Stallscope: Illuminating the Black Box

Leick Robinson
Freescale Semiconductor
7700 West Parmer Lane
Austin, TX 78729
leick.robinson@freescale.com

Abstract

As microprocessors become increasingly more complex, cycle-accurate simulation has become a valuable tool for performance analysis and microarchitectural exploration. However, parallelism, complex interdependencies, and deep pipelining in modern superscalar processors make it difficult to identify how a particular microarchitectural design feature ultimately affects performance, particularly in the early stages of the pipeline. This makes it difficult to accurately determine the best microarchitectural tradeoffs. To gain insight into processor behavior, the Stallscope performance analysis tool was developed; it addresses these problems by revealing the root causes for IPC loss and identifying the primary contributors. It tracks the causal chain of stall events through the pipeline, through operand dependencies, and through pipeline flushes, and can identify whether stalls at a particular stage in the pipeline ultimately affect instruction completion, to what degree, and by which pathways. Its highly localized design makes it easy to instrument an existing cycle-accurate simulator. During the design and analysis of Freescale Semiconductor's next generation processor, Stallscope has been used to resolve counterintuitive and seemingly paradoxical observations, such as why a proposed prefetching algorithm that reduced the number of instruction fetch misses resulted in lower overall performance. This paper describes the Stallscope methodology and implementation.

1. Introduction

Because of the complexity of modern superscalar processors, interactions between different blocks of the microarchitecture can impact performance in ways that may not be obvious or intuitive. Cycle-accurate simulation has become a valuable tool for performance analysis and microarchitectural design exploration. However, even in simulation, complex interdependencies can make it difficult to identify cause and effect. In particular, the influence of early pipeline stalls on the measured instructions per cycle (IPC) can be unclear. This makes it difficult to identify the bottlenecks in the system that have a genuine impact on performance and accurately determine the optimal microarchitectural design choices.

Stallscope is a performance analysis tool that reveals the root causes for IPC loss and identifies the primary contributors. It tracks the propagation of stalls through the pipeline and can identify whether stalls at a particular stage in the pipeline ultimately effect instruction completion, to what degree, and by which pathways. Its highly localized design makes it easy to instrument an existing cycle-accurate simulator.

2. Motivation

As stated in the introduction, in modern microarchitectures, it is difficult to determine how a particular design feature truly affects performance. We first encountered this during our evaluation of alternative branch prediction algorithms. Tests with our cycle-accurate performance model showed that, for some branch predictor alternative, branch prediction rate increased, but IPC was reduced.

Similarly, in more recent investigations, the evaluation of a promising instruction prefetching algorithm led to seemingly contradictory results. Although tests with our cycle-accurate performance model showed that this algorithm reduced the number of instruction cache misses, the overall IPC went down.

Post-analysis of statistics collected at the individual units is insufficient to explain these sorts of counterintuitive results because it does not reflect the complex interaction between the units. So, a better

analysis tool was needed that would give us visibility into the cause and effect chain between early pipeline stalls and instruction completion at the end of the pipeline; this led to the development of Stallscope.

3. Fundamental Stallscope structure

3.1. Top-level goals

The rate at which we complete instructions is the metric that defines performance, so Stallscope was designed to answer the question "why do we sometimes fail to complete instructions." Each cycle, there is the potential to complete one or more instructions, depending on the completion width of the processor. In Stallscope, these are referred to as *completion opportunities*. Every cycle that we do not complete any instructions, or that we complete fewer than the number of completion opportunities, Stallscope seeks to answer the question "why did we fail to complete?" for each of the completion opportunities lost.

In most situations, the reason for a lost completion opportunity can be attributed to an earlier stall elsewhere in the pipeline. That stall, in turn, may have been the result of a previous stall elsewhere. By following the logical cause-and-effect chain of events, Stallscope determines the original event that ultimately led to the completion stall and the associated loss of performance.

