The Optimum Pipeline Depth for a Microprocessor

A. Hartstein and Thomas R. Puzak
IBM - T. J. Watson Research Center
Yorktown Heights, NY 10598
hart@watson.ibm.com
trpuzak@us.ibm.com

Abstract

The impact of pipeline length on the performance of a microprocessor is explored both theoretically and by simulation. An analytical theory is presented that shows two opposing architectural parameters affect the optimal pipeline length: the degree of instruction level parallelism (superscalar) decreases the optimal pipeline length, while the lack of pipeline stalls increases the optimal pipeline length. This theory is tested by analyzing the optimal pipeline length for 35 applications representing three classes of workloads. Trace tapes are collected from SPEC95 and SPEC2000 applications, traditional (legacy) database and on-line transaction processing (OLTP) applications, and modern (e. g. web) applications primarily written in Java and C++. The results show that there is a clear and significant difference in the optimal pipeline length between the SPEC workloads and both the legacy and modern applications. The SPEC applications, written in C, optimize to a shorter pipeline length than the legacy applications, largely written in assembler language, with relatively little overlap in the two distributions. Additionally, the optimal pipeline length distribution for the C++ and Java workloads overlaps with the legacy applications, suggesting similar workload characteristics. These results are explored across a wide range of superscalar processors, both in-order and out-of-order.

1. Introduction

One of the fundamental decisions to be made in the design of a microprocessor is the choice of the structure of the pipeline. In this paper we explore the question as to whether or not there an optimum pipeline depth for a microprocessor that gives the best performance. The problem is treated both analytically and by simulation. We show that there is indeed an optimum pipeline depth and determine the values of the parameters that yield this optimum performance. We also find that we can understand the results in an intuitive way. The optimum depth is found to depend on the detailed microarchitecture of the processor, details of the underlying technology used to build the processor, and certain characteristics of the workloads run on the processor. Essentially, there is a tradeoff between the greater throughput of a deeper pipeline and the larger penalty for hazards in the deeper pipeline. This tradeoff leads to an optimum design point.

These questions have been addressed previously, but not in the coherent theoretical framework that we employ here. Kunkel and Smith [1] considered the impact of pipeline depth, in the context of gate delay/segment, on the performance of a scalar processor, specifically addressing the effect of latch delays. Recently, Agarwal, et.al. [2] in analyzing microarchitecture scaling strategies, employed simulations, similar to ours, but considered combinations of pipeline and microarchitectural changes, which didn't allow them to elucidate the various dependencies observed. The most coherent previous treatment was given by Emma and Davidson [3]. They provide a theoretical treatment for an in-order scalar processor without the detailed simulations to test their theoretical predictions. Our theoretical treatment includes both out-of-order and superscalar processes, and we provide detailed simulations that confirm our predictions.

A simple intuitive way to see that performance will be optimal for a specific pipeline depth is to consider the changes in the CPI (Cycles / Instruction) and the cycle time of a processor as we change pipeline depth. In a qualitative way we can see that increasing the depth of the pipeline will increase the CPI as illustrated in Fig. 1 for a particular design. This is simply because each instruction must pass through more processor cycles for a deeper pipeline design. Our detailed simulations show that this increase is fairly linear, although this point is not necessary for our argument. At the same time the cycle time of the processor decreases

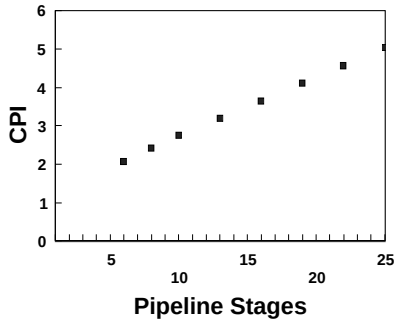

Fig. 1 shows the dependence of the CPI on the number of pipeline stages.

