Measuring Control-Dependency Limits on Instruction-Level Parallelism

Vance Maverick   Tim Wagner   Robert Wahbe

May 6, 1994

Abstract

Control dependencies are a major barrier to exploiting instruction-level parallelism (ILP). Branch prediction combined with speculative execution is one technique for reducing the impact of control dependencies on achieving ILP. A more aggressive technique is to execute, in parallel, both arms of conditional branches. Previous researchers have suggested that the latter alternative would be a significant improvement over simple speculative execution but would require an unreasonable amount of hardware resources.

We performed a simulation study to determine the performance improvement provided by *extended speculative execution*. At each branch, our simulated machine could choose either to predict or to execute *both* arms of the branch. The instruction fetch units were designed with realistic latencies and cache prefetching strategies; the execute units assumed an optimistic rate of 5 instructions committed per cycle. The number of parallel execution paths could be varied from 1 to infinity.

Average instructions per cycle were computed from traces of actual programs, including several from the SPEC benchmark suite. No program averaged more than 3 instructions per cycle, even with generous execute rates and a perfect cache model. The crucial barrier was instruction fetch, which was seldom able to prepare long runs of instructions for the execute units. We show that no more than 4 parallel units can be kept busy, and that the performance improvement of infinite parallel hardware over the best profile-based branch prediction is negligible.

1 Introduction

Parallel execution has long been understood to be an important path to improved computing performance. Unfortunately, explicitly exposing parallelism using high-level programming abstractions has proven difficult for programmers, compiler writers, and machine architects. Alternatively, processors can exploit *instruction-level parallelism* (ILP), overlapping the execution of independent machine instructions which have familiar sequential semantics. The key advantage of this approach is that programmers can continue to reason about their programs using the same language and architectural model. The complexity of managing the parallelism is hidden by the processor.

Limitations on achieving high rates of ILP can arise either through intrinsic limitations, due to the sequential nature of the programming model, or because of a given machine's implementation. One obvious machine

```
add  r1, r2
add  r3, r4
```

a. Resource conflict.

```
add  r1, r2
add  r3, r1
```

b. Data dependency.

```
add  r1, r2
bz   r1, L1
add  r3, r1
```

c. Control dependency.

Figure 1: Three barriers to instruction-level parallelism.

limitation occurs because of *resource conflicts*, two instructions which contend for a single functional unit. For example, in Figure 1a, the two instructions can only be executed in parallel if the processor has multiple adder units. *Data dependencies* between machine instructions, on the other hand, are an intrinsic problem. The instructions in in Figure 1b cannot be issued simultaneously since the value of `r1` is defined by the first instruction and subsequently used by the second instruction. A third barrier to ILP arises through *control dependencies.* In Figure 1c, the instruction that follows the branch depends on the value of `r1`: in a straightforward implementation, the processor must wait until the branch target is computed before it can fetch the next instruction. Obviously, control flow is inherent in the program, but the barrier to ILP it induces can be addressed by a sophisticated hardware design.

Of these barriers, resource conflicts are the least significant. With the exception of memory operations, increasing the number of functional units is relatively inexpensive. Machines with multiple functional units have been available since the 1960's, and the control of these resources is well understood [9].

The data dependency barrier is more difficult. Data dependencies can be mitigated by register renaming and compiler techniques such as software pipelining [5], but they remain a fundamental characteristic of the sequential programming style, and constitute a ceiling on the possible ILP [11].

Today, control dependencies are the most serious barrier to achieving ILP. If the processor must wait on pending computations before it can fetch the instruction following a branch, several cycles may be lost before the instruction can be issued. This cost is known as the *branch penalty.* It is especially severe in a superscalar or super-pipelined processor, because each wasted cycle has a high opportunity cost in terms of instructions foregone. Since branches occur approximately every 5–20 instructions in most non-scientific programs [3], branch penalties occupy a significant fraction of the running time in such processor designs.

To reduce the branch penalty, instructions from one arm of the branch may be fetched, issued, and executed before the branch target is computed; this is known as *speculative execution*, because the instructions will need to be cancelled if the branch takes a different direction. This approach permits the processor to parallelize instructions from both before and after the branch. The cost depends on how often the processor guesses the right direction. It has long been understood that the two successors of a branch instruction are not in general equally likely, and that they can be predicted with some accuracy [8]. Considerable research has been devoted to improving branch prediction, largely in order to minimize the frequency with which the branch penalty is paid.