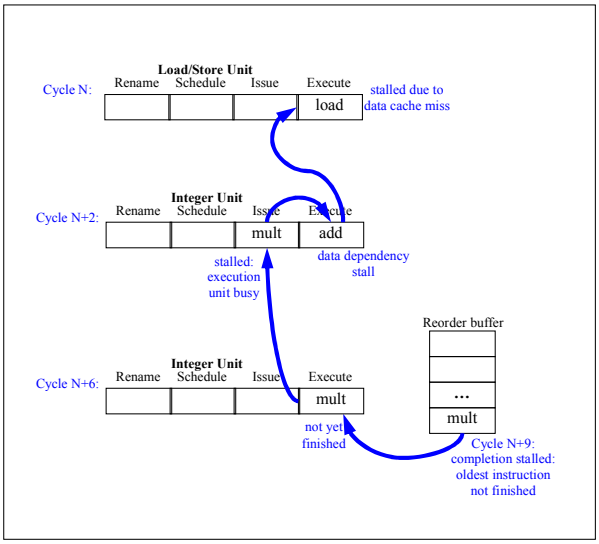

Figure 1. Cause-and-effect stall event chain

For example, consider the case in Figure 1 where the oldest instruction in the reorder buffer is a multiply instruction that is unable to complete because its execution has not yet finished. In this example, it would be finished and ready to complete except, on an *earlier* cycle, its execution had been blocked by an older add instruction. The add, in turn, was stalled due to a data dependency on a load instruction. Finally, the load was stalled due to a miss in the data cache. So, in this case, the add instruction's failure to complete was ultimately due to a load miss in the data cache.

3.2. Implementation requirements

In addition to providing the cause and effect information that we need, there were also implementation requirements that Stallscope needed to satisfy:

- it should be as efficient as necessary in order to minimize its impact on the speed and performance of the model
- it must be unintrusive and easy to instrument
- it must be extensible and expandable

The first requirement was addressed through judicious choice in the low-level design. Measurements of our instrumented simulator showed that Stallscope imposed about a 20% increase in total runtime, which was well within our performance needs.

Since we were instrumenting an existing model, it was essential that Stallscope be as unintrusive and easy to instrument as possible. Stallscope also needed to be highly extensible and expandable so that the instrumentation could evolve with the model. Both of these goals are accomplished with Stallscope's highly localized design.

3.3. Stall events

Each cycle that an instruction fails to proceed from one stage to the next stage, a *stall event* is recorded. For each stall event, we also identify the causal stall event, if any, that was the primary immediate, local cause of this stall. For example, if an instruction is stalled in the Decode stage because the flow control (in this example) reported that Rename was occupied and stalled the previous cycle, the current stall event in Decode might be labeled "Rename full", and the causal stall event would be whatever the reason was for the Rename stall last cycle. Consequently, the Stallscope instrumentation is *localized* in that each stage only needs to interact directly with the immediately adjacent stages that affect this stage or that are affected by it.

Each stall event is labeled with a *stall type* (such as "icache miss", "Rename empty", or "branch mispredict – predicted taken"). There is no global or "master" list

of stall types; this would inhibit extensibility. Instead, each component of the simulator is free to manage its own set of locally defined stall types, called a *stall set*, or a group of components can share a stall set, as the local modeling needs dictate. The internal Stallscope implementation automatically combines the stall sets in use "behind the scenes" at run time as stall events are accumulated (as discussed below).

As mentioned above, each stall event can have one causal stall event that led to this stall. That causal stall event can, in turn, have a causal event, and so forth, and so each stall event is at the head of a cause-and-effect chain of stall events.

From this, we can define two fundamental types of stall events: "leaf" stalls and "non-leaf" stalls. Leaf stalls are at the end of a cause-and-effect stall chain, they have no causal stall event, and so they represent the *fundamental cause* of the subsequent chain of stall events. The designation "leaf stall" comes from the fact that these will be the leaf nodes of the hierarchical stall event tree that is formed when stall events are accumulated; this is discussed in more detail below.

At any given time, particular stall type should be used to create either leaf stalls or non-leaf stalls, but not both. Note, however, that, as the microarchitectural model develops, former leaf stall types can become non-leaf stall types as the Stallscope instrumentation is extended into new parts of the model.

Stall event *propagation* denotes the mechanism by which causal stall events are conveyed within the model to the locations where they may be the *immediate* cause of other stall events. There are three fundamental mechanisms of stall event propagation:

- pipeline propagation, where events propagate to the immediately adjacent stages of the pipeline
- propagation through operand dependencies
- flush propagation, where the event that caused a pipeline flush is broadcast to the affected entities

3.4. Pipeline propagation

Pipeline propagation can proceed upstream or downstream (see Figure 2):

- An *empty stall* represents the case when instructions fail to move on to the next stage because there is nothing in the stage to move on. Empty stalls propagate downstream.
- A *full stall* represents the case where an instruction is blocked or prevented from propagating to the next stage. This can be due to stalled instructions ahead of this one, unavailable resources, or explicit interlocks. Full stalls propagate upstream to the previous stage.

  Full stalls also represent the case where an instruction is queued behind other instructions. In this case, ahead of this instruction the queue is "full", and so full stalls also propagate "upqueue".

