Increasing Processor Performance by Implementing Deeper Pipelines

Eric Sprangle, Doug Carmean
Pentium® Processor Architecture Group, Intel Corporation
eric.sprangle@intel.com, douglas.m.carmean@intel.com

Abstract

One architectural method for increasing processor performance involves increasing the frequency by implementing deeper pipelines. This paper will explore the relationship between performance and pipeline depth using a Pentium® 4 processor like architecture as a baseline and will show that deeper pipelines can continue to increase performance.

This paper will show that the branch misprediction latency is the single largest contributor to performance degradation as pipelines are stretched, and therefore branch prediction and fast branch recovery will continue to increase in importance. We will also show that higher performance cores, implemented with longer pipelines for example, will put more pressure on the memory system, and therefore require larger on-chip caches. Finally, we will show that in the same process technology, designing deeper pipelines can increase the processor frequency by 100%, which, when combined with larger on-chip caches can yield performance improvements of 35% to 90% over a Pentium® 4 like processor.

1. Introduction

Determining the target frequency of the processor is one of the fundamental decisions facing a microprocessor architect. While historical debate of pushing frequency or IPC to improve performance continues, many argue that modern processors have pushed pipelines beyond their optimal depth. With the fundamental debate raging, most agree that the engineering complexity and effort increases substantially with deeper pipelines. Focusing on single stream performance, and using the Pentium® 4 processor as a baseline architecture, this paper will conclude that pipelines can be further lengthened beyond the Pentium® 4 processor's 20 stages to improve performance. We assert that architectural advances will enable even deeper pipelines, although engineering effort and other considerations may be the real limiter.

2. Overview

We will propose a model to predict performance as a function of pipeline depth and cache size. First, we will determine the sensitivity of IPC to the depth of important pipelines. Then, we will describe how a cycle can be thought of as the sum of "useful time" and "overhead time", and that the frequency can be increased by reducing the amount of "useful time" per cycle. We will then show that deeper pipelines can increase the frequency to more than offset the decrease in IPC. We will then describe how execution time can be thought of as the sum of "core time" and "memory time" and show how "memory time" can be reduced with larger caches. Finally, we will show how the combination of deeper pipelines and larger caches can increase performance significantly.

3. Fundamental processor loops

Performance can monotonically increase with increased pipeline depth as long as the latency associated with the pipeline is not exposed systematically. Unfortunately, due to the unpredictable nature of code and data streams, the pipeline cannot always be filled correctly and the flushing of the pipeline exposes the latency. These flushes are inevitable, and pipeline exposures decrease IPC as the pipeline depth increases. For example, a branch misprediction exposes the branch misprediction pipeline, and the exposure penalty increases as the pipeline depth increases. The L1 cache pipeline can also be exposed if there are not enough independent memory operations sent to the L1 cache to saturate the pipeline. Of course, some pipeline latencies are more important than others. We simulated the performance sensitivities to the various loops on a Pentium® 4 processor like architecture to understand which loops are the most performance sensitive.

4. Simulation methodology

We conducted our experiments using an execution driven simulator called "Skeleton", which is a high level simulator that is typically used for coarse level architectural trade-off analysis. The simulator is layered on top of a uOp-level, IA32 architectural simulator that executes "Long Instruction Trace (LIT)"s. A LIT is not, as the name implies, a trace, rather it is a snapshot of processor architectural state that includes the state of system memory. Included in the LIT is a list of "LIT injections" which are system interrupts that are needed to

1063-6897/02 $17.00 © 2002 IEEE

simulate system events such as DMA traffic. Since the LIT includes an entire snapshot of memory, this methodology can execute both user and kernel instructions, as well as wrong path effects. Our simulation methodology uses carefully chosen, 30 million instruction program snippets to model the characteristics of the overall application.

Our simulations are based on a Pentium® 4 like processor described in Table 1. The results will be limited to the suites listed in Table 2 for a total of 91 benchmarks that are comprised of 465 LITs.

Table 1: Simulated 2GHz Pentium® 4 like processor configuration.

Core
3-wide fetch/retire
2 ALUs (running at 2x frequency)
1 load and store / cycle
In-order allocation/de-allocation of buffers
512 rob entries, load buffers and store buffers
Memory System
64 kB/8-way I-cache
8 kB/4-way L1 D-cache, 2 cycle latency
256 kB/8-way unified L2 cache, 12 cycle latency
3.2 GB/sec memory system, 165ns average latency
Perfect memory disambiguation
16 kB Gshare branch predictor
Streaming based hardware prefetcher

Table 2: Simulated Benchmark Suites

Suite	Number of Benchmarks	Description
SPECint95	8	spec.org
Multimedia	22	speech recognition, mpeg, photoshop, ray tracing, rsa
Productivity	13	sysmark2k internet/business/ productivity, Premiere
SPECfp2k	10	spec.org
SPECint2k	12	spec.org
Workstation	14	CAD, rendering
Internet	12	webmark2k, specjbb

5. Efficiency vs. pipeline depth

Figure 1 shows the relative IPC as the branch misprediction penalty is increased from 20 to 30 cycles. We can determine the average branch misprediction latency sensitivity by calculating the average IPC degradation when increasing the branch misprediction latency by one cycle.