1.1 Extended Speculative Execution

In this paper, we explore *extended speculative execution*, in which instructions can be executed, in parallel, from both arms of a conditional branch. The number of parallel execution paths is limited by the number of *virtual processors.* Each virtual processor consists of fetch and execute units. Branches that are highly predictable continue to use branch prediction, but those that are not can speculate down both paths of execution. In the former case, a misprediction requires restart. In the latter case, one of the two virtual processors must be killed. However, the other one is already executing the correct instructions and thus wastes no cycles. When a branch speculates in this fashion, the only branch penalty induced by the fetch unit is the startup cost of the new virtual process (modelled as a single cycle in our simulator).[1]

Some scalar machines, including the IBM 370/Model 168 and the IBM 3033 have attempted limited multiple path execution [7]. However, current researchers, including Wall [11] and Johnson [4], have downplayed this strategy. They point out that it can entail duplicating all the hardware of the processor, and that, as the instruction fetch unit gets further ahead of the resolution of branches, the number of speculative executions grows exponentially. As Wall writes, a real architecture will enforce some limit on the amount of "fanout" permitted before instruction issue must wait on branch completion. (In our model, this appears as the limit on the number of virtual processors that can be simultaneously active.)

In Section 2, we present the basic model of a processor with extended speculative execution. In Section 3, we

[1] If there are no available processors, then the branch must be predicted, since the only alternative would be to stall.

describe a simulator for this architecture and the detailed components of the model. Using the simulator, we explore the relationship between three quantities: the number of virtual processors, the branching strategy (and the accuracy of branch prediction), and the fraction of ideal ILP achieved. In Section 4, we discuss the parameters and results of the several experiments we carried out, and then summarize our conclusions in Section 5.

2 Execution Model

In this architecture, program execution is distributed among one or more virtual processors. At any given time, at most one of these processors is executing instructions which will prove useful; the rest are speculating along what will eventually prove to be untaken paths. When a process commits a branch instruction, any instructions it was executing which depended on the results of that branch are also committed. Execution results for a virtual processor remain hidden from the rest of the machine until this commit point. When a virtual process turns out to be executing the wrong path, it must be killed. Killing a virtual processor means its execution stream has no effect on the user-visible machine state. The number of instructions that can be executed before their status is known is limited by the size of the pipeline and instruction window; this limit was never reached in our simulation.

In order to simplify our study of control dependencies, we have eliminated functional conflicts and memory bandwidth as possible barriers to instruction level parallelism. Each virtual processor is assumed to have a sufficient number of functional units to accommodate the available instruction parallelism. Each virtual processor maintains a private memory cache for memory operands that have not been committed; all virtual processor data accesses first check the private memory cache. Instructions have a latency of one cycle. Each virtual processor has an independent instruction cache port; these ports operate simultaneously without conflicting. The instruction cache is assumed to be perfect, but it does require a single-cycle latency to transfer a new address when prefetching is unsuccessful.

The entire state of a virtual processor can be *forked*, with a fixed latency of one cycle, to any idle virtual processor. Forking copies all architectural processor state from one virtual processor to another.[2] Forking is the mechanism by which both sides of a branch can be speculatively executed in parallel.

2.1 Branch Prediction

Many branches are highly predictable, going one direction or another with a probability near 100%. On a benchmark set including the C programs from the SPEC92 benchmark suite, we found that over 50% of the dynamic branches were 95% predictable or better. When the fetch unit of our processor reaches such a branch, it may be wise to execute only the likely arm of the branch: computational effort expended on the other arm is probably wasted, and uses a virtual processor that may be better assigned to a less predictable branch along the taken path.

Three different models for branch prediction were used for the simulation. As a lower bound, one predictor always chose the fall-through arm. The other two branch predictors represented good hardware and software prediction. The hardware model uses a two-bit code per branch. The software prediction is provided by profiling the program with the *same* input.