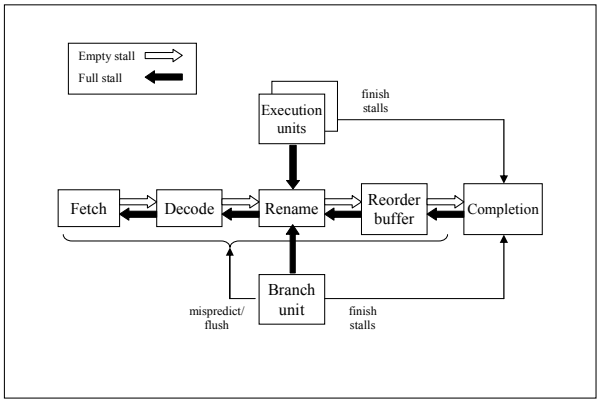

Figure 2. Stall propagation for a simplified pipeline

At each stage, the timing of pipeline propagation depends explicitly upon the local flow control.

3.5. Operand dependencies

An instruction can also stall because of data dependence on another instruction. The stall must be due to one of three reasons:

- The other instruction might have encountered one or more stall events during its execution.
- For most instructions, if no stalls are encountered, the instruction will finish within a characteristic minimum latency period. If the minimum latency has elapsed and the instruction has still not finished, then it is a *long executing* instruction (an example would be an integer divide).
- If no stalls have been encountered and the characteristic minimum latency has not yet elapsed, then the two instructions are too close together in the instruction stream and there simply hasn't been time for the other instruction to finish.

The second and third cases cause a leaf stall event to be generated.

The first case is the most interesting. Here, the current instruction is stalled because of a stall event that has occurred in the *other* instruction's execution

path. We need a mechanism to record the stalls that each instruction has encountered so that we may propagate stall events when a stall due to data dependence occurs. To facilitate this, each instruction maintains a history of its execution stalls.

However, the other instruction may have encountered multiple stall events, but the current stall can only have at most one causal stall event. How do we choose which one to propagate? After careful consideration and some experimentation, we determined that the most recent stall condition is usually the most critical; if the older stall events had not occurred, it is likely that the more recent stall condition would simply have been encountered sooner and stalled the instruction for a longer period of time. So, the rule of thumb is "if there is more than one stall event to choose from, choose the most recent".

The consequence of this rule is that the instruction stall event history is modeled as a stack. Each stall event encountered during an instruction's execution is pushed onto the history stack. If an instruction is stalled for multiple cycles for the same reason, a separate stall event will be pushed onto the instruction's stall history for each cycle of stall; each of the stall events is unique and may have a different causal event chain, but they will all be of the same stall type. Each cycle that an instruction is stalled waiting for its operands, the causal stall event is obtained by popping the next most recent stall event off of the source instruction's stall history.

Note, however, that the source instruction's stall event history must not be actually modified when the most recent stall event is "popped" off. If multiple instructions are stalled waiting for data from the same source instruction, each needs to independently "pop" the same stall event(s) off the source instruction's history, starting with the most recent. This was implemented in Stallscope by the development of the MultiStack container class. If the instruction is still stalled after the source instruction's stall event history has been exhausted, then all remaining cycles of stall must be due to the second or third case above.

By similar reasoning, if an instruction is stalled waiting for operands from more than one source instruction, the stall event history for the youngest instruction is accessed.

3.6. Flush propagation

One consequence of the modern superscalar architecture is that sometimes there is no choice but to flush the pipeline. For example, because instructions can execute out of order, the speculative instructions after a mispredicted branch may already have finished execution and be awaiting completion by the time the mispredict is discovered. In this situation, the designers may choose to flush the pipeline and restart instruction fetch on the current path. In simulation, the same mechanism that is used to broadcast the flush can also be used to send the reason for the flush as a causal stall event.

On the cycle after the pipeline is flushed, Stallscope associates an *empty stall* event with each pipeline stage; the causal stall event for these empty stalls is the broadcast stall event containing the reason for the flush. As the pipeline refills in the cycles following the flush, these empty stall events will propagate downstream until they ultimately drain out of the end of the pipeline.

In some microarchitectures, due to optimizations and timing constraints, a flush can also cause temporary interlocks to be set in the pipeline. If any instructions fetched after the flush become stalled behind these interlocks, the flush causal stall event will become the causal event for the resulting *full stall*. This full stall will propagate upstream as the instructions back up in the preceding stages waiting for the interlock to be released.

3.7. Combining the mechanisms: instruction completion

Some stages may combine the different propagation mechanisms. The completion/commit stage is one of the more complex and serves as a good example.