It is interesting to note that SPECint95 is much more sensitive to the branch misprediction latency than the other application classes. To a lesser extent SPECint2k also shows greater sensitivity to branch misprediction latency than the other application classes. In this sense, SPECint95 in particular is not representative of general desktop applications because of the higher branch misprediction rates.

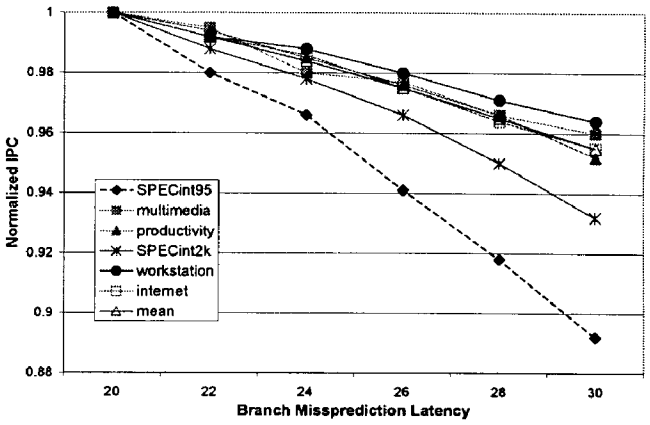

Figure 1: Normalized performance vs. branch misprediction latency.

To understand the sensitivity to the ALU loop latency, we started with a baseline processor that implements half clock cycle add operations, like the implementation in the Pentium® 4 processor. The Pentium® 4 processor pipelines the ALU operation into 3 "half" cycles: lower 16 bit ALU, upper 16 bit ALU, flag generation [2]. Figure 2 shows the effect of increasing the ALU latency from 1 half clock cycle to 3 full clock cycles while keeping the ALU throughput constant. Hence, for a workload that consists of independent ALU operations, we would expect to see no increase or degradation in performance, but for a stream of dependent ALU operations, execution time would increase linearly with the ALU latency.

In Table 3, we show the performance impact of adding an additional full cycle to a given loop. For example, the impact of increasing the ALU latency by a full clock cycle is 4.76%. As Table 3 shows, the ALU loop is, by far, the most performance sensitive loop on integer applications.

Table 3: Average percentage performance degradation when a loop is lengthened by 1 cycle.

Suite	ALU	L1 cache	L2 cache	Br Miss
SPECint95	5.64	0.72	0.32	1.08
Multimedia	3.84	2.08	0.54	0.40
Productivity	7.00	2.20	0.50	0.48
SPECfp2k	0.76	1.08	0.24	0.26
SPECint2k	4.96	2.56	0.90	0.68
workstation	3.16	2.64	0.82	0.36
internet	3.96	2.00	0.46	0.45
Average	4.76	2.04	0.54	0.45

It is important to note that the performance results are a strong function of algorithmic assumptions in the microarchitecture. For example, we would expect L2 cache sensitivity to be a function of the L1 cache size and branch misprediction latency sensitivity to be a function of the branch predictor.

We are also making the approximation that these sensitivities have a constant incremental impact on IPC for the pipeline length ranges we are interested in. For example, a path with a 10% sensitivity would drop performance to 90% on the first cycle and $(1\text{-}10\%)^2$ or to 81% on the second cycle.

Typically, a larger portion of the engineering effort allocated to a project is spent on the latency sensitive paths. The effort is spent developing aggressive architectural and circuit solutions to these paths, as well as careful analysis of the specific implementations. Solutions such as clustering[1] or slicing[2] are typically employed to limit performance degradation as pipeline frequency is increased.

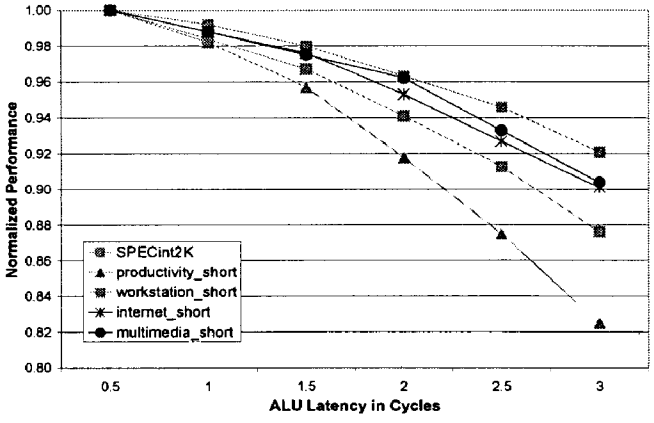

Figure 2: Performance vs. ALU latency.

6. Pipeline depth vs. frequency

Consider the branch misprediction pipeline in our Pentium® 4 like processor. The 20 stage misprediction pipeline includes the time required for a branch instruction to be issued, schedule, resolved and send a signal back to the front end to redirect the instruction stream.

A 2GHz Pentium® 4 processor has a 500ps cycle time, with a portion of the cycle used for skew, jitter, latching and other pipeline overheads. The cycle time that is not used for pipeline overhead is then dedicated for useful work. Assuming that the pipeline overhead per cycle is 90 ps, one can calculate the total "algorithmic work" associated with the branch misprediction pipeline as the number of stages * useful work/stage, or (20 stages* (500ps – 90ps) = 8200 ps of algorithmic work in branch miss loop).

