Instruction Scheduling and Executable Editing

by Eric Schnarr and James R. Larus

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
(608) 262-9519
{schnarr,larus}@cs.wisc.edu

Abstract

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. The resulting disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost instrumentation for measurement, simulation, or emulation. Instrumentation code that executes in previously unused processor cycles is effectively hidden. On two superscalar SPARC processors, a simple, local scheduler hid an average of 13% of the overhead cost of profiling instrumentation in the SPECINT benchmarks and an average of 33% of the profiling cost in the SPECFP benchmarks.

1. Introduction

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. For example, the most recent generation of RISC chips—the Digital Alpha 21164, IBM/Motorola Power PC 620, SGI R10000, and Sun UltraSPARC—are 4-way superscalar processors that execute up to four independent instructions in a single cycle. Even recent high-end x86 processors—the Intel Pentium Pro and AMD K5—are 3-way superscalars. Unfortunately, exploited instruction-level parallelism lags far behind the hardware capacity. Cvetanovic and Bhandarkar found that programs running on a 2-way superscalar Digital Alpha 21064 could dual issue only 20–50% of their instructions, which means that in 67–90% of cycles, only one instruction executed [3]. Similarly, Diep et al. found that on a 4-way superscalar Power PC 620, four integer SPEC benchmarks completed an average of 1.05–1.25 instructions per cycle and three floating-point SPEC benchmarks completed an average of 1.0–1.9 instructions per cycle [4].

This large disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost or even no-cost instrumentation for measurement, simulation, or emulation. Scheduling reduces instrumentation's perceived cost by putting instrumentation instructions in previously unused processor cycles. Instrumentation code that executes in unused cycles is effectively hidden.

Program instrumentation has been used for many purposes, including performance measurement, computer architecture simulations, and software fault isolation and error detection [1]. Although direct instrumentation typically incurs lower cost than alternative approaches, the increased program running time caused by instrumentation is occasionally a limiting factor and is always an annoyance. At one extreme, in parallel or real-time systems, instrumentation overhead can perturb system behavior by introducing time dilation. Even for less demanding applications, the cost of program profiling or error detection is currently too high for production codes. Previous instrumentation systems expended considerable effort to formulate efficient instrumentation code sequences [11], place them parsimoniously [2], and insert instrumentation without affecting a program's behavior [9]. However, few systems attempted to exploit instruction-level parallelism by scheduling the instrumentation code into a program.

This paper describes a simple instruction scheduler that we added to the EEL executable editing library [10]. We applied the scheduler to a common program instrumentation, which profiles basic block execution frequencies using a four-instruction sequence inserted into almost every basic block. On the 3-way superscalar SuperSPARC processor, scheduling hides an average of 11% of the SPECINT overhead and an average of 44% of the SPECFP overhead. On the 4-way UltraSPARC scheduling hides an average of 15% of overhead for the SPECINT benchmarks, but only 17% of overhead for the SPECFP benchmarks. This is the because EEL produces a worse schedule than was originally found in these highly optimized programs. Factoring out the differences in scheduling algorithms, we find that the integer benchmarks remain about the same (13%), while scheduling profile instrumentation in the SPECFP benchmarks hides 27%. In the future, these results may improve, and scheduling become even more attractive, with a more accurate and aggressive instrumentation scheduler and wider microarchitectures that offer further opportunities to hide instrumentation.

To implement instruction scheduling, we extended EEL with the specific details of a processor's microarchitecture. EEL records this and other architectural information using a concise, high-level specification describing the machine's instruction set architecture. A tool called Spawn [10] translates these specifications into executable C++ code, which becomes the part of EEL that decodes and interprets machine instructions. As part of this work, we extended the architecture specification language to capture salient features of a machine's microarchitecture and used this information to drive an instruction scheduler in EEL. This paper first describes this language extension and how EEL uses the information to schedule instructions. Then some timing measurements are given for scheduling instrumentation into the SPEC95 benchmarks.

The paper is organized as follows. Section 2 describes related work. Section 3 presents the extensions to Spawn and shows how they represent the information necessary for instruction scheduling. Section 4 describes our experiments on scheduling program instrumentation and presents the results. Finally, Section 5 concludes.

2. Related Work

This paper extends several areas of previous work. Patil and Fischer used a different form of parallelism to hide instrumentation overhead. On a multiprocessor, the ran on a second processor the code to check pointer and array accesses [12]. The additional processor reduced the perceived overhead of error checking by 2–55%. Our approach is more widely applicable since instruction-level parallel processors are, or soon will be, ubiquitous and because instruction-level parallelism permits a tighter coupling between program and instrumentation.

Proebsting and Fraser described an efficient algorithm for detecting structural hazards in single pipeline processors and a language for concisely describing these machines [13]. Our description language is more verbose, but it also captures the syntax and semantics of machine instructions and scheduling constraints for superscalar machines. Our algorithm is also more general, if less efficient, since it works for superscalar processors as well as single pipeline machines.