In our model, there are three basic reasons why completion can stall:

1. There are no instructions to complete (the reorder buffer is empty).
2. The oldest instruction is not finished.
3. Various structural hazards and interlocks in completion

Furthermore, there are three reasons why the oldest instruction may not be finished:

2a. it might have encountered one or more stall events during its execution

2b. if the reorder buffer was *nearly empty*, the instruction may have reached the bottom of the reorder buffer (become the oldest instruction) too quickly, before the characteristic minimum latency has elapsed, so that, after all execution stalls (if any) have been accounted for, it has still not yet had time for execution to finish.

2c. after both of the above have been accounted for, if the instruction still has not finished, then it is a long executing instruction

The stalls in categories 1 and 2b are encountered after the pipeline has been flushed or allowed to drain and has not yet refilled. So, to determine the causal stall events for these categories, we maintain a history of the most recent empty stalls that have propagated along the pipeline downstream to completion, back to the last flush (and its associated flush stall event).

The causal stall events for stalls in category 2c are popped off of the instruction's stall history, in the same manner as described for operand dependencies.

Finally, the stalls in categories 2c and most of those in category 3 are leaf stalls, with the exception of some interlock stalls that trace their cause to the most recent flush stall event.

3.8. Stall accumulators

The function of Stallscope is to determine the fundamental causes of stalls at some point in the pipeline. In our case, we are interested in completion stalls, which directly affect IPC.

Accordingly, we added a *stall accumulator* to the completion stage of our processor model. Each cycle that completion is stalled, the stall accumulator consumes the associated stall event. The stall event and all nested events in its causal stall chain are accumulated into a hierarchical statistical tree summary showing the causes of all completion stalls.

Figure 3 shows an example of the top levels of a hierarchical tree summary, filtered so that only those contributors that account for more than 0.5% of the total number of completion opportunities are displayed.

The nodes at the uppermost level of the tree represent the immediate stall conditions that occurred in completion, along with counters indicating their relative frequency. Each of these nodes then branch to the next level of nodes, which represent the events that were the direct cause of the completion stall events, and those in turn branch to their direct causal events, and so forth, until the branches terminate at the leaf stalls.

For example, the top-level tree in Figure 3 shows that 63% of the completion opportunities were lost because the oldest instruction was not yet finished. 36% of the completion opportunities were lost due to a *load/store* instruction that is the oldest instruction is not yet finished. Of those, one third (12%) are not yet finished because they are waiting for load miss data.

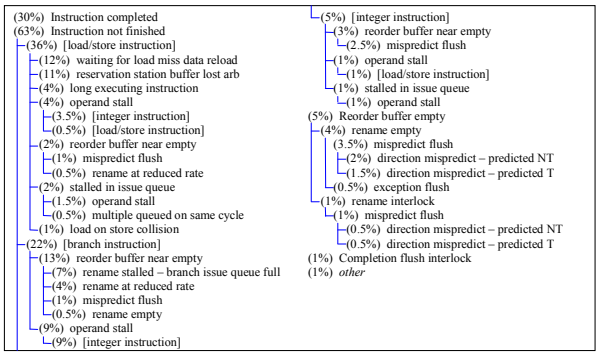

Figure 3. Top levels of an example hierarchical tree summary (filtered)

Note that the same stall type can occur at multiple nodes in the tree, as their may be multiple cause-and-effect pathways by which a particular stall can ultimately stall completion. For example, a load miss in the data cache might cause the load instruction to directly stall completion if it becomes the oldest instruction before the data has been read from memory. Alternatively, an earlier stall might have allowed the data read to finish before the load is ready to complete and thus removed the direct causal path, but a younger instruction with data dependency on the load could be sufficiently delayed that it, in turn, stalls in completion.

4. Simulation methodology

In our work, Stallscope was used to instrument our existing cycle-accurate simulator that models the next-generation PowerPC processor. This simulator is an *extremely* accurate, highly detailed model of the hardware currently under design, suitable for performance analysis and exploration of design tradeoffs.

Our analysis utilized 185 workloads and benchmarks that included general-purpose and networking workloads from SPEC2000, SPEC95, EEMBC, and proprietary workloads.

5. Analysis and reporting

The accumulated hierarchical tree is typically very large; several means are employed to distill meaning out of this vast quantity of data. Our experience has shown that the most useful are the:

- top-level hierarchy
- leaf summary
- twig summary
- node summary

The top-level hierarchy displays the nodes at the first N levels of the tree hierarchy, along with the percentage of each node's contribution to the total number of completion stalls. In practice, we have found the first four levels to be useful.