In these calculations, we have included the communication time in the "useful work" component of the cycle time. The communication time includes the latency of wire delays, and therefore the "useful work" in a path is a function of the floorplan. This is particularly relevant in areas such as the branch misprediction loop, where the latency of driving the misprediction signal from the branch resolution unit to the front end becomes a key component of the overall loop latency

If we assume that the overhead per cycle is constant in a given circuit technology, in this case 90 ps, we can increase the processor frequency by reducing the "useful time" per cycle. As the useful time per cycle approaches zero, the total cycle time approaches the "overhead time". Because of the constant overhead, the frequency does not approach infinity but rather 1/90ps or 11.1GHz as shown in Figure 3. There are many other practical limits that would be reached before 11GHz, some of which will be discussed later.

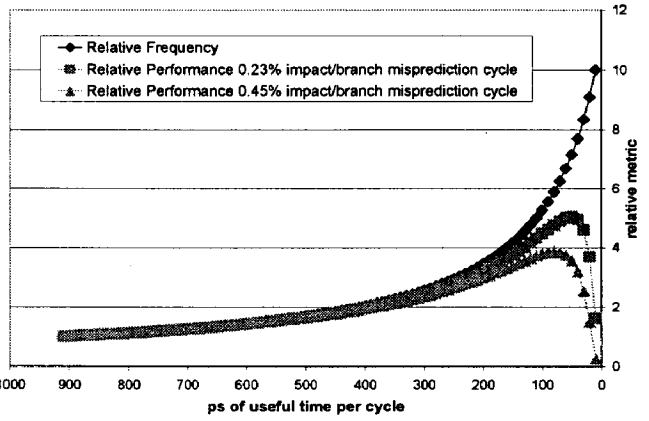

Figure 3: Frequency and relative performance vs. ps of useful time per cycle.

As we increase the pipeline depth, the staging overhead of the branch misprediction pipeline increases, which increases the branch misprediction latency, effectively lowering the overall IPC. If we assume IPC degrades by 0.45% per additional branch misprediction cycle (the average branch loop latency sensitivity from Table 3), we can calculate the overall performance as a function of "useful time" per cycle, as shown in Figure 3. (note that we are comprehending the efficiency impact of increasingly *only* the branch misprediction latency in this example).

As expected, the overall performance degrades when the decrease in IPC outweighs the increase in frequency. From Figure 3 we see that performance tracks closely with frequency until the useful time per cycle reaches about 90 ps, which equates to a cycle time of 180ps. As a point of comparison, the 180ps cycle time is roughly half the cycle time of a Pentium® 4 processor. If we assume that the sensitivity is cut in half, or 0.23% per additional branch misprediction cycle, we see that the potential overall performance increase is higher and the optimal point has a smaller useful time per cycle.

7. Off-chip memory latency

It is important to note that in the proceeding analysis, the percentage of time waiting for memory was held constant. This was a simplification that is technically incorrect, as the percentage of time waiting for memory increases as the core performance increases. This assumption does not change the optimal frequency, as minimizing the core time will minimize the overall program execution time. Further, we will show that the percentage of time that is spent waiting for memory can be reduced by increasing the size of the on-chip L2 cache. In subsequent discussions, we will show that the cache miss rate will decrease as the square root of the increase in L2 cache size.

8. Pipelining overhead

In the Pentium® 4 processor, the clock skew and jitter overhead is about 51ps [3]. In a conservative ASIC design, the overhead is the sum of the clock skew and jitter combined with the latch delay. In a standard 0.18um process, a typical flop equates to about 3 FO4 delays, with the FO4 delay being about 25ps [5]. Therefore in a 0.18um process, pipeline overhead would come out to about 75ps + 51ps = 125ps. In a custom design flow, most of the clock skew and jitter overhead can be hidden by using time borrowing circuit techniques. Time borrowing uses soft clock edges to reduce or eliminate the impact of clock skew and jitter [4] which would yield a 75ps pipeline overhead. In an extreme custom design style, the flop overhead could be reduced by using techniques like pulsed clocks and/or direct domino pipelines, yielding a sub-50ps pipelining overhead at the cost of a much larger design effort.

At the extreme edge of pipelining, here defined as a cycle time of less than 300ps in a 0.18um process, the design effort increases rapidly because of the minimum/maximum delay design windows that arises as the pipeline cycle is reduced. The minimum delay must always be larger than the sum of clock skew and jitter + latch hold time. If this constraint is not honored, then the output of combinational logic may be lost, through transitions, before it can be latched [4].

We will assume that most pipeline interfaces can be at least partially time borrowed and therefore use an average overhead of 90ps per cycle, which is the nominal overhead assumed on Pentium® 4 processor[5].