Gyllenhaal described the machine description language (HMDES) used in the Illinois IMPACT compiler [6]. This language, like Spawn's description language, describes instruction encoding and scheduling constraints. HMDES describes instructions from several perspectives, which together provide the basic information needed by instruction schedulers: instruction latencies and resource usage. Spawn represents this information more concisely as a single semantic expression for a group of instruction, which describes when these instruction acquire and release microarchitectural resources. Spawn descriptions also capture instruction semantics and binary instruction encodings.

Schuette detected processor errors using unused instruction slots on a VLIW processor [15]. His control-flow monitoring technique inserts check operations into unfilled VLIW instructions. This is similar to EEL's rescheduling of instrumentation, except that EEL also reschedules the original instructions and can handle many different forms of instrumentation. Because a VLIW has fixed size instructions, Schuette's instrumentation did not increase code size. Whereas on a superscalar machine, instrumentation code has a secondary effect of reducing performance by increasing instruction cache misses.

3. Spawn Extensions

Any executable editing tool depends on an accurate description of a machine's instruction set architecture. While most parts of EEL are machine independent, it still must disassemble, analyze, and modify binary machine instructions. Past experiences with executable editing tools demonstrated that the code that manipulates binary instructions is simple, but tedious and difficult to debug. For example, in the profiling and tracing tool qpt [9], over 2,000 lines of hand-written C code exist to manipulate binary instructions. Subtle errors in this code were difficult to detect by inspection and often lay undiscovered for months before a new input executable exercised them.

To remedy this problem and make EEL more easily portable across different processors, it uses a concise description of a machine's instruction set architecture written in a high-level architecture description language. These descriptions are short—the SPARC is 333 lines—and similar in form to the descriptions often found in architecture manuals, which makes them easier to validate. Errors overlooked in a code review are also more likely to arise during testing, since a single description of an instruction's encoding and semantics underlies many different EEL instruction manipulation functions and different instructions often share common code in the description file.

Spawn [10] is the tool that converts an architecture description into the C++ code used by EEL. Spawn analyzes descriptions written in SADL (Spawn Architecture Description Language) to detect errors and extract the syntactic and semantic information needed by EEL. Spawn then reads an annotated C/C++ file and replaces its annotations by appropriate code produced using information extracted from the description. The resulting C/C++ file is compiled and linked into EEL to provide efficient functions for manipulating binary instructions.

Instruction scheduling requires more detailed information than the architectural descriptions that sufficed previously for EEL. In particular, scheduling requires a model of a machine's microarchitecture, especially its execution pipeline. Since a microarchitecture is specific to each particular processor implementation, this level of detail entails writing many more descriptions, so each description should be concise and easy to modify. To support instruction scheduling, we extended SADL to include pipeline timing and resource utilization information. This new information can be combined with the instruction semantic description to provide a complete map of an instruction's actions as it moves through a processor's execution pipeline.

Ideally, one would like to separate a microarchitecture-specific description from a general architecture description. SADL only partially accomplishes this goal by supporting functions that can encapsulate some of the timing and resource usage characteristics. The coupling of architectural and resource allocation information is necessary to permit Spawn to determine not only which registers are read from and written to, but also when values are read and when result values become available. On the other hand, our experience modeling the hyperSPARC, SuperSPARC, and UltraSPARC processors showed that the architecture semantics are easily carried over into a description of different microarchitectures, despite large changes in the pipeline description.

A (micro)architecture description, written in SADL, describes several aspects of a machine's architecture: instruction encodings, instruction semantics, architectural registers, and pipeline resources. The first three aspects (encodings, semantics, and registers) are described elsewhere [10]. Section 3.1 describes how pipeline resources and timing operators are combined with instruction semantics to encode details of an execution pipeline. Section 3.2 describes how an instruction scheduler uses this information to predict pipeline behavior.

3.1. Describing an Execution Pipeline

The example in figure 2 shows how the architectural semantics and microarchitectural timing and resource