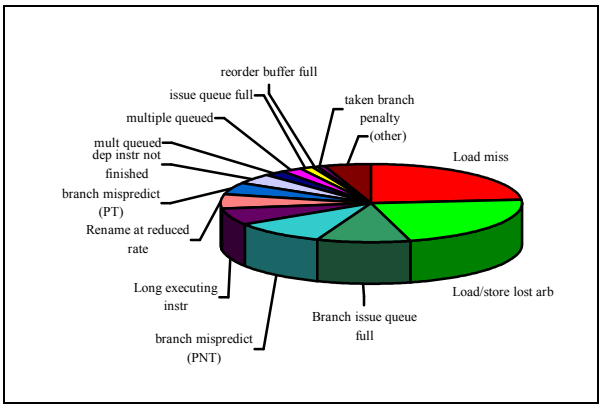

Figure 4. Example leaf summary of major completion stall contributors

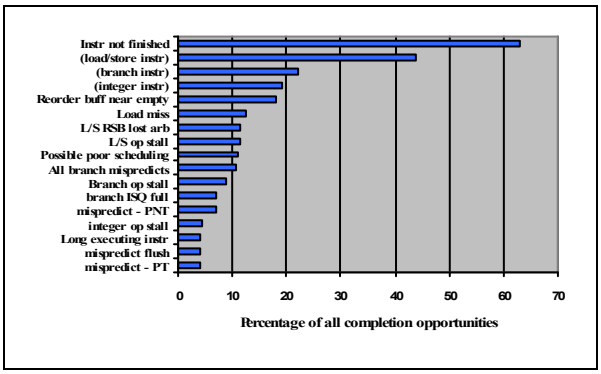

Figure 5. Example node summary of major completion stall contributors

The leaf summary displays the contribution of each leaf stall type to the total number of completion stalls. The twig summary is similar, but it also shows the stall events directly caused by each leaf stall (and their relative incidence), the stall events in turn caused by *those* events, and so forth, for the desired number of levels.

As noted previously, a particular type of stall event can contribute to the overall IPC loss through multiple pathways. So, to compute the leaf and twig summaries, we first invert the summary tree; the top level of the inverted tree is the leaf summary, and the top N levels are the twig summary.

Finally, to compute the node summary, we walk the hierarchical summary tree and accrue the total contribution for each of the stall types, both leaf and non-leaf. The flexibility of the Stallscope design also allows for the insertion of *informational* events into the causal stall chain; these events do not actually represent the occurrence of a stall, but instead provide additional useful information about the causal stall chain. For example, in our work, when any of the various types of branch mispredict stalls occurred, we found it useful to insert a general "branch mispredict" summary event in the stall chain in front of the specific branch mispredict stall event. This allowed us to easily read the total contribution from all branch mispredicts in the node summary.

However, informational events go beyond simple convenience. For example, we recently investigated potential improvements to the instruction scheduling algorithms. Heuristics were used to detect when a stall event was potentially unnecessary and avoidable.

When this occurred, a "poor scheduling" informational event was inserted into the stall chain behind the stall event. The subsequent node summary information was used to isolate the primary "hot spots" where scheduling improvements would have the greatest impact.

Figures 4 and 5 show examples of the data produced by the leaf and node summaries.

6. Inevitable complexities

In this section, we discuss some of the more advanced design challenges encountered during the development of Stallscope.

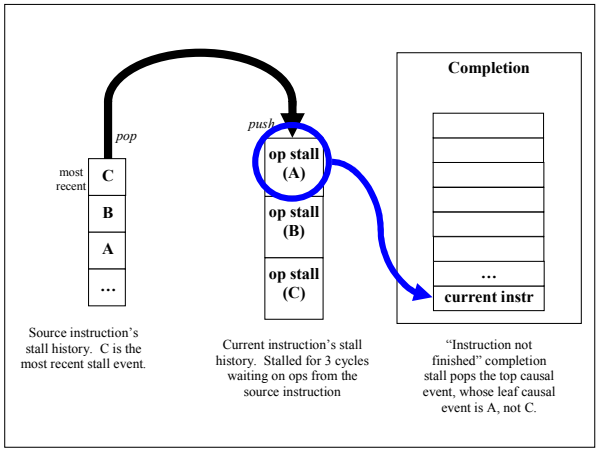

Figure 6. Stall history reversal

6.1. Stall history reversal

As noted previously, each cycle that an instruction is stalled waiting for its operands, the causal stall event is obtained by popping the next most recent stall event off of the source instruction's stall history. However, if the

operand stall persists for more than one cycle, then, as each stall event is popped off of the source instruction's stall history and pushed onto the stalled instruction's stall history, the order of the original stall events is reversed, as shown in Figure 6.