For all three branch prediction strategies, we tag each conditional branch instruction with a two-bit code representing the prediction. In addition to marking the branch taken or untaken, this code indicates whether the prediction is strong or weak. This can be thought of as a confidence level (obviously additional bits could be used to provide more refined gradations). In the fixed prediction scheme these are pre-determined. In the hardware prediction scheme they represent the state machine at the branch at the time of prediction. For

[2] We are defining here the semantics of forking, not suggesting an actual hardware implementation. Since the fraction of machine state written to between forking and joining is typically small, an efficient implementation would exploit this sparseness in its design.

the software scheme they are inserted via opcode selection, and could be determined by the compiler [10], by a tool which analyzes the executable image [1, 12], or by profiling the program on sample inputs [3]. We investigate only the profile-driven software approach.

3 Implementation of the Model

In order to investigate this model in a realistic setting, we employed trace-driven simulation. A cycle-level simulator was written in C, and consumed as input the following data:

- A dynamic trace (basic block level) of program execution.
- The control flow graph for each procedure of the program.
- Branch prediction statistics for the profiling model (hardware prediction was simulated on-line).

The control flow graphs for each procedure were gathered using a modified version of `qpt` [6] called `qpt-266`. The main technical challenge in building control flow graphs from the code segment of the binary is determining the target of *indirect control transfers*. Most control transfer instructions have statically resolvable targets; however, the target of an indirect control transfer instruction is specified via a register and can only be determined at run-time. Common programming language abstractions such as function pointers, case statements, virtual functions, and continuations are typically implemented using indirect control transfers. `qpt` uses compiler-dependent idioms to recognize these constructs and thus determine the target of the indirect control transfer.

In addition to constructing the control flow graph, `qpt-266` inserted instructions in to the program to monitor execution. As the instrumented program executes, a dynamic basic block trace is produced.

In our simulations, branch predictions were derived by profiling the program, using the same trace. In the terminology of Ball and Larus [1], we used the *perfect static predictor*. In our own research [10], we found that realistic profiling (based on different inputs) performs only slightly worse than the PSP. The average miss rate of the PSP was 11%; the miss rate from realistic profiles was 13%.

One complexity of the implementation was that, while the trace provided the execution pattern for the real thread, any others required on-the-fly trace generation. Without a full data simulator, we could not generate true execution paths. Instead, we used the CFG and *distribution sampling* from the profile to produce a realistic pattern of control flow. Each conditional branch, indirect call, and indirect jump had its outcome decided by this random sampling on a profile for the same input that produced the trace.

3.1 Simulator Components

The simulator implementation was designed along the same lines as the fundamental architectural quantities in the model: instruction cache, thread management, and individual threads. The latter were decomposed into instruction cache ports, fetch units and execute units.

The instruction cache implements a multi-ported, perfect cache model. Each cache hit requires a single cycle between presenting the address and receiving the block. Blocks are aligned to eight word boundaries; thus a misaligned fetch limits the bandwidth. Each cache port performs buffering and prefetching: if the next request is to a starting location within or immediately following the previous block's range, then it is available in the same cycle (i.e., such reads have zero-cycle latency).

The fetch units perform a partial decoding of the available instructions read from the cache port. All the instructions up to and including the first one which produces a change in control flow are issued into the instruction window on the next cycle. Direct calls, returns, and unconditional branches require no further processing (the fetch unit will deliver the new address to the instruction cache on the next cycle).