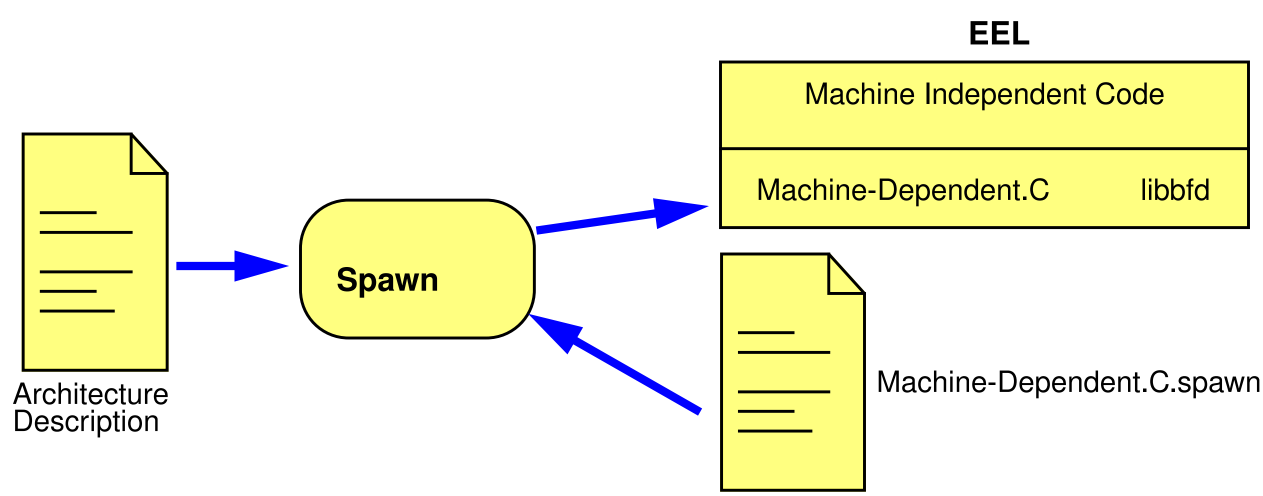

Figure 1: Spawn translates an architecture description written in SADL and an annotated C/C++ file into executable C++ code, which becomes the part of EEL that decodes and interprets machine instructions.

usage might be described for three ALU instructions on a ROSS hyperSPARC processor [14]. It starts by defining all the pipeline resources needed to model the timing behavior of the instructions. Second, the architectural registers are defined, along with aliases used to restrict the register types and incorporate resource usage information. Finally, the instruction semantics are defined along with their resource usage and timing characteristics.

Microarchitecture resources, such as register ports and ALUs, are described by a unit name and a value representing the number of copies of this resource in the processor. The ROSS hyperSPARC [14] described by this example is a dual-issue superscalar processor, with an ALU for arithmetic operations and an ALU for memory address calculations (LSU). The **ALUr** and **ALUw** units represent read and write ports to the register file for the arithmetic ALU. Similarly, the **LSUr** and **LSUw** units are ports used by the LSU. The values **multi** and **single** are used later to indicate if an instruction can be dual issued, or if it must execute by itself.

Architectural registers are described using the **register** declaration, and specifying the type and number of registers in the register file. This example declares a register file for the SPARC integer registers, where there are 32 registers each 32 bits wide. Aliases provide alternative ways to access the register file. They allow the registers to be accessed as an alternative type, and allow arbitrary SADL expressions to be combined with the register access. In this example, aliases are used to access the registers as 32 bit signed integers, and to specify the use of either an ALU read port or an ALU write port. The types are necessary later on in the description to disambiguate overloaded values and operators.

Description of the pipeline behavior of an instruction is combined with the instruction's semantic description. SADL commands **A**, **R**, **AR**, and **D** are used to describe when units are acquired and released and when the pipeline advances. The command **A** *<unit>* [*<num>*] acquires *<num>* copies of the unit, or stalls the pipeline if not enough copies of the unit are available. If *<num>* is omitted, it is assumed to be 1. **R** *<unit>* [*<num>*] releases *<num>* copies of a unit. The command **AR** *<unit>* [*<num>* [*<delay>*]] acts like the **A** command, but it also releases *<num>* copies of the same unit after the instruction executes for *<delay>* cycles. **AR** is handy for acquiring a resource for a fixed amount of time, without having to do explicit **R**'s later in the semantic expression. The command **D** [*<delay>*] advances the pipeline by *<delay>* cycles. In each cycle of an instruction's execution, Spawn applies all unit release events for the cycle first, followed by the all the unit acquire events. Extra delays are inserted if there are not enough free resources to accommodate all the unit acquire events.

```
// *** Define processor resources ***

// 2-way superscalar
unit Group 2

// flags for dual/single issue
val multi is AR Group, ()
val single is AR Group 2, ()

// ALU and LSU capacities
unit ALU 1, ALUr 2, ALUw 1
unit LSU 1, LSUr 2, LSUw 1


// *** Define registers ***

// general purpose registers (GPR)
register untyped{32} R[32]

// Alias for ALU read/write from GPR
alias signed{32} R4r[i]
   is AR ALUr, R[i]
alias signed{32} R4w[i]
   is AR ALUw, R[i]


// ** Define instructions ***

// Defining operators
val [ +         -
      &         |         ^     ]
 is (\op.\a.\b.
     A ALU, x:=op a b, D 1, R ALU, x)
  @ [ add32     sub32
      and32     or32      xor32 ]

// Defining shift operators
val [ <<        >>        >>    ]
 is (\op.\a.\b.
     A ALU, isShift, x:=op a b,
     D 1, R ALU, x)
  @ [ sll32     srl32     sra32 ]

// Get the second source operand
// or immediate value
val src2
 is iflag=1 ? #simm13 : R4r[rs2]

// Semantic description of the
// instructions add, sub, and sra
sem [ add       sub       sra   ]
 is (\op. multi,
     D 1, s1:=R4r[rs1], s2:=src2,
     R4w[rd]:=op s1 s2)
  @ [ +         -         >>    ]
```

Figure 2: Semantic description of the SPARC instructions add, sub, and sra. The resource usage and timing information are for the ROSS hyperSPARC microarchitecture.