If the instruction is later stalled for a cycle in completion, the leaf stall at the end of the accumulated stall chain will be stall event A, the oldest leaf stall event, not the most recent (which is event C).

The solution for this is to replace the stall history stack with a moderately more complex structure. The new data structure incorporates the addition of a *consecutive stall queue* at the head of the stall stack. A sequence of consecutive operand stall events that depend on the same source instruction will be appended to the end of the head queue. When the sequence terminates (the instruction is no longer stalled), the contents of the head queue will be pushed onto the stall history stack, preserving the correct ordering.

6.2. Over-reporting stalls

Consider the case shown in Figure 7a, where instruction Y is a long executing instruction with data dependency on instruction X, but neither instruction encounters any execution stalls (in this case, we assume that X and Y are issued to different execution units, so we split the pipeline in the figure to show that the separate execution paths). Because Y spends an extra cycle in execution, completion is stalled for one cycle with the "long execution" stall event reported.

In Figure 7b, we see the case where instruction X is stalled in execution for one cycle due to causal event A, which is pushed onto X's stall history. Because X and Y were issued to different execution units, there is no structural hazard, but Y's data dependence causes it to stall for one cycle (in cycle N+2) waiting for data from X. The operand stall, with leaf causal event is event A, is pushed onto Y's stall history.

X's stall causes completion to stall for one cycle (in cycle N+2), and the "instruction not finished" stall event, whose stall chain ends with leaf causal event A, is accumulated.

In cycle N+4, Y is not finished and cannot complete because, as before, it is a long executing instruction and needs to spend two cycles in the execution stage. However, following the rules for completion stalls, since Y's stall history is not empty, the causal event is popped off of Y's stall history and another "instruction not finished" stall event whose stall chain ends with leaf causal event A is accumulated.

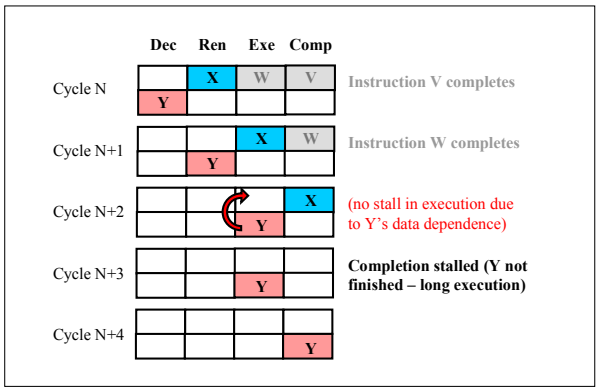

Figure 7a. No execution stall

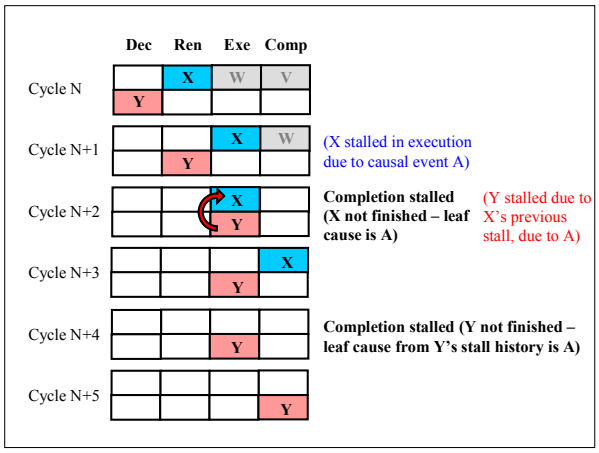

Figure 7b. Execution stall over-reported

Clearly, this is not correct, because event A can only stall the instruction stream for at most 1 cycle. Another way to look at it is that causal stall A did not change the number of cycles that Y was the oldest instruction. We can see that although Y now completes a cycle later than before, the cycle on which it becomes the oldest instruction in the reorder buffer is also delayed by a cycle due to X's stall. So, the effect on completion of causal event A *has already been accounted for*.

The solution is to constrain the stall accumulation in completion so that each unique leaf stall event is only accumulated once. If completion is stalled because the oldest instruction is not yet finished, then events are popped off of its stall history until one is found whose leaf causal event has not yet been accumulated. If, as in the case above, the stall history is exhausted without finding a stall event to accumulate, then the usual rules are followed and a "reorder buffer empty" or "long execution" stall event will be accumulated.

