UltraSPARC is a superscalar processor that can execute up to four instructions per cycle [Sun97]. The instructions issued together are known as a group. The UltraSPARC has many grouping rules, a few are listed here in order to gain an understanding of instruction grouping: integer instructions, loads, and stores can only be issued from the first three slots; floating point instructions, branches, and nops can be issued from any slot; only one load/store can be issued per group; only one control transfer instruction can be issued per group. Grouping is also dependent on data dependencies. An instruction will not be grouped with another if a RAW or WAW dependency exists between them. Scheduling without lookahead may create a schedule that contains many partially filled groups near the end of the schedule due to the grouping rules. If it could be observed via lookahead that placing an instruction into the schedule would cause a later group to issue only partially full, and that a different instruction being inserted into the schedule would produce better results later in the schedule, then the scheduler could be optimized.

## 5 Conclusions

We have shown that scheduling constraints are an observable phenomenon that can be realized via a high precision timing mechanism. We have considered several timing mechanisms, such as the platform independent `getTimeOfDay` function and the hardware dependent `TICK` register on the UltraSPARC. We conclude that the `getTimeOfDay` function is too coarsely defined to be accurate in this setting, but hope that someday a hardware independent and accurate clock timing mechanism will be available. We present a hardware dependent timing platform to time sequences of instructions using the `TICK` register that reveals observable differences in exe-

cution time to be used in instruction scheduling. We consider the size of the search space and make observations on how to reduce the size using categories.

This research provides hope that a platform independent version of the timing system can be integrated into a compiler that would produce a fairly good schedule. This would be done automatically, given as input only the instruction set and possibly some category information. Such a system would eliminate the time required to produce handwritten schedulers and allow a good scheduler to be available almost effortlessly after a new architecture is introduced. Simulators are another area in which accurate knowledge of the timing characteristics of instructions can be used to make simulation more realistic. Future work involves integrating the timing harness presented here into a real life scheduler to evaluate its effectiveness.

# 6 Future Work

The area that remains to be examined is the actual integration of such a timing system into an instruction scheduler. As noted in the discussion, the system should perform all the timings at compile-compile time in order to have reasonable compile times. Using only the last instruction from the partial schedule and no lookahead or rollbacks, it is hypothesized that this could be done using a weighted graph where the nodes reflect the instruction categories. The weight of the edge from `category A` to `category B` is the amount of time it takes to execute an instruction representative of `category A` followed by a representative instruction of `category B`. The graph can be constructed offline at compile-compile time and used to perform instruction scheduling quickly compile time. The scheduler would simply look at the category the last instruction already scheduled belongs to and compare the weights corresponding to all categories in the ready set. The edge with the minimum weight would be chosen. Ties would be broken arbitrarily.

The weighted graph technique can be expanded to have the nodes retain the last few instruction scheduled in order to achieve even more efficient schedules. Lookahead and rollback could also be incorporated into this framework.

## 7 Acknowledgements

I would like to thank my advisor, Mark Bailey, for his guidance and support during this project and for the past four years. My fellow students Chris LaRosa and Andrew Magyar have both helped tremendously improve this work by editing. Clark Coleman at the University of Virginia gave some useful insight into timing mechanisms. Thank you to everyone who has put up with me this final week and listened to be whine about writing. And finally, to the entire Computer Science Department, its been a great four years, thank you for everything.