Note that, in the past, the global skew has been kept under control through better circuit techniques. For example, the Pentium® Pro processor global skew was 250ps [6] with an initial cycle time of 8000ps (3.1% cycle time) and the Pentium® 4 processor used a global skew of 20ps [3] with an initial cycle time of 667ps (3.0% of cycle time). However, this paper will assume overhead does not scale with frequency, and we will use 90ps as a baseline overhead time

9. The limits of pipelining

Implicit in our pipeline scaling analysis is that the pipeline depth can be arbitrarily increased. While this assumption is generally true, the complexity associated with increasing pipelines increases rapidly in some of the fundamental loops. Some of the pipelines in a processor include "loops" where a stage requires the result of the previous stage for execution. In these loops that require value bypassing between stages, any latency increase will directly reduce the processor's overall IPC. As we have shown, some of the loops are more critical than others (especially the ALU loop, L1 cache latency loop and branch misprediction loop). We will look at a case study to better understand the fundamental limits of pipelining.

9.1. Pipelining the RAT

The Pentium® 4 processor register renaming algorithm is similar to those implemented in other out-of-order processors, such as the Alpha 21264 [1]. The register renaming algorithm involves several steps where architectural registers are mapped to physical registers. The first step requires that the destination register in a given uOP is mapped to a physical register. Then, a mapping process renames all the register sources in the uop to the physical registers assigned to the previous uop that generated this particular register instance. A "Register Alias Table" (RAT) holds the mapping from

architectural to physical register. Algorithmically, a uop reads the RAT to determine the physical register for each of its architectural source registers and then writes the RAT to record the physical location of its architectural destination. The next uop (in program order) reads and writes the RAT and so on. In this scheme, it is possible for a uop source to match the destination of the previous uop.

If we stretch the pipeline so that an update to a RAT entry followed by a read to the same entry takes 2 cycles, then a level of bypassing is needed to cover the write to read latency as seen in Figure 4, 2 stage pipeline. The multiplexer that is used to implement the bypassing increases the amount of useful work to cover the additional latency. If the pipeline is further stretched to 3 cycles, then an additional bypass stage is needed (Figure 4, 3 stage pipeline), but because the bypass is done in parallel with the RAT, the amount of useful work in the critical path does not increase. As we increase the depth of the pipeline in the RAT, the amount of useful work increases when going from 1 to 2 cycles, and then remains constant thereafter. This is because once we include a final bypass mux, we do not need to add additional bypass muxes in the rename path as the pipeline depth is increased.

What is the limit of pipelining for the RAT? Eventually the number of muxes needed to cover the write to read latency in the RAT causes the delay through the muxes to be larger than the delay through the RAT (Figure 4, 4 stage pipeline). When this happens, the amount of useful work as we go through the path is increased again. At this point we can continue to increase the depth of the pipeline, but at the expense of increased latency

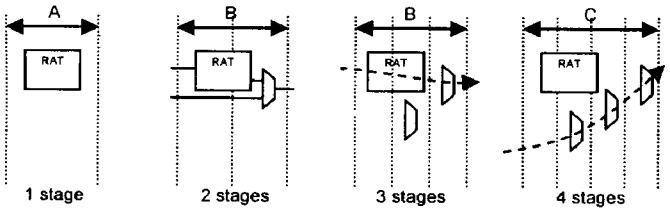

Figure 4: Pipelining the RAT

As we continue to increase the depth of the global pipeline, the next interesting challenge is posed by the register file. When the depth of the register file pipeline increases to the point where individual components of the array access need to be pipelined, it becomes convenient to add a latch immediately after the bit-line sense amplifiers. However, adding a latch in the word-line access or within the bit-line drive becomes very problematic. Rather than attempting to add a latch within a word or bit line, the preferred method is to partition the structure into one or more pieces. Our analysis implicitly assumes we can overcome this problem through partitioning, or some other means, and that all of the pipelines that are scaled do not add useful work to the critical path.

9.2. Pipelining wires

There are plenty of places in the Pentium® 4 processor architecture where the wires were pipelined [9]. While it is straightforward to calculate the percentage of the processor that can be reached in a cycle, it is relatively uninteresting, as there is an existence proof that pipelining wires is an effective mechanism to overcome intrinsic wire latency.

10. Overall performance vs. pipeline depth

We can estimate overall performance vs. pipeline depth when using the same fundamental algorithms implemented in the Pentium® 4 processor architecture. We will assume we can pipeline the next fetch address generation loop (through architectural techniques, for example [7][8]) and the renaming loop without increasing the latency for back to back operations. However, the L1/L2 cache access time as well as the branch misprediction latency will increase. We will also assume that the ALUs in Pentium® 4 processor are running at the minimum possible latency, and that higher frequency designs will require additional latencies.

Based on these assumptions, we can build a model to estimate performance vs. pipeline depth. We will quote the branch misspredict pipeline depth, but we will scale all 4 of the critical loops. For example, we can calculate the frequency of a processor with a 50 stage branch misprediction loop by dividing the total useful time in the loop (assuming 90ps overhead per stage) as 20 stages * (500 ps – 90ps) = 8200ps. Dividing the total algorithmic time of 8200ps by 50 stages implies 164 ps useful time per stage. Adding back the 90ps overhead gives us a cycle time of 254ps for a frequency increase of 96%.