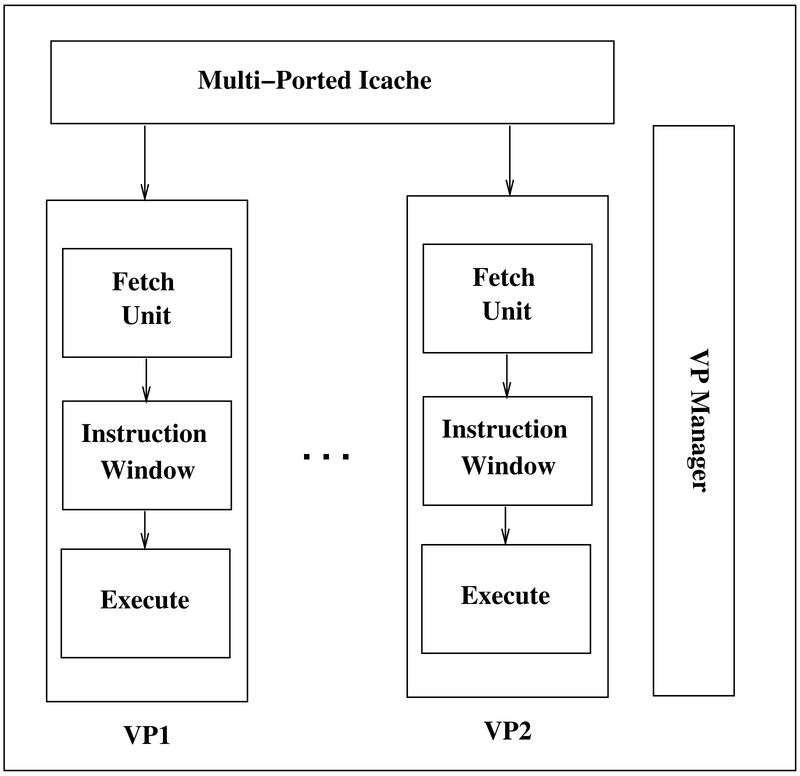

Figure 2: High-level diagram of simulator.

On indirect calls and indirect jumps the target is *computed*, and is hence subject to data dependencies. When this happens, we stall the fetch unit until the instruction commits and the target is known. Since this happens much less frequently than branches, stalling is less of a problem here. However, a more aggressive model would attempt to predict the most likely target address for indirect transfers of control as well.

Conditional branches represent the most interesting case. We use the current processor model (1 to infinite processors) and the branch prediction strategy to decide what to do. If the branch is predictable, then a single arm is chosen and the branch instruction is tagged with the information needed to restart at the other arm if the prediction proves to be incorrect. If the branch is unpredictable and a virtual processor is idle, then we fork: allocate the idle processor, copy the current state to it, and allow both arms of the branch to proceed in parallel. In the forking case the branch instruction carries an identifier for the child virtual process.

The fetch unit issues its instructions to the instruction window, a fixed-size FIFO buffer. They are read from the window by the execute unit, which can complete instructions at a rate of five per cycle. (This is the average execute rate determined by Johnson for a superscalar processor with no control-flow impediments to achieving ILP. In this simulation, we do not directly model data dependencies.) Any conditional control flow instructions are scheduled with one cycle between them. Control flow instructions are reported to the fetch unit when they commit. For indirect calls and jumps, this releases the fetch unit from its stalled state. For conditional branches the situation depends on the fetch unit's decision (to predict or fork), the guessed direction of the branch, and the actual direction.

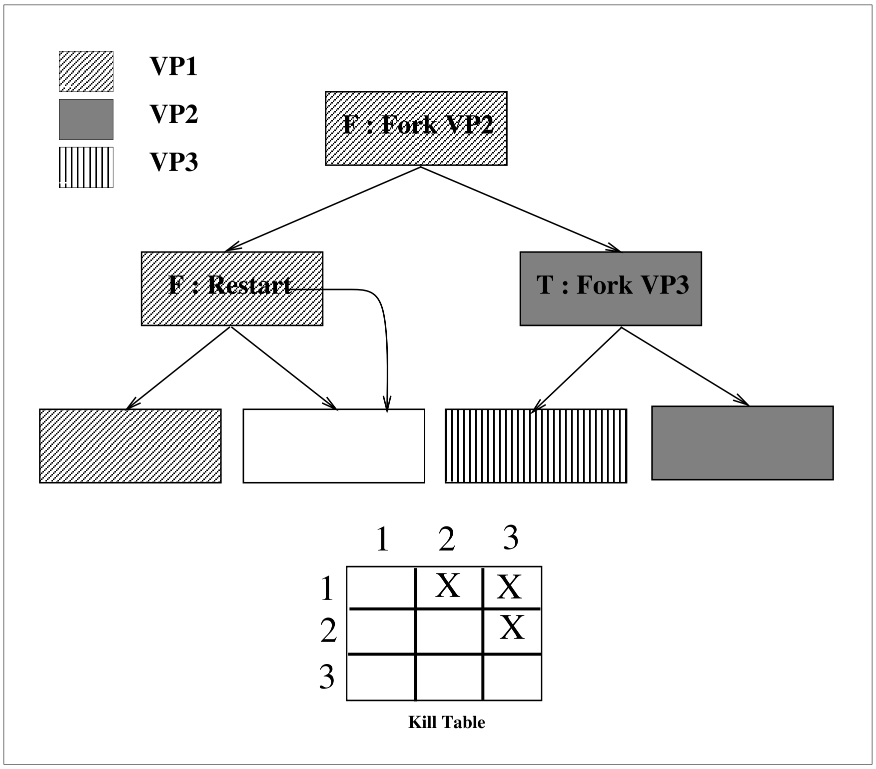

Figure 3: Example showing three threads and the kill table entries for them.

