Instructions Scheduling for Highly Super-scalar Processors*

Bill Appelbe, Raja Das, and Reid Harmon
College of Computing, Georgia Tech
Atlanta, GA 30332-0280
bill, raja, reid@cc.gatech.edu

Abstract

Super-scalar processors can execute multiple instructions out-of-order per cycle and speculatively execute instructions through branches. Such processors invalidate many of the assumptions of traditional instruction scheduling.

This article analyzes the impact of super-scalar processor architecture upon instruction scheduling. The compile-time schedule is shown to significantly impact performance, despite out-of-order execution. The problem of determining an optimal schedule at compile-time is shown to be NP-complete. A variety of heuristics for instructions scheduling are applied to benchmarks, and it is shown that traditional depth-first instruction scheduling performs badly compared to a variety of breadth-first instruction scheduling heuristics.

Modern high-performance microprocessors, such as the HP PA-8000[12] or the PowerPC 620[4], are all super-scalar, meaning that they can simultaneously fetch, decode, and execute several instructions at once. Current processors are often at least 4-way super-scalar. Together with the ability to execute multiple instructions in a single cycle, such highly super-scalar processors usually have several other characteristics that significantly impact compiler optimization and instruction scheduling, including:

Run-time Instruction Scheduling After instructions have been fetched and decoded, they are placed in an instruction reorder buffer awaiting execution. Instructions are executed out-of-order, meaning that an instruction in the instruction buffer is scheduled for execution when all of its operands are available.

Run-time Register Renaming Most instruction sets are limited to about 32 real registers. Reuse of registers thus creates anti-dependences: a write into a register can conflict with a preceding read. Register renaming eliminates such dependencies by internally mapping instruction set registers to a larger set of virtual registers. Each time a register is written, it is dynamically assigned a new virtual register, thus converting the program dynamically into a single-assignment form.

Speculative Execution The direction of conditional branches can be predicted at run-time, using a cache of previous branch directions. If the outcome of a conditional branch prediction is incorrect, all instructions subsequent to the branch are re-executed. Modern processors can speculatively predict through many branches and recover from an incorrect prediction in a single cycle.

Run-time Load/Store Dependence Checking Register renaming eliminates all but true dependences between register accesses when instructions are decoded. However, dependences between memory access may not be known until the effective address of the LOAD and STORE instructions are calculated at run-time. This would seem to imply that LOAD and STORE instructions must be executed in order. However, LOADs and STOREs can speculatively execute out of order, provided that such instructions are committed to memory in order, and runtime dependences are detected before instructions commit. If a runtime dependence violation is detected, then instructions are re-executed from the point where the dependence violation occurred as with mispredicted branches[10].

*This work supported in part by NSF grant ASC-9624087.
These advances in hardware have the goal of increasing the mean number of Instructions executed Per Cycle (IPC). The effect of some of these mechanisms is fairly obvious. For example run-time register renaming eliminates the need for a compiler to consider anti-dependences between register usage when scheduling instructions. However, the effect of other mechanisms is not so clear.

Intuitively, run-time instruction scheduling with a large instruction buffer and speculative execution might seem to reduce the need for compile-time instruction scheduling. The rational for this conjecture is that the processor should be able to find available parallelism, no matter what the compiler-generated order of instructions within a basic block. However, the analysis and benchmark results in this paper refute this conjecture.

The paper first analyzes run-time instruction scheduling for basic blocks and loops for an ideal $k$-way super-scalar processor (one which can execute $k$ instructions of any type in a single cycle). The analysis shows that the problem of determining an optimal schedule at compile-time is NP-complete, and suggests that breadth-first heuristics would perform better on programs with large basic blocks. Benchmark studies on kernels of the SPEC95 benchmarks confirm this, despite the overhead of spill-code introduced by breadth-first instruction scheduling.

Related Work

There exists a large body of work on instruction scheduling for super-scalar processors (e.g., [2, 3, 11, 14] However, almost all this research has been for “in order” execution of instructions. Instruction scheduling for the HP PA-8000, an out of order 4-way super-scalar processor is discussed in [5]. They show significant performance gains from a combination of sophisticated instruction scheduling and machine specific optimizations. However, it is not possible to draw any general conclusions about instruction scheduling for out of order super-scalar processors from their results as there was no detailed breakdown of the performance contributions of optimizations and instruction scheduling.