An instruction scheduling algorithm must also know when registers are read from and written to. When an instruction reads a register, Spawn records in which cycle the read occurs. Writes are more difficult, since most pipelined implementations forward values between instructions [7]. When a value is computed, Spawn remembers the cycle in which the computation finished. When this value is written back into a register, Spawn records the cycle in which the value was computed, not when the register assignment took place. Therefore in SADL, the computation of an instruction's result value must be computed in the cycle immediately preceding the cycle in which the value becomes available to the next instruction.

The example in figure 2 continues by describing the semantics and pipeline behavior of the instructions **add**, **sub**, and **sra**. This is accomplished using **val** declarations which act like macros, and **sem** declarations which bind semantic expressions to instruction mnemonics. The effect of the **sem** statement in this example is to bind the instructions **add**, **sub**, and **sra**, to their semantic expressions which: (1) restrict the pipeline to at most 2 simultaneous instructions and advance the pipeline by one cycle; (2) acquire one or two ALU read ports and read the register values; (3) acquire the ALU functional unit and compute the instruction's result value; (4) advance the pipeline by one cycle and release the ALU read ports and the ALU functional unit; (5) acquire an ALU write port and update the destination register; and (6) advance the pipeline by one cycle and release the ALU write port. From this description, Spawn infers that these instructions can be dual issued, execute in 3 cycles, read their operands in cycle 1, produce a value at the end of cycle 1 that subsequent instructions can use, and update the register file in cycle 2. Note that an extra cycle was put before reading the registers, since the **sethi** instruction produces a value which is available at the end of cycle 0, and can be used by another instruction issued in the same cycle.

Given a machine description written in SADL, Spawn analyzes the instruction encodings, semantics, and timing. Instructions with identical timing and resources allocation patterns are grouped together to save space in the generated C++ code. Each group records the number of cycles it takes for a member instruction to pass completely through the pipeline, the resources acquired in each cycle, and the resources released in each cycle. Spawn also associates a cycle number with every access to the integer or floating point register file, for both reads and writes. For reads, this number indicates the pipeline execution cycle when the read occurs. For writes, it indicates when the written value actually becomes available to subsequent instructions, not when the value is written to the register file.

3.2. Predicting Pipeline Behavior

The key metric used by EEL's instruction scheduler is the number of cycles that the next instruction must wait before entering the execution pipeline. SADL describes the resource usage and timing for individual instructions. This information can describe the execution behavior of many superscalar processors executing a straight-line sequence of instructions.

Spawn passes this information to an instruction scheduler by filling in annotations in the C++ function, **pipeline_stalls**, which given a sequence of instructions, computes when the next instruction can start execution (see

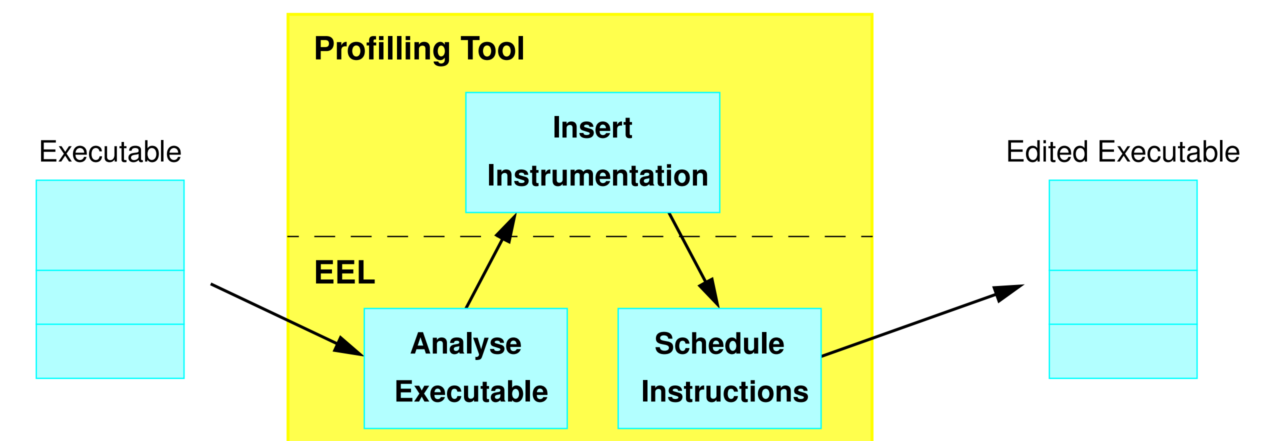

Figure 3: EEL schedules instrumentation by first analyzing the executable and allowing the application specific code (e.g., the program profiling tool) to select and place instrumentation. A new executable is then generated, which includes the instrumentation instructions. Scheduling is performed on each basic block as it is laid out in the new executable, causing the original and new instructions to be scheduled together.