3.2 Managing Virtual Processors

Since several branches may be outstanding at any one time, a machine with more than one virtual processor must consider the problem of transitive dependencies: when a branch is committed, it is possible that two or more virtual processors must be killed as a result. To handle this, we must augment the tag information carried within the branch instructions themselves as they flow through the execute pipeline with another data structure: the *kill table*.

Technically, the kill table is the transitive closure over the dependency graph between virtual processors. It is updated at fork and kill times, and is consulted when a branch instruction is committed. The algorithm is shown below (**A** and **B** represent two virtual processors).

- **A** handles a branch by predicting
  - The address of the unpredicted arm is written to the branch tag, so it can be consulted later if the prediction proves incorrect.
- **A** handles a branch by forking **B**
  - **B** is dependent on any process that **A** is dependent on: copy **A**'s column in the kill table to **B**'s column.
  - No process is dependent on **B**: clear **B**'s row in the kill table.
  - **B** depends on **A**: turn kill table's (**A**, **B**) entry on.
  - Write an identifier for the child processor, **B**, into the tag of the branch instruction. This instruction will be executed by **A**'s execute pipeline.

- Processor **A** is terminated
  - **A** is no longer dependent on any thread: clear **A**'s column in the kill table.
- Branch commits
  - If the branch was predicted correct, do nothing.
  - If the branch was predicted incorrectly, kill any threads which this thread forked: if any items in this threads row are on, kill the corresponding thread.
  - If the branch was forked and the parent is the correct path: terminate the child and any threads which depend on it.
  - If the branch was forked and the child is the correct path: terminate the parent and any threads which depend on it, *but* protect the child and any threads which depend on *it*. Each element which is on in the parent's row and off in the child's row indicates a thread to be killed.

Consider Figure 3, showing three virtual processors executing a region of code that contains three branches. Virtual processor 1 (**vp1**) was the initial thread: when it reached the first branch, the prediction was false with low confidence. Hence **vp1** took the fall-through, but forked **vp2** to explore the taken path. **vp2** in turn forked **vp3**. **vp1** encountered a second branch as well, but since it was predicted false with high confidence it did not result in a fork. The kill table and branch tags are shown for all three outstanding branches. If the initial branch turns out to be false, then **vp2** and **vp3** will be terminated. If it turns out to be true, **vp1** will be killed.

4 Experiments and Results

We ran our simulator on the programs listed in Table 1, measuring the ILP achieved under different conditions. The primary factor varied was the number of virtual processors available to the machine (varied from 1 to infinity). As a secondary factor, we varied the branch prediction model, using one of: predict fall-through, 2-bit hardware, or software profiling. Other parameters provided as inputs to this simulation experiment but held constant are described below.

- Scheduling delay between adjacent control-flow changing instructions: 2 cycles.
- Execute unit maximum rate of consumption: 5 instructions/cycle.
- Instruction size: 4 bytes.
- Cache block size: 8 instructions (32 bytes).
- Delay between fork request and first issue: 1 cycle.
- Delay between kill request and availability: 1 cycle.
- Instruction cache transfer latency (hit time): 1 cycle.
- Instruction window limit: 128 instructions.

Program	Lines	Description
compress	1,503	Unix compression utility
eqntott	3,482	Translate boolean functions to truth table
fpppp	2,718	Quantum chemistry benchmark
permute	78	Compute all permutations of N integers

Table 1: Programs used in this study.