We can calculate the L1 cache latency in cycles by first calculating the algorithmic work as 2 stages * (500 ps – 90ps) = 820ps. Dividing this algorithmic work by the new "useful time" per cycle gives 820ps/164ps per stage or 5 stages. We can calculate the IPC impact of these loops by calculating all of the new loop latencies in cycles and calculating the degradation in IPC due to the increase of these individual loops. Taking the product of the individual components of IPC degradations gives the overall IPC degradation. Multiplying the new frequency by the new IPC gives the final performance curve vs pipeline depth as shown in Figure 5.

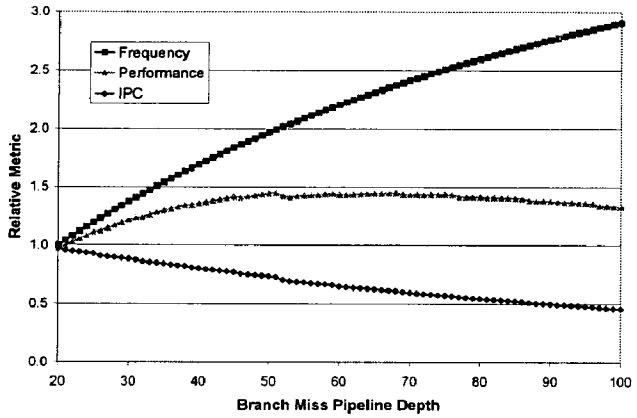

Figure 5: Frequency/IPC/Performance vs. branch misprediction pipeline depth.

Figure 5 shows that performance continues to increase as the pipeline is stretched up until the frequency is about doubled, which occurs when the branch misprediction pipeline reaches about 52 stages. The dips in the IPC and overall performance curves are due to the non-smooth nature of increasing pipeline depth – our analysis assumes that you can't add less then a full cycle to a pipeline.

Table 4 shows the individual loop lengths for a processor running at twice the frequency. Given these lengths, we can also calculate the individual IPC degradations due to each of the 4 loops, as well as the overall IPC degradation multiplier, as shown in Table 4. The relative IPC for each loop is calculated as $(1\text{-sensitivity})^{\text{increase in cycles}}$. The overall relative IPC is the product of the individual relative IPCs.

Even though the branch misprediction pipeline has the least per clock performance sensitivity, the absolute length of the branch misprediction pipeline makes it the loop with the single largest contribution to IPC loss. In a "from scratch" processor design, there is flexibility to change the fundamental algorithms that influence IPC. For example, there is opportunity to reduce IPC degradation by reducing the impact of branch misses through improved branch prediction.

Table 4: Pipeline lengths for a hypothetical Pentium® 4 like processor that runs at twice the frequency.

Loop	Pipeline Length	2x Freq Length	Sensitivity/ cycle	Relative IPC
ALU	0.5	1	4.76%	98%
L1 cache	2	4.5	2.04%	95%
L2 cache	12	32	0.54%	90%
Br Miss	20	52	0.45%	87%
Overall				72%

To validate the assumptions that the overall IPC degradation can be computed as the product of the individual degradations, we performed simulations at multiple effective frequencies, using the same methodology outlined above to generate the pipeline lengths.

Figure 6 shows the performance vs. pipeline depth as predicted by the analytical model and those produced by the performance simulator. As the data in the following chart shows, the simulated results align very closely with those produced by the analytical model for the pipeline depths of interest.

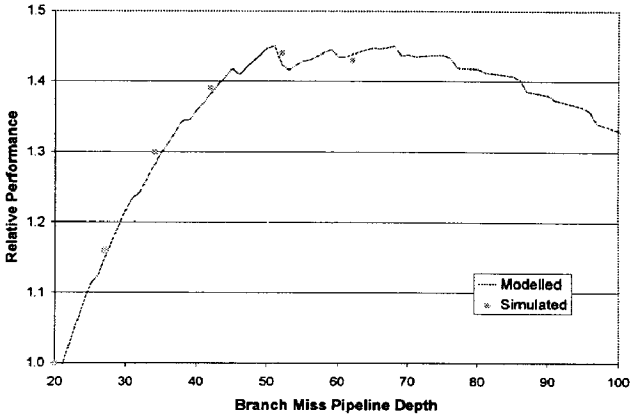

Figure 6: Simulated vs. modeled performance vs. branch misprediction pipeline depth.

11. Decreasing the impact of branch misses

There are many architectural and implementation methods for decreasing the branch misprediction penalty. Obviously, a more accurate branch predictor would decrease the IPC impact of additional cycles by reducing the number of times the branch misprediction loop is exposed. Alternatively, by implementing different architectural algorithms, a design can reduce the amount of useful work in the loop. For example, by more aggressively pre-decoding of instructions, perhaps by implementing a trace cache, a design can employ simpler, lower latency decoders, which reduces the algorithmic work in the branch misprediction loop.

In addition to reducing the algorithmic work, methods could try to reduce the "useless time" in the branch misprediction loop. For example, by implementing the front end to be twice as wide, and run at half the frequency, the amount of clock skew and jitter and latch delay associated with the loop is reduced while keeping the bandwidth the same (assuming the instruction fetch units that are twice as wide can really produce twice the number of uops per cycle). Another method that could reduce clock skew and jitter overhead involves using multiple clocks with smaller clock skew and jitter

overheads within a clock domain, and larger overhead between clock domains.