Appendix A for an overview of this function). This function starts with a representation of the microarchitecture pipeline state generated by previous instructions in the instruction sequence. The pipeline state includes history information, such as the last cycle in which each register was read and written and which units are currently acquired by previous instructions. **pipeline_stalls** uses this state information as it simulates the pipeline execution of the new instruction and computes how many stall cycles are needed to satisfy the RAW, WAR, WAW dependencies and structural hazards.

The Spawn microarchitecture models can describe only a subset of all the factors affecting the integer and floating point pipelines. Our goal has been to describe actual machines rather than hypothetical systems, so for simplicity, the Spawn descriptions only model the execution pipelines themselves. The descriptions contain no information about a processor's memory interface to instruction prefetching, write buffering, or instruction and data cache behavior. **pipeline_stalls** does not compute stalls due to these mechanisms. On the other hand, few scheduling algorithms take these features into account since their behavior is data-dependent (c.f. [8]). In addition, SADL does not yet describe out-of-order execution, since it was not needed for the descriptions produced so far.

4. Scheduling Instrumentation

EEL schedules instructions in a basic block (local scheduling). The instructions in the block come either from the original program (for rescheduling) or a combination of the program and instrumentation code. If instrumentation contains branches, the scheduler only processes the regions of straight-line code. The scheduler uses a common two pass list scheduling algorithm [7]. The first pass starts at the end of the block and works backwards to compute the length (in cycles) of the dependence chain between every instruction and the end of the block. This computation only considers the stalls required between data dependent instructions.

The second pass starts at the beginning of the block and works forward, to order instructions with list scheduling. The instruction with the highest priority of any instruction that can be legally scheduled at this point is put next in the schedule. An instruction's priority is determined primarily by how few stalls it requires before it can start execution (as computed by **pipeline_stalls**). If two instructions require the same number of stalls, the instruction farthest from the end of the block, using the metric computed in the first pass, is scheduled first. If two instructions still have the same priority, the instruction listed earlier in the original code sequence is chosen under the assumption that the instructions were previously scheduled.

When computing data dependencies in both passes, the scheduler conservatively assumes that loads and stores from the original code access the same address. We also assume that loads and stores in instrumentation code access the same address, which differs from the address accessed by original instructions. This permits instrumentation loads and stores, which typically do not conflict with the original loads and stores, more freedom of movement. Since some instrumentation's memory references are more constrained, there are options to limit the movement of instrumentation code.

4.1. Limitations on Hiding Instrumentation

On aggressive superscalar machines, one could hope that all instrumentation code could be hidden in unused pipeline stall cycles. Unfortunately, processor limitations, such as memory latency (a load on the hyperSPARC has a one cycle latency) and resource usage (stores on the hyperSPARC use the LSU for 2 cycles and loads use it for 1 cycle), limit the cycles in which to hide instrumentation. A further problem is that in many programs, most basic blocks are short and so present few opportunity to hide instrumentation. Even when aggressively optimized, the SPEC95 integer benchmarks have average dynamic block size of 2.9 instructions.

Finally, scheduling instrumentation does not reduce instruction (or data) cache misses caused by instrumentation, since the additional instructions increase the code size regardless of how few stalls the program incurs. Lebeck and Wood proposed a model for the instruction cache effects of program instrumentation, which reasonably accurately predicted that instrumentation that increases a program's size by a factor of E, will increase cache misses by $E \times \sqrt{E}$ [11]. Profiling increases a program's text size by a factor of 2–3. Fortunately, many programs have low instruction cache miss rates, so the increase is not significant.

4.2. Scheduling Profiling Instrumentation

We scheduled QPT2's slow profiling instrumentation [2], which adds 4 instructions—set immediate, load, add, and store—into most basic blocks in a program. This code can execute in 4 cycles on both SuperSPARC and UltraSPARC processors. Blocks with a single instrumented single-exit predecessor or a single instrumented single-entry successor are not instrumented. The SuperSPARC experiments ran on dual processor Sun SPARCstation 20 equipped with 50Mhz SUN SuperSPARC [17] processors, running Solaris 2.4. The UltraSPARC experiments ran on an 12-slot Ultra Enterprise 4000/5000 with 167Mhz Sun