There has also been a considerable body of research into compilation for very Long Instruction Word (VLIW) architectures since the earliest VLIW architecture. However, VLIW architectures are also inherently in-order. There is considerable similarity between optimization and instruction scheduling for VLIW architectures and in-order super-scalar processors. For example, both seek to increase available Instruction Level Parallelism (ILP) by code motion across basic blocks and various optimizations to increase the size of basic blocks. By contrast, an out of order super-scalar processor with a deep window of decoded instructions would appear to be able to find available ILP across basic blocks without help from the compiler.

Register allocation and instruction scheduling are closely tied. Pinter [14] used a data structure called the parallel interference graph to do both register allocation and instruction scheduling simultaneously. Work with the Marion reargetable code generator [3] has shown that for in-order super-scalar processors an optimal schedule requires coupling between register allocation and instruction scheduling. By contrast, our results suggest that for a regular, highly super-scalar processor optimization to maximize ILP is preferable to optimization to minimize register spills. In other words, instruction scheduling takes precedence over register allocation.

Ideal Super-scalar Processors

In practice, all super-scalar processors have restrictions upon the type of instructions that can be executed in parallel, based upon hardware limitations and design tradeoffs. Hence, instruction scheduling for current super-scalar processors must take into account machine idiosyncrasies[5].

However, by analyzing the behavior of an ideal super-scalar processor, general conclusions can be drawn which should apply to production super-scalar processors.

The following analysis makes several assumptions:

- The execution time of any instructions is a constant number of cycles.
  In practice, some instructions take a variable number of cycles. In particular, LOAD instructions which result in cache miss may take many cycles to execute.

- Anti and output dependences between register uses and definitions can be ignored because of runtime register renaming.
A super-scalarity of $k$ means that $k$ instructions of any type can commence execution in any cycle.

Suppose we have a run-time dependence graph $G^1$ for a runtime interval of execution. A runtime interval is simply a sequence of consecutively fetched instructions. $G$ is a Directed Acyclic Graph (DAG). Each node of $G$ can be labeled with its execution time (an integral number of cycles). The minimal execution time, $G_{\min-t}$, of $G$ is then the length of a maximal weighted path in $G$, where the weights are the execution time of instructions.

If $G$ has $G_{size}$ nodes, then the available Instruction Level Parallelism (ILP) for $G$, $G_{ILP} = G_{size}/G_{\min-t}$ instructions per cycle. ILP measures the average number of instructions per cycle that can be executed in parallel for a region of a program.

Suppose that the execution time of $G$ on a $k$-way super-scalar processor is $G_t$. By execution time, we mean the number of cycles required to execute all instructions in the graph. Fetching and decoding of instructions can be ignored as they are overlapped with execution of prior instructions, and we are ignoring any lost cycles due to cache misses or mispredicted branches. The actual parallelism achieved, or IPC (Instructions Per Cycle), for $G$, is $G_{IPC} = G_{size}/G_t$. $G_{IPC}$ is bounded by both the super-scalarity $k$ and the available parallelism $G_{ILP}$. The optimum IPC that can be achieved for $G$ for a $k$-way super-scalar processor is $G_{ipc-opt} = \min(G_{IPC}, k)$.

Super-scalar processors follow the run-time scheduling rule of executing the next available instructions in the next cycle, to preserve compile-time dependences. If instructions in $G$ are numbered in the order they are fetched, then instruction $i$ is available if all earlier instructions on which $i$ depends have been executed.

The order of numbering the instructions corresponds to the compile-time schedule of instructions, and clearly influences $G_{IPC}$. For example

Suppose the graph in Figure 1 is executed by a 2-way super-scalar processor. If the instructions are scheduled at compile-time in numerical order, then the graph will be executed in 6 cycles, whereas if the nodes are scheduled in the order $[0, 1, 3, 5, 7, 2, 4, 6, 8, 9]$ than the graph will be executed in 8 cycles.