Finally, designers can tune the speedpaths detected on silicon (by resizing transistors, and rearranging floorplans etc) which might exist because of the difficulties associated with identifying the speed paths pre-silicon. Determining speed paths beyond a given accuracy increases quickly because of complex interactions that determine speed path latencies. These interactions include in-die process variation, interconnect coupling, and the false path elimination problem (many of the speed paths that a tool may detect are "don't care" scenarios).

In our analytical model, we can estimate the upper bound potential of removing the branch misprediction penalty by eliminating the IPC degradation due to the longer branch misprediction penalty. Figure 7 shows the scaling benefits if the branch misprediction penalty could be completely removed, raising the performance increase potential from 45% to 90%. Deeper pipelines will increase the opportunity for new architectural techniques to improve performance.

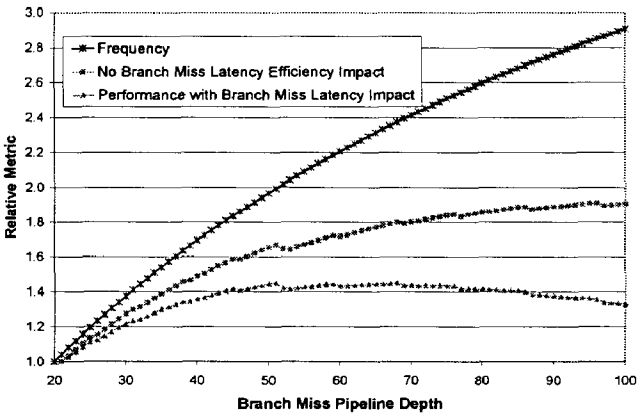

Figure 7: Freq/efficiency/performance vs. branch misprediction pipeline depth

Another way of improving the scaling of pipelines is to reduce the overhead due to pipelining (latch, clock skew, jitter, etc) through improvements in circuits and design methodologies. Moving from the 90ps overhead to a 50ps overhead, which is potentially achievable in an extreme custom design, the model predicts that the potential speedup improvement increases from about 45% to 65%, and as expected the optimal pipeline depth increases as the pipeline overhead is reduced, as shown in Figure 8.

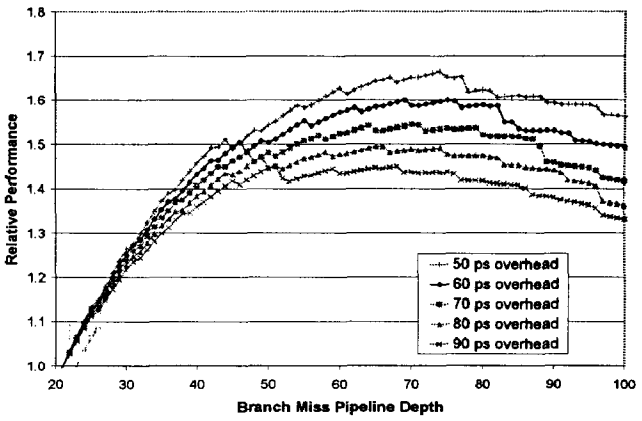

Figure 8: Performance vs. pipeline depth vs. pipeline overhead

12. Percentage time waiting for main memory

Program execution time can be broken into the "core time" (which scales with processor frequency) and "off-chip time" which is associated with off chip memory latencies. As we explained, our previous calculations have assumed that the percentage of time waiting for memory remains constant as the core performance is improved, which is wrong.

We can calculate the "core time" by running an application on two systems that differ only in processor frequency. We can get the performance of SPECint2k on multiple Pentium® 4 processors run between 1.5Ghz and 2GHz (we need to make sure the compiler does not change when quoting these numbers).

Figure 9 compares SPECint2k base (run on the Intel D850GB motherboard) for the Pentium® 4 processor vs. perfect scaling and shows that the Pentium® 4 processor converts about 65% of frequency increase into performance improvement. This degradation should be about the same for similar modern CPUs that use 256kB caches and have about the same performance (a quick analysis SPEC reported scores will confirm this). Therefore we can conclude that SPECint2k spends about 35% of its time waiting for main memory. This is important because it indicates the upper bound speedup achievable is 1/0.35 or 2.85x assuming we don't reduce the "off-chip time" (for example by improving the prefetching algorithms or increasing the size of the L2 cache).

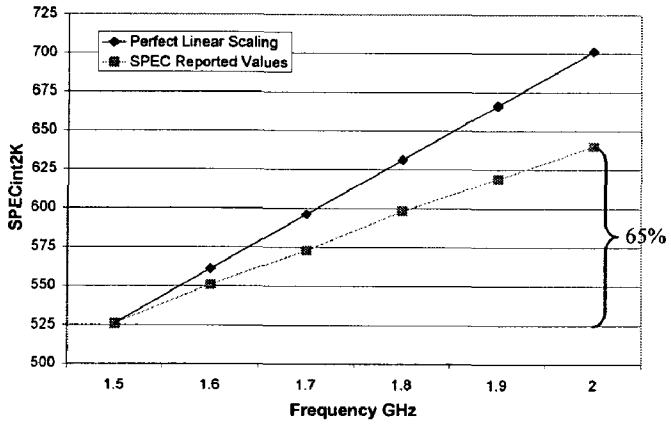

Figure 9: SPECint2K vs. frequency

We can calculate the percentage of time waiting for memory for each of our benchmark suites by varying the frequency of the simulated processor plus and minus 200 MHz, and running the LITs. Table 5 shows the percentage of time spent in core on the various benchmark suites on our base processor configuration. Notice that the percentage of core time generated by the simulator for SPECint2k matches closely the values reported to SPEC (65% calculated from SPEC numbers vs. 67% for the simulated results) which gives some confidence that the simulator is reasonably modeling the off chip memory system (bandwidth, latencies and prefetcher algorithms).