Benchmark	Avg. BB Size	Uninst. Time	Inst. Time	Sched. Time	% Hidden
099.go	2.9	739.2	1830.7 (2.48)	1582.4 (2.14)	22.7%
124.m88ksim	2.2	432.8	1208.2 (2.79)	1081.4 (2.50)	16.4%
126.gcc	2.2	305.9	833.4 (2.72)	798.7 (2.61)	6.6%
129.compress	3.0	278.9	523.8 (1.88)	482.6 (1.73)	16.8%
130.li	2.0	395.3	856.4 (2.17)	760.8 (1.92)	20.7%
132.ijpeg	6.2	438.0	678.7 (1.55)	646.8 (1.48)	13.3%
134.perl	2.4	428.3	1025.1 (2.39)	963.0 (2.25)	10.4%
147.vortex	2.1	538.9	1224.0 (2.27)	1136.3 (2.11)	12.8%
CINT95 Average	**2.9**		**2.28**	**2.09**	**14.8%**
101.tomcatv	13.8	310.1	360.9 (1.16)	354.1 (1.14)	13.4%
102.swim	49.0	447.4	471.5 (1.05)	532.8 (1.19)	-255.0%
103.su2cor	10.2	315.7	368.6 (1.17)	357.9 (1.13)	20.2%
104.hydro2d	4.7	608.8	805.3 (1.32)	724.8 (1.19)	41.0%
107.mgrid	32.4	582.7	643.7 (1.10)	579.2 (0.99)	105.8%
110.applu	12.5	471.8	566.6 (1.20)	541.5 (1.15)	26.5%
125.turb3d	6.1	655.5	917.6 (1.40)	907.3 (1.38)	3.9%
141.apsi	10.4	312.6	384.6 (1.23)	375.8 (1.20)	12.2%
145.fpppp	33.9	869.5	960.2 (1.10)	955.6 (1.10)	5.0%
146.wave5	10.9	362.4	375.9 (1.04)	376.3 (1.04)	-3.2%
CFP95 Average	**18.4**		**1.18**	**1.15**	**16.7%**

Table 1: Slow profiling instrumentation on the UltraSPARC. *Avg. BB Size* is the (dynamic) average basic block size (instructions). *Uninst. Time* is a program's un-instrumented execution time (seconds). *Inst. Time* is a program's instrumented, but unscheduled execution time. The number in parentheses is the ratio to the un-instrumented time. *Sched. Time* is the instrumented time after scheduling. Finally, *% Hidden* is the fraction of instrumentation overhead hidden by scheduling.

Benchmark	Avg. BB Size	Uninst. Time	Inst. Time	Sched. Time	% Hidden
099.go	2.9	741.1 (1.00)	1775.9 (2.40)	1582.4 (2.14)	18.7%
124.m88ksim	2.2	394.9 (0.91)	1185.6 (3.00)	1081.4 (2.74)	13.2%
126.gcc	2.2	306.6 (1.00)	824.7 (2.69)	798.7 (2.61)	5.0%
129.compress	3.0	273.2 (0.98)	522.8 (1.91)	482.6 (1.77)	16.1%
130.li	2.0	407.7 (1.03)	853.8 (2.09)	760.8 (1.87)	20.8%
132.ijpeg	6.2	449.9 (1.03)	687.9 (1.53)	646.8 (1.44)	17.3%
134.perl	2.4	431.6 (1.01)	1000.6 (2.32)	963.0 (2.23)	6.6%
147.vortex	2.1	532.5 (0.99)	1277.9 (2.40)	1136.3 (2.13)	26.6%
CINT95 Average	**2.9**		**2.29**	**2.12**	**13.2%**
101.tomcatv	13.8	321.0 (1.03)	363.2 (1.13)	354.1 (1.10)	21.5%
102.swim	49.0	510.6 (1.14)	543.8 (1.06)	532.8 (1.04)	33.0%
103.su2cor	10.2	310.5 (0.98)	370.5 (1.19)	357.9 (1.15)	21.1%
104.hydro2d	4.7	570.9 (0.94)	791.3 (1.39)	724.8 (1.27)	30.2%
107.mgrid	32.4	508.9 (0.87)	590.8 (1.16)	579.2 (1.14)	14.2%
110.applu	12.5	466.7 (0.99)	575.8 (1.23)	541.5 (1.16)	31.4%
125.turb3d	6.1	666.6 (1.02)	937.5 (1.41)	907.3 (1.36)	11.1%
141.apsi	10.4	319.5 (1.02)	401.1 (1.26)	375.8 (1.18)	31.0%
145.fpppp	33.9	885.6 (1.02)	1113.5 (1.26)	955.6 (1.08)	69.3%
146.wave5	10.9	352.8 (0.97)	376.4 (1.07)	376.3 (1.07)	0.0%
CFP95 Average	**18.4**		**1.22**	**1.16**	**27.3%**

Table 2: Slow profiling instrumentation on the UltraSPARC, with original instructions first rescheduled by EEL.