**Theorem 1** Determining an optimal schedule for a $k$-way super-scalar processor is NP-complete if the super-scalarity is greater than 2, even if all instructions execute in unit time.

**Proof (sketch):** Determining an optimal schedule for a $k$-way super-scalar processor is equivalent to job scheduling for identical parallel machines with precedence constraints[9, page 35].

---

1 A runtime dependence graph is more precise than a compile-time dependence graph because represents actual dependences between instructions at run-time.
Determining if an optimal schedule exists is NP-complete if the number of processors (super-scalar) is greater than 2, even if all instructions execute in unit time [7, page 239]. Many approximation algorithms have been developed, such as Graham’s List Scheduling (LS), which gets within a factor 2 of the optimal schedule, but it has been shown that there is a lower bound on approximation algorithms (no better than 4/3 of the optimal solution in the worst case, unless P = NP [9, page 17]).

Instruction Buffering

Our model assumes that a k-way super-scalar processor has one of each type of functional unit, such as an adder or multiplier or load/store unit. However, one other functional unit which is not explicitly taken into account in the above analysis is the reorder buffer. Instructions are placed in the reorder buffer when they are decoded, but not removed from the reorder buffer until they are committed, or retired. An instruction can commit only after it has been executed and all preceding instructions in the instruction stream have been committed2. When an instruction commits, any data written by the instruction are committed to physical registers and/or memory. To keep pace with execution, a k-way super-scalar processor must be able to commit at least k instructions in order per cycle.

In-order commitment places another constraint upon instruction schedules. A poorly chosen instruction schedule can cause the reorder buffer to fill, and fetching and decoding of instructions to cease, even if there is more available ILP. For example, if k instructions fetched in a single cycle depended upon each other in sequence, then it will take at least k cycles before the last instruction is retired from the reorder buffer. Meanwhile, another k2 instructions could have been fetched, decoded, and added to the reorder buffer. However, by scheduling instructions in a “breadth first” fashion in G, then overflow of the reorder buffer can be avoided.

If there are no restrictions upon the compiler generated instruction schedule for G, aside from true dependencies, then it is possible to generate a bounded instruction schedule.

Theorem 2 An optimal schedule for a k-way super-scalar exists which requires an instruction buffer of just 3 × k instructions if the processor has “nop” instructions.

Instruction Level Parallelism In Loops

As shown above, to obtain optimum ILP within a block of code, instruction scheduling is necessary (and is NP-complete). A more practical question is: how useful is instruction scheduling within a loop, or an arbitrary path through a control flow-graph.

Loops with No Dependences

We first consider the case of a loop, with no control dependences within the loop and no loop carried dependences. We show that a k-way super-scalar processor will eventually converge on an optimal instruction schedule: one with an IPC which is the minimum of the loop’s ILP and the super-scalar k. This optimal schedule is achieved no matter what the compiler generated instruction schedule. If there are no loop carried dependences, the processor can always find available work in future iterations to keep it busy.

Theorem 3 Assuming that there are no loop carried dependences, the loop is executed at least k times, and the processor has a super-scalar of at least k, then at least one functional unit must be “busy” (has no idle cycles) throughout the periodic schedule.

Proof: (sketch, by contradiction)

We first consider the simple case of all instructions taking a single cycle to execute.

Assume that in the periodic schedule resulting from the run time instruction scheduler, each functional unit has at least one idle slot in each interval of length k.

Now take the last loop iteration executed, in cycles [c_{last}−k..c_{last}]. The first instruction in this last iteration, node1, could have been executed in the cycle range [c_{last}−N+(k+1)..c_{last}−N+K], as there was a spare functional unit of whatever type this instruction needed. By induction, node1 could have been executed in the

2 Generally, this is enforced to allow precise exception handling and recovery from branch mispredictions.
cycle range \([c_{last}-(N-i)+(K+1)]\cdots c_{last}-(N-i+K]\). Hence the instruction scheduler would have found a shorter schedule.

If an instruction takes longer than a single cycle, then when such an instruction in the last loop iteration was moved into an available cycle slot in an earlier iteration it might not fit. However, by delaying all the instructions following the available slot, the instruction in the last loop iteration could have been moved forward resulting in a shorter schedule (the delay in instructions introduced is shorter than the time taken to execute the moved instruction).