Table 5: Percentage of time spent in core calculated by simulating at 2 different frequencies.

Suite	% of core time
SPECint95	79
SPECint2k	67
Productivity	79
Workstation	76
Internet	70
Multimedia	74
SPECfp95	66
SPECfp2k	66
Average	72

13. Performance vs. cache size

To this point, all of the simulations have used the Pentium® 4 processor 256kB L2 cache configuration. As the frequency (performance) of the core is increased, the percentage of time spent waiting for memory increases. A common rule of thumb says that quadrupling the cache size will halve the miss rate of the cache. Figure 10 and Figure 11 show that this rule of thumb is quite accurate for cache size ranges from 0.5 kB to 8 MB for SPECint2k and Sysmark2K. Since 30 million instruction traces might not be long enough to warm up an 8 MB cache, these simulations were done using much longer traces. While it is true that an individual benchmark can fit in a given cache size, the benchmark suites show that the rule of thumb holds for the average across all benchmarks in the suite.

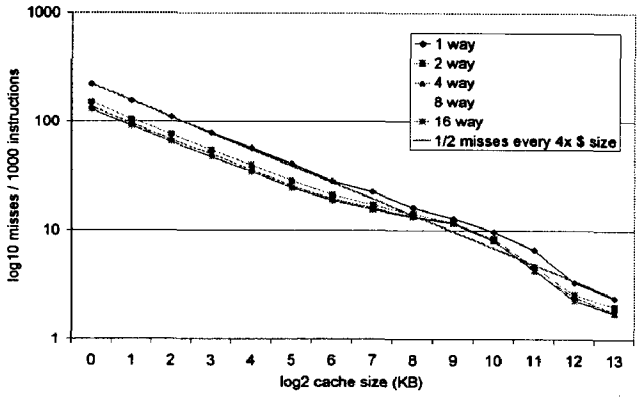

Figure 10: L2 cache misses/1000 instructions (SPECint2k average)

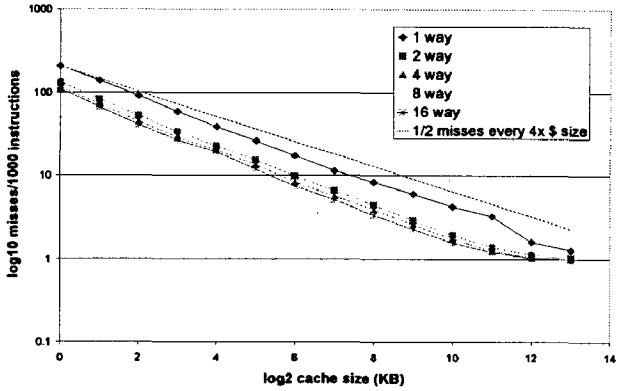

Figure 11: L2 cache misses / 1000 instructions (Sysmark2k average)

14. Increasing pipeline depth *and* L2 cache

Assuming that the miss rate decreases as 1/(sqrt (cache size)) as Figure 10 and Figure 11 suggest, then we can hold the percentage of time waiting for memory constant if we quadruple the size of the cache every time we double to core performance. We can apply this rule of thumb to our pipeline scaling model to estimate the speedups possible by both increasing pipeline depth and increasing L2 cache size as shown in Figure 12.

Assuming that 72% of the time scales with frequency with a 256kB cache, (the overall average for our benchmarks shown in Table 5), we can assume 28% of the time spent waiting for memory and will scale as 1/sqrt(cache size). The data shows that speedups of about 80% are possible when the pipeline is stretched to double frequency and L2 cache size is increased from 4MB to 8MB. Notice that the optimal pipeline depth is not a function of cache size, which makes sense because minimizing the core time is independent of minimizing the memory time.

In this analysis, we are not increasing the L2 cache latency as we increase the size, which is incorrect. Some portion of the L2 latency is a function of the L2 cache size.

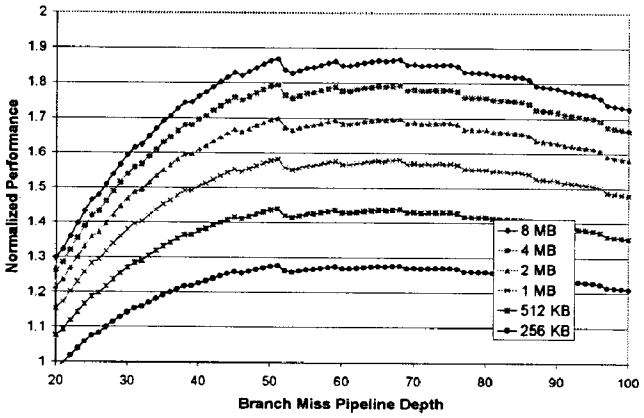

Figure 12: Performance vs. pipeline depth for different L2 cache sizes.