# 8 Appendix A

```
#include<iostream>
#include<fstream>
#include<string>

using namespace std;

// Purpose : To generate a test program filled with the
//           instructions from the instr array, compile
//           the test program, run the test program, and
//           print out the timing results
int timer(string instr[], int length){
  int res; //Hold the timing result

  // Set up for file output
  ofstream outfile("test.cc"); // Open the file for writting
  assert(outfile); // Make sure file is good

  // Generate the test file
  outfile << "#include<stdio.h>" << endl;
  outfile << "int min(int x, int y){" << endl;
  outfile << "return (x < y) ? x: y;}" << endl;
  outfile << "int main(){" << endl;
  outfile << "register int begin asm(\"%g1\");" << endl;
  outfile << "register int end asm(\"%g2\");" << endl;
  outfile << "int mini;" << endl;
  outfile << "const int MAX = 50;" << endl;
  outfile << "int results[MAX];" << endl;
  // Run the test MAX times
  outfile << " for(int i = 0; i < MAX; i++){" << endl;
  // Start timer
  outfile << "asm volatile(\"rd %tick, %g1\");" << endl;
  // Insert instructions from array into test space
  for(int i = 0; i < length; i++){
    outfile << "asm volatile(\"" << instr[i] << "\");" << endl;
  }
  // End timer
  outfile << "asm volatile(\"rd %tick, %g2\");" << endl;
  // Store result into array
  outfile << "results[i] = end-begin;" << endl;
  outfile << "}" << endl;
  // Caclulate minimum time
  outfile << "mini = results[0];" << endl;
  outfile << " for(int i = 0; i < MAX; i++){" << endl;
  outfile << "mini = min(mini, results[i]);}" << endl;
  outfile << "printf(\"%d\",mini);" << endl;
  outfile << "}" << endl;
  // Close the file
  outfile.close();
```

```
  // Compile this new file
  system("gcc -S test.cc");
  system("/usr/ccs/bin/as -xarch=v8plus -o test.o test.s");
  system("gcc -o tester test.o");
  // Run it and write result to log file
  system("tester >> result.log");

  // Remove generated files
  system("rm test.*");
  system("rm tester");

  // Open result file
  ifstream result("result.log");
  // Read in result
  result >> res;
  // Remove log file
  system("rm result.log");
  return res;
}
```

# 9 References

[Bak91] Henry G. Baker. Precise instruction scheduling without a precise machine model. *ACM SIGARCH Computer Architecture News*, 19(6):4–8, 1991.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *Proceedings of the conference on Programming language design and implementation*, pages 229–240. ACM Press, 1991.

[BR91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the conference on Programming language design and implementation*, pages 241–255. ACM Press, 1991.

[BR95] Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 46–56. IEEE Computer Society Press, 1995.

[BS97] Aaron B. Brown and Margo I. Seltzer. Operating system benchmarking in the wake of lmbench: a case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 214–224. ACM Press, 1997.

[Col97] Christian S. Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 57–70. ACM Press, 1997.

[Col02] Christian S. Collberg. Automatic derivation of compiler machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):369–408, 2002.

[Dav86] Jack W. Davidson. A retargetable instruction reorganizer. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 234–241. ACM Press, 1986.

[EH00] Dawson R. Engler and Wilson C. Hsieh. Derive: a tool that automatically reverse-engineers instruction encodings. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 12–22. ACM Press, 2000.

[GK92] Torbjorn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *Proceedings of the 5th ACM SIGPLAN conference on Programming language design and implementation*, pages 341–352. ACM Press, 1992.

[GM86] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN symposium on Compiler construction*, pages 11–16. ACM Press, 1986.

[Kip02] Harald Kipp. *GCC-AVR Inline Assembler Cookbook*. egnite Software GmbH, version 1.6 edition, 2002.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 122–126. IEEE Computer Society Press, 1987.

[MMB02] Amy McGovern, Eliot Moss, and Andrew Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning, Special Issue on Reinforcement Learning*, 49(2/3):141–160, 2002.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[MS98] Larry McVoy and Carl Staelin. mhz: Anatomy of a micro-benchmark. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.

[MUC+98] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[RG99] V. Janaki Ramanan and R. Govindarajan. Resource usage models for instruction scheduling: two new models and a classification. In *Proceedings of the 13th international conference on Supercomputing*, pages 417–424. ACM Press, 1999.

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient dag construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 93–102. ACM Press, 1991.

[Sun97] Sun Microsystems, Palo Alto, Ca. *UltraSPARC User's Manual*, July 1997.

[WG94] David L. Weaver and Tom Germond. *The SPARC Architecture Manual*. SPARC International, version 9 edition, 1994.

[WP87] David W. Wall and Michael L. Powell. The mahler experience: using an intermediate language as the machine description. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 100–104. IEEE Computer Society Press, 1987.