The only exception to this is that a pipeline flush can cause completion to stall for many cycles. So,

flush events are marked as *global* events, and are not subject to the unique accumulation rule.

6.3. The statistical tail – unbounded stall chains

During execution of long workloads, on rare occasions we encountered a case where the instruction stream contained a very long chain of instructions, each of which had a data dependency on the previous, and so the causal stall chain for an operand dependency could become very long (containing hundreds of thousands of stall events). Managing these very long stall chains slowed the model significantly and could consume a *huge* amount of memory.

These extremely long causal chains are generally uninteresting, as they typically contributed only a minute fraction of a percent to the overall number of stalls. So, it was reasonable to truncate extremely long causal chains. In practice, since the top-level summaries and leaf summaries were of primary interest, we preserved the beginning and end of the stall chains and "snipped out" the middle. Although this has the potential to distort the node summary results, testing showed that in practice these extremely long stall event chains are sufficiently rare that the difference is so small as to be imperceptible.

7. Stallscope usage

Here we discuss several problems that we were able to successfully address using Stallscope.

7.1. Identify bottlenecks

The most general use of Stallscope is to identify the major bottlenecks in the simulator. Analysis of the leaf and node summaries for all workloads showed that the three major contributors to performance loss were load misses, branch mispredicts, and stalls due to poor scheduling, in that order. Although this conclusion was independently confirmed by the microarchitects designing each of these subcomponents of the microprocessor, it is important to note that the Stallscope analysis was able to rigorously identify the bottlenecks for the entire CPU, without the need for focused, in-depth knowledge of any particular subcomponent.

In practice, Stallscope instrumentation of the model was an iterative process. Analysis of the intermediate results with a partially instrumented model allowed us to extend the instrumentation to the areas that exhibited the greatest contribution to performance loss and thus were of greatest interest.

7.2. Branch prediction anomalies

As mentioned previously, the Stallscope development was initially motivated by observed anomalies in the analysis of alternative branch prediction algorithms.

In the case where an increase in branch prediction rate was coupled with an IPC loss, although the unit statistics for the branch predictor showed that there were fewer branch mispredicts, a comparison of the leaf summaries for each configuration showed that there was an increase in the number of cycles that the branch mispredicts caused completion to become stalled. Further analysis of the pathways by which the mispredict stalls propagated to completion allowed us to conclude that, although there were fewer mispredicts, the new algorithm tended to shift the mispredicts to a point in the instruction stream where the pipeline was more vulnerable to front-end stalls and therefore had a greater impact on IPC. This sort of timing correlation might not have been evident without the insight that Stallscope provided.

7.3. Prefetching

During our investigation of a promising instruction prefetching algorithm, we observed the non-intuitive result that, for most workloads, instruction cache misses decreased, but overall performance also went down.

Stallscope revealed two causes for this effect. For some workloads, although there were fewer instruction cache misses with the new prefetching algorithm, these misses were more likely to occur at times when the pipeline was nearly empty, and so the front-end stalls were able to propagate to completion.

For other workloads, Stallscope revealed an increase in completion stalls that could trace their source to *data* cache misses. Analysis showed that, for these cases, with the new prefetching algorithm, the instruction cache misses now occurred at points in the instruction stream where they were more likely to be concurrent with data cache misses, and so they were now competing for the memory subsystem.

7.4. Scheduling analysis

Our bottleneck analysis showed that the scheduling algorithm sometimes led to poor choices that resulted in IPC loss. In the most extreme case, for one of the

workloads, Stallscope predicted that over 30% of the IPC loss was due to poor scheduling.

With some slight modification, Stallscope was enhanced to produce intermediate partial summaries at regular intervals. Once the "hot spots" where Stallscope predicted the greatest impact due to poor scheduling were identified, further analysis of these small, focused regions in the instruction stream with pipeline visualization tools allowed us to characterize these regions. The observation of common features in these parts of the instruction stream enabled us to design and implement a modified scheduling algorithm that capitalized on these characteristics.

The improved scheduling algorithm showed an across-the-board increase in performance. In nearly all cases, the Stallscope prediction was within 20-30% of the IPC improvement realized by the new algorithm. In the most extremely affected workload noted above, the new algorithm resulted in a 600% speedup for the identified hot spots, and an overall 35% increase in performance for the workload as a whole.

8. Related work

As discussed in the overview paper by Yi and Lilja [1], most current performance analysis is rudimentary, based on the evaluation of one or two key metrics; this only gives a "high-level, net-effect view picture" of the microarchitectural behavior.