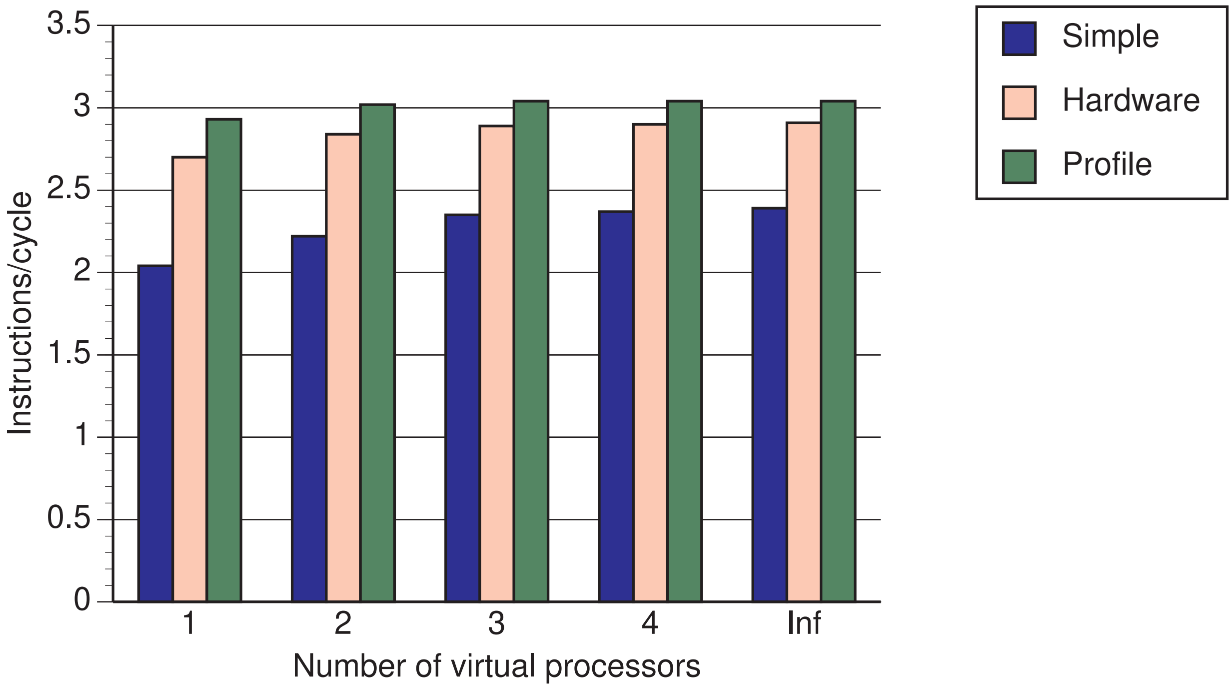

Figure 4: Instruction-level parallelism as a function of the number of processors and the branch prediction technique.

4.1 Core Experiment

For each program, we collected a trace of the first 10,000 instructions and used it to drive the simulator.[3] We performed 15 runs for each program, using 1, 2, 3, 4, or an infinite number of virtual processors, crossed by the three branch prediction models. The results from each program were averaged.[4]

Figure 4 shows the average measured ILP for each choice of parameter settings. The overall improvement from a single-processor model with simple branch prediction to an infinite-processor model with accurate branch prediction is only 50%. Each column cluster represents the three branch prediction models for a given number of virtual processes. The vertical axis shows average instructions/cycle, which is directly correlated to program execution time.

[3] This simple approach to limiting simulation overhead does not effectively characterize programs that consist of phases with distinct behavior. A better approach would be to sample the trace at several different points. We expect to implement this capability in the near future.

[4] None of the programs exhibited behavior which deviated significantly from the average, so no explicit analysis of variance was performed.