Benchmark	Avg. BB Size	Uninst. Time	Inst. Time	Sched. Time	% Hidden
099.go	2.8	1873.1	4695.1 (2.51)	4417.9 (2.36)	9.8%
124.m88ksim	2.3	1226.2	3003.2 (2.45)	2876.7 (2.35)	7.1%
126.gcc	2.2	863.4	2543.9 (2.95)	2466.8 (2.86)	4.6%
129.compress	3.0	1529.7	1751.3 (1.14)	1845.4 (1.21)	-42.5%
130.li	2.0	1066.3	2501.8 (2.35)	2101.6 (1.97)	27.9%
132.ijpeg	6.4	1153.8	1810.9 (1.57)	1716.7 (1.49)	14.3%
134.perl	2.3	1113.2	2187.8 (1.97)	2190.5 (1.97)	-0.3%
147.vortex	2.1	1721.7	4395.3 (2.55)	3900.4 (2.27)	18.5%
CINT95 Average	**2.9**		**2.19**	**2.06**	**10.9%**
101.tomcatv	11.4	1287.4	1420.2 (1.10)	1391.6 (1.08)	21.5%
102.swim	66.1	1280.0	2239.3 (1.03)	2214.7 (1.02)	41.5%
103.su2cor	10.1	1099.6	1385.3 (1.26)	1303.0 (1.18)	28.8%
104.hydro2d	4.4	2255.5	2760.5 (1.22)	2599.8 (1.15)	31.8%
107.mgrid	46.9	1481.2	1566.6 (1.06)	1628.5 (1.10)	-72.5%
110.applu	9.3	1661.3	2008.5 (1.21)	1853.6 (1.12)	44.6%
125.turb3d	5.7	1974.3	2858.9 (1.45)	2745.3 (1.39)	12.8%
141.apsi	11.8	911.2	1073.8 (1.18)	1020.7 (1.12)	32.6%
145.fpppp	28.2	2655.7	3916.2 (1.47)	3190.9 (1.20)	57.5%
146.wave5	13.3	1116.9	1466.4 (1.31)	1095.9 (0.98)	106.0%
CFP95 Average	**20.7**		**1.23**	**1.13**	**43.5%**

Table 3: Slow profiling instrumentation on the SuperSPARC.

UltraSPARC processors [16] running Solaris 2.5. The test programs were compiled with the Sun C and Fortran compilers (version 4.0), using the options "-fast -xO4 -xarch=v8 -xchip=super -xdepend -dn" and "-fast -xO4 -xarch=v8 -xchip=ultra -xdepend -dn" for the SuperSPARC and UltraSPARC respectively. We did not use the compiler options that generate UltraSPARC-specific code, since our instruction scheduler is currently configured for the SPARC version 8 instruction set. In all cases, we ran the programs using the scripts that come with the SPEC95 benchmarks, specifying the longest running ("ref") inputs.

Table 1 contains measurements for the UltraSPARC. Scheduling hides an average of 15% of slow profiling overhead for integer benchmarks, and 17% for floating point benchmarks. The integer programs execute many small basic blocks (average 2.9 instructions per block), so there is little opportunity to schedule added instrumentation among the original program instructions. This is compounded by the fact that for purely integer codes, the UltraSPARC can launch at most two instructions in parallel, instead of its maximum four instructions per cycle.

Neither of these problems should affect floating point programs, so it is at first surprising that they performed no better than the integer benchmarks. In fact, scheduling can hide a greater percentage of the instrumentation overhead for floating point programs, but performance is lost due to other factors. EEL's instruction scheduler is quite simple, as it only schedules within basic blocks using a simple heuristic algorithm. It does not perform as well as the optimizers in the SUN C and Fortran compilers that compiled the benchmarks. Since EEL schedules original program instructions as well as added instrumentation, it can produce a worse schedule for the original instructions while trying to hide the added overhead. Hence the benefit of hiding overhead is lost by de-scheduling in the floating point benchmarks, with their highly optimized, long basic blocks.

To factor out the effect of EEL's scheduler, we performed a variation of the previous experiment on the UltraSPARC. First EEL's scheduler reschedules the benchmarks, without adding any instrumentation. Then slow profiling instrumentation was added, but not scheduled among the original instructions. Finally both the added instrumentation and original instructions were scheduled together. Table 2 contains the results of this experiment. The results for integer benchmarks are the similar to before (average 13% hidden), but the results for floating point benchmarks are better. Scheduling now hides 27% of instrumentation overhead for the floating point benchmarks, with no significant outliers.

Table 3 contains measurements for benchmarks compiled for and run on the SuperSPARC, and the results are similar to those obtained for the UltraSPARC. Scheduling hides an average of 11% of profiling overhead for the integer benchmarks, and 44% of the overhead for floating point benchmarks.

5. Conclusion

This paper investigated the benefits of combining instruction scheduling with executable editing. Measurements on the SPEC95 benchmarks show that on a 3-way superscalar machine scheduling can hide an average of 11–44% of the overhead introduced by program profiling instrumentation. On a more modern, 4-way superscalar, the scheduler can hide an average of 16–17% of the profiling overhead. A sub-optimal scheduling algorithm limits the visible improvement from scheduling instrumentation, since it may reschedule original program instructions poorly.