Note that the proof is constructive; it shows that the instruction scheduler must have found a shorter schedule, if there were sufficient loop iterations, and no loop carried dependencies. Such a schedule must have had at least one functional unit busy throughout the periodic schedule.

However, use of a poor instruction schedule comes at a price: the number of iterations required to converge can be as many as the number of instructions in the loop, and the instruction buffer needs to be able to hold the product of the super-scalarity and the number of instructions in the loop. In loops with large numbers of instructions, this means that compilers must either try to schedule instructions better within the loop block, or, if the loop block contains less ILP than the super-scalarity, use software pipelining or loop unrolling to increase ILP within the loop.

**Loops with Dependences**

Loop carried dependences greatly alter the above scenario. Even a single loop carried dependence can prevent run-time scheduling from converging on an optimal schedule.

The graph in Figure 2 is the same as that in Figure 1, with a single loop carried dependence added (the dotted arc). As before, if it is executed by a 2-way super-scalar, an iteration can take either 6 or 8 cycles, depending upon the compiler generated schedule. The compiler generated schedule is significant because no work in subsequent iterations can be commenced before the present iteration is complete.

Suppose that some of the dependences in Figure 2 are deleted, resulting in Figure 3.

The result is now that no matter what the compiler generated schedule, the IPC quickly converges to 2 (the optimum). If the nodes are in the worst case schedule \([9, 1, 3, 5, 7, 2, 4, 6, 8, 9]\), the processor will interleave execution of nodes in the order:

\(S_1, (1_1, 3_1), (5_1, 7_1), (2_1, 3_1), (4_1, 3_2), (6_1, 5_2), (8_1, 7_2), \ldots\)

The ability of the processor to converge upon an optimal schedule depends upon the availability of "sufficient" instructions in subsequent iterations that can be executed before any instructions in subsequent
iterations that depend upon loop carried dependences (LCDs).

Nodes in a dependence graph can be partitioned into 3 sets:

\( G^{LCD} \) The subgraph of \( G \) of nodes that are in a LCD cycle.

Nodes which are sources and sinks of LCDs are in \( G^{LCD} \), together with nodes on paths between nodes in \( G^{LCD} \).

\( G^{\text{pre-LCD}} \) The subgraph of \( G \) of nodes with a path to nodes in \( G^{LCD} \).

These can be computed before the LCDs.

\( G^{\text{post-LCD}} \) The subgraph of \( G \) of nodes with a path from nodes in \( G^{LCD} \).

These depend upon calculating the LCDs.

**Theorem 4** If the sum of the execution times of the nodes in \( G^{LCD} \), divided by the super-scalarity \( k \) is greater than the difference between the best and worst case execution times for compiler schedules for the subgraph \( G - G^{LCD} \), then the processor will converge on an optimal schedule.

Unfortunately, this means that determining for a given graph \( G \), whether scheduling matters, is NP complete (as we have to determine minimum and maximum execution times for \( k \) processors). Also, in practice there are not many nodes in \( G^{\text{pre-LCD}} \), as these correspond to constants or loop invariants.

**Heuristics for Instruction Scheduling**

The analysis above show that instruction scheduling is significant, even for an idealized super-scalar processor. The analysis also suggests that breadth-first style instruction scheduling within a basic block would potentially expose more ILP.

To quantify the significance of instruction scheduling we used a reconfigurable micro-architecture simulator, coupled with an compiler, instruction scheduler and register allocator. This allows us to precisely determine the cycles required to execute standard computational benchmarks on different super-scalar architectures, using various instruction scheduling strategies. The environment we used consists of

**Simulator** SuperDLX[13], which simulates the DLX[8] instruction set. Our simulator includes support for a reconfigurable, multi-level cache, specification of the numbers and latencies of all functional units and various branch prediction strategies. The latencies of all functional units and caches were taken from a high-end MIPs workstation.
Compiler gcc -O2

Instruction Scheduler and Register Allocator This was done as a post-pass on the assembler generated by gcc. First all the dependence and flow-graph information is recreated, then the instruction scheduler is called, then the register allocator. Live ranges are accurately computed using webs. Register allocation is done from the innermost loop outwards, minimizing spill and reload instructions.