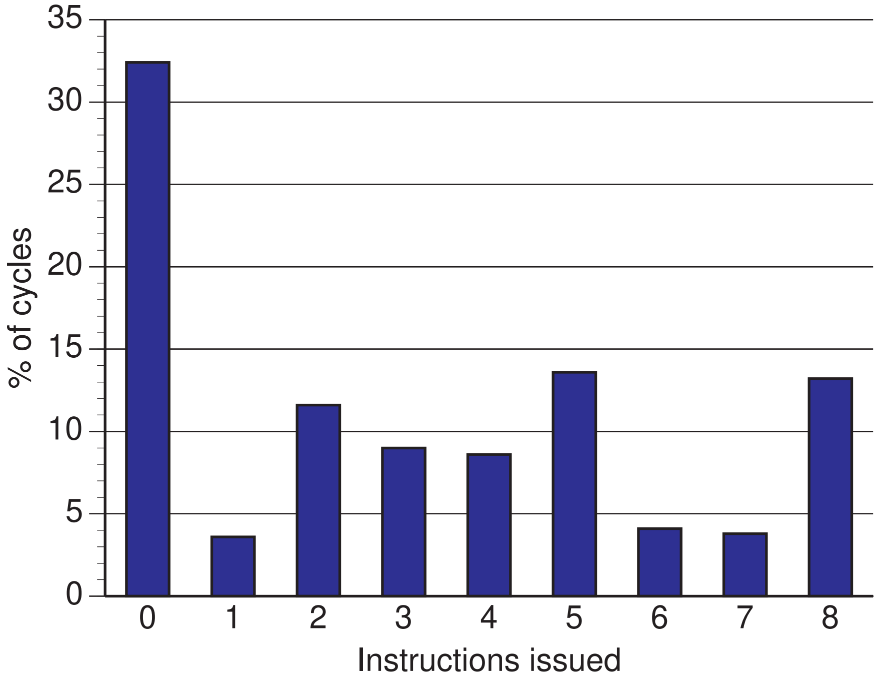

Figure 5: Histogram of the number of issuable instructions on each cycle. Maximum is 8, the size of an instruction cache line; average is 3.13.

4.2 Instruction Fetch

Since the execute unit's rate exceeds the observed rate, we can conclude that the performance bottleneck remains in the fetch unit. To further understand why this bottleneck occurs, we instrumented the simulator to gather statistics on the distribution of issue block size: the number of instructions transferred from the fetch unit to the instruction window on a given cycle. Figure 5 shows the observed distribution for a typical program.

This distribution is affected by several factors. Instruction cache block alignment restrictions keep the block read by the fetch unit itself from being full (this distribution, not shown, is a random distribution between 1 and 8, because the compiler makes no attempt to pack the basic blocks onto aligned boundaries). Once read, the fetch unit can only issue a single control-flow instruction: because these occur frequently in non-scientific codes, the average number of issuable instructions is further reduced. When the PC is altered (e.g., a conditional branch is predicted taken or an unconditional branch is encountered) one cycle is spent waiting for the instruction cache transfer, which stalls the fetch unit and creates a zero-issue cycle in this distribution. Stalling to wait on computed addresses also adds zero-issue cycles.[5]

The average number of issued instructions is 3.13, very close to the highest measured ILP. Based on this observation, we believe instruction fetch to be the crucial bottleneck for aggressive superscalar architectures.

4.3 Processor Utilization

Other researchers have dismissed extended speculative execution because of the (purported) exponential growth in resource usage as the "speculation window" grows. Figure 6 shows a histogram of the number of processors active in every machine cycle, running our benchmark suite with hardware branch prediction and

[5] Technically, the fetch unit would also stall if the instruction window filled. Because the bottleneck is in the fetch unit, this never happened: the instruction window never caused the fetch unit to block on the programs we traced.