The next chart takes the same data and normalizes each of the cache size configurations so we can extract the performance improvement due only to the increase in pipeline depth. The data shows that increasing the pipeline depth can increase performance between about 30 and 45%, depending on how much of the speedup is watered down by waiting for the memory system.

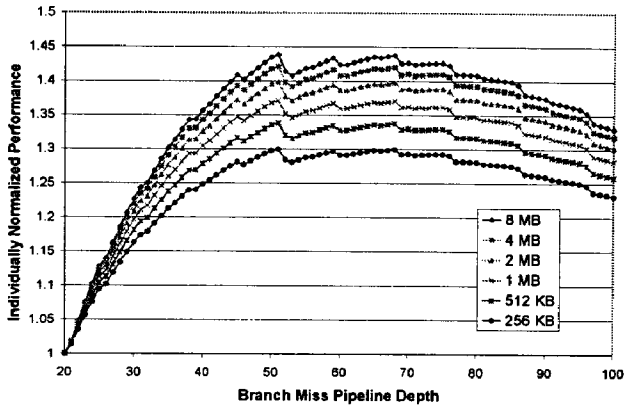

Figure 13: Performance vs. pipeline depth vs. L2$ size normalized.

15. Difficulties with deeper pipelines

Of course there are many issues associated with deeper pipelines that are beyond the scope of this paper. For example, *deeper pipelines* may imply more complex algorithms. On the other hand, *wider machines* may also imply more complex algorithms, so one might conclude that higher performance implies more complex algorithms. Given that we are attempting to build a higher performance processor, the fair question is "when is it easier to achieve higher performance through width vs. deeper pipelines?" The answer to this question may differ based on which part of the processor is being analyzed. For example, on the Pentium®4 processor, the answer in the fetch unit of the processor was presumably "wider is easier to achieve performance" since the fetch unit of the processor runs at half of the base frequency and achieves throughput by increasing width. This also makes sense in light of the relatively low branch misprediction latency sensitivity. The front end needs total bandwidth through whatever means possible and is less concerned with latency. On the other hand, the high sensitivity to latency in the execution core motivated running this piece of the processor at twice the base frequency.

There are many other problems associated with deeper pipelines. Deeper pipelines will put more pressure on accurate timing tools. New algorithms may need to be developed which will increase the number of interactions that need to be validated. More accurate architectural simulators will be needed to model those interactions to estimate performance and tune the architecture. Increasing performance through deeper pipelines will also increase power (although wider machines will also increase power).

16. Pipeline scaling and future process technologies

To a first order, increasing frequency by stretching the pipelines and increasing frequency by improving process are independent. Some components of skew and jitter will scale with process but some may not. Wires will not scale as fast as transistors [10], so wire dominated paths will need to be stretched even further (even an equivalent architecture, migrated to a future process, will require re-pipelining).

17. Conclusion and future directions

A simple model was discussed to predict processor performance as a function of pipeline depth and cache size. The model was shown to correlate to a simulator, and the simulator was shown to correlate to submitted SPEC results. Based on this model, we show that processor performance can in theory be improved relative to the Pentium® 4 processor by 35 to 90% by both increasing pipeline depth and cache size.

This paper argues that pipelines can be further optimized for performance given current architectural and circuit understanding. Better architectural algorithms and circuit techniques will increase the benefit of pipeline scaling. For example, SMT, which increases parallelism, should improve pipeline scaling. There are many exciting engineering challenges associated with deeper pipelines that will keep architects and designers entertained for years to come.

18. References

[1] R.E. Kessler. "The alpha 21264 microprocessor." IEEE Micro, 19(2):24-36, March/April 1999

[2] D. Sager et al., "A 0.18-um CMOS IA-32 microprocessor with a 4-GHz integer execution unit." In ISSCC Dig. Tech. Papers, February 2001, pp. 324-325

[3] N. Kurd et. al., "A Multigigahertz Clocking Scheme for the Pentium® 4 Microprocessor," in ISSCC Dig. Tech. Papers, February 2001, pp. 404-405.

[4] D. Harris, "Skew-Tolerant Circuit Design," Academic Press

[5] Personal communications with Rajesh Kumar, Pentium Processor Circuit Group, Intel

[6] R. P. Colwell and R. L. Steck. "A 0.6um BiCMOS processor with dynamic execution," International Solid State Circuits Conference (ISSCC) Digest of Technical Papers, pages 176-177, February 1995.

[7] Andre Seznec, Stephan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-Block Ahead Branch Predictors. In Proceedings of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pages 116--127, Cambridge, Massachusetts, October 1996.

[8] D. H. Friendly, S. J. Patel and Y. N. Patt. Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism. Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December, 1997.

[9] Personal communications with David Sager, Pentium Processor Architecture Group, Intel

[10] M. Horowitz, R. Ho, and K. Mai. "The future of wires." In Proceedings of the Semiconductor Research Corporation Work-23 shop on Interconnects for Systems on a Chip, May 1999.