We used several different instruction scheduling strategies:

gcc default gcc output (no post-pass instruction scheduling)

depth-first from the dependence graph

breadth-first from the dependence graph

sn-start soonest necessary start time order (in an optimal schedule for unbounded processors) (a variation on breadth first that takes execution time of nodes into account)

gcc-i/s tries to schedule the instruction whose operands are mostly likely to be available soonest (another variation of breadth first, that takes preceding instructions into account)

Figure 4 shows the normalized execution time (execution time divided by the execution time of the slowest of the 5 instruction schedules) vs 8 benchmarks (7 SPEC95 benchmarks and the trivial DAXPY kernel) at a super-scalar of 4³.

Several general conclusions can be drawn. Firstly, as expected, breadth-first heuristics usually do much better than depth-first heuristics. gcc’s default schedule is mostly depth-first, like most traditional compilers⁴, and it performs badly. Secondly, no one heuristic performs best, although the breadth-first heuristic that delays unneeded computation (sn-start) generally does best. Finally, the gains are often very significant, but highly benchmark dependent. In general, instruction scheduling is most significant in applications with

⁴ At a super-scalar of 4, we model 4 of all functional units, meaning that 4 of any operation can be executed in any cycle, subject to dependencies. The size of the instruction buffer was set at 16 times the super-scalar

⁴ The standard Aho, Sethi & Ullman[1] algorithm for code generation for trees is depth-first to minimize register usage
large basic blocks. Thus applu, with an inner block with over 100 instructions, shows a speedup of almost 200% from any depth-first schedule over gcc’s schedule. By contrast, gains for benchmarks with small basic blocks, such as lap or go, or daxpy are smaller or non-existent. Each benchmark has its own quirks (e.g., compress is load/store bound, so scheduling has little effect).

It is instructive to see what happens at a lower super-scalarity. Figure 5 shows the normalized execution time for the benchmarks at a super-scalarity of 2.

As expected, relative performance gains from better scheduling decline at lower super-scalarity, although the decline is only dramatic for applu.

Instruction Scheduling and Spilling

Traditionally, instruction scheduling generally tried to avoid spilling at all costs (hence depth-first heuristics were popular). By contrast, breadth-first heuristics may introduce spilling. Figure 6 shows the number of instructions added for spills and reloads for the benchmarks above.

Three benchmarks required significant numbers of spill instructions for breadth-first instruction schedules: applu, mgrid, and swim. However, all of these benchmarks ran significantly faster using breadth-first schedules, despite the extra spill code, at super-scalarities of both 2 and 4. The DLX instruction set has about 28 available integer registers. To simulate the effect of restricting the number of registers and forcing even more spill code we can force the scheduler to use fewer registers. For example, in applu, reducing the number of available registers from 28 to 18 forced the number of spill instructions added to increase from about 20 to almost 150. Execution times increased for all schedules, by between 10% and 30%. Spill instructions were needed for both breadth-first and depth-first schedules. However, depth-first heuristics still outperformed breadth-first heuristics, though by a smaller margin.

Instruction Scheduling and Optimization

Our results suggest that to extract maximum performance out of super-scalar processors, it is not sufficient to rely on the processor’s out-of-order speculative execution across basic block boundaries. Breadth-first instruction scheduling can provide significant performance improvements. The larger the basic blocks, the
more benefit that can be extracted from instruction scheduling. This suggests that instruction scheduling should be undertaken in combination with optimizations to increase the size of basic blocks, such as: loop fusion, loop unrolling, and predicated execution (which requires hardware support). We are undertaking studies to quantify the importance of such optimizations.

**Conclusion**

Our analysis shows that, contrary to expectations, compile-time instruction scheduling is significant for out-of-order super-scalar processors, but determining an optimal schedule is NP-complete. The analysis suggests that breadth-first heuristics should perform best, and that large basic blocks present the most opportunities for scheduling. Benchmark studies show that performance can depend quite dramatically upon the compile-time instruction schedule, at both super-scalarities of 2 and 4, for benchmarks with large basic blocks in critical loops.

**References**