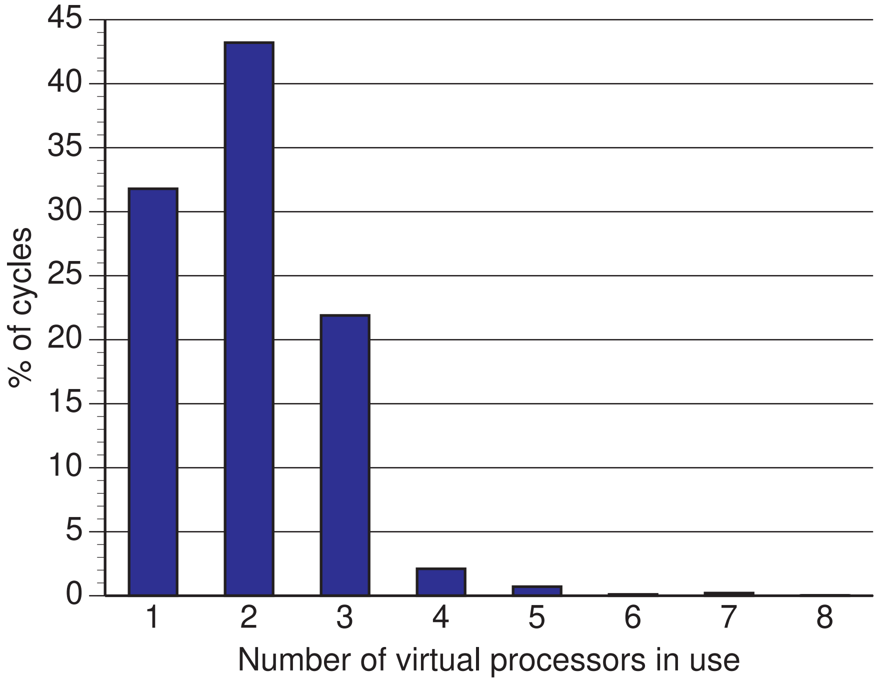

Figure 6: Histogram of the number of processors in use.

infinitely many processors. Only very rarely (in fewer than 1% of the cycles) were more than four processors active at any time. Thus, while this hardware scheme is hard to justify (unless a design to implement it proved cheaper than implementing branch prediction), is does *not* exhibit exponential blow-up. The presence of highly predictable branches, the existence of data dependencies, and the inherent delays in instruction issue all conspire to produce a limit far below the mythical 16, 32, or 64 units that have been suggested.

5 Conclusions and Future Work

We have explored the limits of instruction-level parallelism by simulating a novel machine architecture: one in which branch predictions and replicated fetch/execute units are combined to avoid branch prediction failures. This architecture handles unpredictable branches by extended speculative execution: when possible, it executes both sides of these branches, killing one path when the branch target is computed. Each stream of instructions is handled by a separate virtual processor, capable of computing 5 instructions per cycle in its execute unit. Despite several optimistic assumptions about cache behavior and execution rates, we found that the actual instruction-level parallelism achieved by our simulated machine was about 3. The bottleneck remains the instruction issue rate.

In a related simulation study, Wall [11] estimated an ILP limit of about 5, by making reasonable assumptions about branch prediction, register renaming, functional unit latencies, and data dependencies. However, he did not explicitly model instruction fetching, instead assuming that large windows of instructions were continuously available.

Our work may be seen as an extension, taking Wall's ILP figure of 5 as a limit on the throughput of a single virtual processor, but adding a realistic model of instruction fetch. In our measurements, this fetch unit was able to provide 3.13 instructions per cycle, on average, to its execute unit. This figure is only slightly more than our measured ILP, which reached at most 3.04.

Several opportunities exist for extending this research. First, the simplistic execute unit, though adequate for modeling systems in which the fetch unit is the bottleneck, should be replaced by a full data-dependency scheduler for future work. Second, we can explore more generous models of instruction fetching, permitting multiple conditional branches to be issued each cycle (it is instructive to locate the sources of the bottleneck, even if the hardware needed to circumvent them is difficult or impossible to fabricate at the current time). The fetch unit can be modified to predict or speculate on jumps and indirect calls as well as conditional branches. Finally, we can simulate the effect of a profile-driven optimizer in the compiler that would use trace scheduling [2] to improve performance by choosing fall-through branches whenever possible and eliminating expensive calls and unconditional branches from the dynamic trace.

However, unless these enhancements improve the throughput of the instruction fetch units significantly, the lesson of this study is that enhanced speculative execution is not worth its implementation cost. In our simulations, a single-processor model performed nearly as well as a model with unlimited virtual processors. Further, good branch prediction, whether derived from hardware or software strategies, is crucial for achieving high performance, regardless of the degree of speculative execution.

References

[1] Thomas Ball and James R. Larus. Branch prediction for free. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[2] Joseph A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[3] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, MA, 1992.

[4] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[5] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[6] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin-Madison, March 1992.

[7] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17:6–22, 1984.

[8] James E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[9] R. M. Tomasulo. An efficient algorithm for exploiting multiple functional units. *IBM Journal*, 11:25–33, January 1967.

[10] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

[11] David W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Santa Clara, CA, 1991.

[12] Youfeng Wu and James R. Larus. Static branch frequency and program profile anaysis. *Submitted to ASPLOS*, 1994. Extended abstract.