Already, the benefits of scheduling program instrumentation are clear enough that existing and future instrumentation systems should adopt this simple technique to reduce instrumentation overhead. In addition, this approach promises to help reduce the cost of error checking, such as array bounds or null pointer tests, to a level at which it may routinely be included in production code.

Acknowledgments

Many thanks to Gurindar Sohi for the use of the UltraSPARC machine used to collect numbers for an early version of this paper, and to Bob Roessler for arranging the loan of the machine used in our initial UltraSPARC experiments. Mark Hill suggested that the scheduler model the L2 cache in the UltraSPARC. Vinod Grover and Kurt Goebel provided helpful comments on a draft of this paper.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation and Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

References

[1] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe, "Efficient and Language-Independent Mobile Programs," in *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127-136, May 1996.

[2] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, July 1994, pages 1319–1360.

[3] Zarka Cvetanovic and Dileep Bhandarkar. Characterization of the Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, April 1994.

[4] Trung A. Diep, Christopher Nelson, and John Paul Shen. Performance Evaluation of the PowerPC 620 Microarchitecture. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 163–174, June 1995.

[5] Linley Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9(2):9–15, February 16 1995.

[6] John C. Gyllenhaal. A Machine Description Language for Compilation. Master's thesis, Department of Electrical Engineering, University of Illinois, Urbana IL, September 1994.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[8] Daniel R. Kerns and Susan J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, June 1993.

[9] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[10] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[11] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 220–230, May 1995.

[12] Harish Patil and Charles Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG '95)*, St. Malo, France, May 1995.

[13] Todd A. Proebsting and Christopher W. Fraser. Detecting Pipeline Structural Hazards Quickly. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 280–286, Portland, Oregon, January 1994.

[14] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[15] Michael A. Schuette. Exploitation of Instruction-Level Parallelism for Detection of Processor Execution Errors. Ph.D. thesis, Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA, January 1991.

[16] SUN Microsystems, Inc. *UltraSPARC-I User's Manual*, August 1995.

[17] Texas Instruments. *SuperSPARC User's Guide*, October 1993.

Appendix A: Function pipeline_stalls

```
unsigned long
pipeline_stalls(unsigned long cycle,  // cycle when mi starts executing
                UnitValues &state,    // current pipeline state
                const mach_inst* mi)  // next instruction
{
    unsigned long stalls = 0;

  {{INST mi CATEGORY any::
    // All Spawn annotations now refer to instruction mi.

    unsigned long gid = {{GROUP}};    // mi's timing group
    long ii;

    // Trace[] records the resources used by this instruction in the current cycle.
    unsigned long trace[{{UNITS COUNT}}];
    for(ii=0; ii<{{UNITS COUNT}}; ++ii) trace[ii] = 0;

    // Search for stalls
    unsigned long mi_cycle = 0;   // current cycle in mi's pipeline
    while(mi_cycle <= {{GRP gid CYCLES}}) {

        // Units[] records the number of unused resources in this cycle
        // after allocating resources for all previous instructions.
        unsigned long* units = state[cycle];
        bool advance = true;

        // Test for structural hazzards.
        if(advance)
            for(ii=0; ii<{{GRP gid ACQUIRE mi_cycle COUNT}}; ++ii) {
                unsigned long unit_val =
                    units[{{GRP gid ACQUIRE mi_cycle UNIT ii}}] -
                    trace[{{GRP gid ACQUIRE mi_cycle UNIT ii}}];
                if(unit_val < {{GRP gid ACQUIRE mi_cycle NUM ii}}) {
                    advance = false;
                    break;
               }
           }

        // Test for RAW hazzards.
        if(advance)
            for(ii=0; ii<{{R READ COUNT}}; ++ii)
                if({{R READ ii TIME}} == mi_cycle &&
                   cycle < state.write_cy[0][{{R READ ii}}]) {
                    advance = false;
                    break;
               }

        // Similar tests for WAR and WAW hazzards.
        ...

        ++cycle;  // Advance the execution pipeline for previously scheduled instructions.

        // Advance instruction pipeline or record the stall
        if(advance) {
            for(ii=0; ii<{{GRP gid ACQUIRE mi_cycle COUNT}}; ++ii)
                trace[{{GRP gid ACQUIRE mi_cycle UNIT ii}}]
                    += {{GRP gid ACQUIRE mi_cycle NUM ii}};
            ++mi_cycle;
            for(ii=0; ii<{{GRP gid RELEASE mi_cycle COUNT}}; ++ii)
                trace[{{GRP gid RELEASE mi_cycle UNIT ii}}]
                    -= {{GRP gid RELEASE mi_cycle NUM ii}};
        } else
            ++stalls;
    };;
  }}

    return stalls;
}
```