However, some efforts are being made to improve performance analysis tools and methodologies. Tune et al. [2, 3] and Fields et al. [4] have explored the use of dynamic critical path prediction to improve processor performance. In this approach, a dependency graph between *execution* events is constructed, and the critical path is derived from this graph. In contrast, Stallscope tracks dependencies between *stall* events; all stall events are processed *locally*, so the potential overhead of a global dependency graph is avoided.

One phenomenon addressed in this paper is the failure of local unit analysis to identify stall conditions that ultimately have no effect on execution time. Fields, Bodík, and Hill [5] take a different approach to this issue through their *slack* analysis. This analysis builds on their previous critical path work by identifying timing paths that could be delayed without impacting the total execution time.

Stallscope only reveals the primary sources of IPC loss; it does not address whether or not the IPC loss could be fully recovered by eliminating the stall source; in some cases, the elimination of a primary stall source serves only to uncover a secondary stall source that had been masked by the primary. Fields et al. [6] introduce the interaction cost methodology to quantitatively evaluate this effect. Although, in practice, our tests with Stallscope showed that the greatest contributors to IPC loss often have only moderate overlap with secondary stall sources, tracking and collecting secondary stall sources in Stallscope is a promising area of further development in order to better capture more subtle effects.

Yi, Lilja, and Hawkins [7] use the statistical Plackett and Burman method to quantify the relative significance of the available parameters in a microarchitectural model. However, their approach and the interaction cost analysis noted above require the set of parameters or events of interest to be identified in advance. This can be a problem with our highly detailed cycle-accurate simulators that precisely model existing or future hardware; in our current model, there are over 70 different stall events that could potentially impact performance. While many of these will have negligible contribution to the overall execution time, it can be difficult to identify those that most strongly affect performance.

Both of these performance analysis methodologies can be complementary to Stallscope; Stallscope can potentially be used to identify the major bottlenecks in the system and determine which parameters to evaluate, while the P & B and interaction cost methodologies could be used to further analyze the sensitivity of the model to those parameters.

CPI Stack is a performance analysis tool that was independently developed at IBM and is fairly similar to Stallscope. Currently, CPI Stack has been implemented in simg5, the 970 cycle-accurate simulator [8]. A variant of CPI Stack that utilizes hardware performance counters has also been used for workload characterization [9].

9. Conclusions

Stallscope is a performance analysis tool that fills a much-needed role in modern microarchitecture exploration and development. It allows the true impact of tradeoffs at each stage of the pipeline to be quantified. In this paper, we presented the motivation and goals that guided the development of Stallscope, and discussed the details of Stallscope's structure and design, with focused attention on some of the more difficult design challenges.

We showed some examples and mentioned some of the successes we have had using Stallscope. These included our use of Stallscope to analyze and explain the counterintuitive results observed in the evaluation of branch prediction and instruction prefetching

algorithms, as well as the use of Stallscope in "hot spot" analysis of the existing instruction scheduling algorithms.

References

[1] Joshua Yi and David Lilja, "Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations," IEEE Transactions on Computers, Vol. 55, No. 3, March 2006.

[2] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder, "Dynamic prediction of critical path instructions," Proceedings of the Seventh International Symposium on High-Performance Computer Architecture, Jan 2001.

[3] Eric Tune, Dongning Tullsen, and Brad Calder, "Quantifying Instruction Criticality," The 11th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2002.

[4] Brian Fields, Shai Rubin, and Rastislav Bodík, "Focusing processor policies via critical-path prediction," Proceedings of the 28th Annual International Symposium on Computer Architecture, Jun– Jul 2001.

[5] Brian Fields, Rastislav Bodík, and Mark D. Hill, "Slack: Maximizing Performance Under Technological Constraints," 29th International Symposium on Computer Architecture.

[6] Brian Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn, "Using Interaction Costs for Microarchitectural Bottleneck Analysis," 36th International Symposium on Microarchitecture, 2003.

[7] Joshua Yi, David Lilja, and Douglas Hawkins, "A Statistically-Rigorous Approach for Improving Simulation Methodology," Proc. International Symposium on High-Performance Computer Architecture, 2003.

[8] Mark Szymczyk, "simg5," ADHOC/MacHack 20, 2005.

[9] Bill Maron, Thomas Chen, Duc Vianney, Bret Olszewski, Steve Kunkel, and Alex Mericas, "Workload Characterization for the Design of Future Servers," Proc. IEEE International Symposium on Workload Characterization, 2005, p. 129.