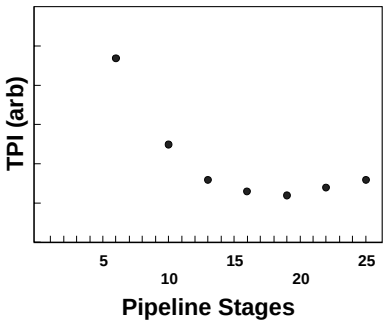

Fig. 3 shows TPI as a function of pipeline depth. The TPI scale is arbitrary.

as the pipeline depth is increased. This is an obvious result in that the amount of work does not increase with pipeline depth, so the total time needed to perform the logic functions of the processor stays the same, but the number of pipeline stages available to perform the work increases. This means that each stage can take less time. We will come back to this point later in a more formal way, but for now the result is shown qualitatively in Fig. 2.

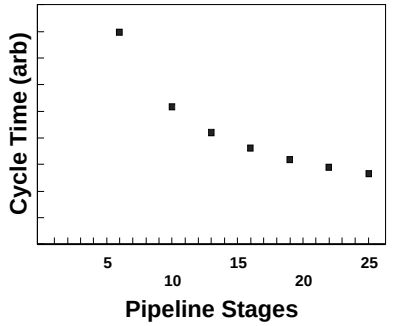

Fig. 2 depicts the dependence of the cycle time on the pipeline depth.

The true measure of performance in the processor is the average time it takes to execute an instruction. This is the Time / Instruction (TPI), which is just the inverse of the MIPs (Million Instructions per second) number, used to report the performance of some processors. The TPI is just the product of the cycle time and the CPI. This is shown qualitatively as a function of pipeline depth in Fig. 3. The optimum performance is then given as the minimum TPI point in the figure.

In the next section we present a formal theory of performance as a function of the pipeline depth. In Section 3 we introduce a methodology, which allows for the simulation of a variable pipeline depth microprocessor. In Section 4 this simulator model is used both to verify the formal theory and to quantify these results. Section 5 summarizes the differing results obtained for different workloads; and Section 6 contains further discussion.

2. Theory

In order to develop an expression for the optimum pipeline depth, we start by considering how a processor spends its time. The total time, T, is divided into the time that the execution unit is doing useful work, T_{BZ}, and the time that execution is stalled by any of a number of pipeline hazards, T_{NBZ}. This notation stands for the busy and not-busy times, which have been discussed previously [4, 5]. Typical pipeline stalls include branch prediction errors and both decode and execution data dependencies.

Let us first consider the busy time for the execution unit. The time that the e-unit is busy can be expressed in terms of the number of instructions, N_I, and the time for an instruction to pass each stage of the pipeline, t_S. The busy time is given by

$$T_{BZ} = N_I t_S \qquad (1)$$

for a scalar machine. Each pipeline stage delay can be expressed as

$$t_S = t_o + t_p/p, \qquad (2)$$

where t_p is the total logic delay of the processor, p is the number of pipeline stages in the design, and t_o is the latch overhead for the technology, being used. Eq. 2 says that the stage delay has two components, the portion of the total logic delay assigned to each pipeline stage and the latch overhead between pipeline stages. Substituting Eq. 2 into Eq. 1, we arrive at an expression for the e-unit busy time.

$$T_{BZ} = N_I(t_o + t_p/p), \qquad (3)$$

So far we have only considered a processor that executes instructions one at a time. For a superscalar processor multiple instructions may be executed at the same time. Therefore, less busy time is required to execute the instructions in a program. We model this by introducing a

parameter, α, which is a measure of the average degree of superscalar processing whenever the e-unit is busy. Eq. 3 is then modified as follows.

$$T_{BZ} = \frac{N_I}{\alpha}(t_o + t_p/p) \qquad (4)$$

It should be noted that α is not the superscalar issue width. Rather it is the actual degree of superscalar processing averaged over a particular piece of code. As such it varies with the workload running on the processor. α is obtained only when the e-unit is busy; the not busy times being caused by various hazards, to which we will now direct our attention.

Let us first consider the simple case in which each hazard causes a full pipeline stall. In this case the not busy time will be given in terms of the number of hazards, N_H, and the total pipeline delay, t_{pipe}.

$$T_{NBZ} = N_H t_{pipe} \qquad (5)$$

The total pipeline delay is just the product of each pipeline stage delay, t_S, and the number of pipeline stages in the processor. Combining this fact with Eqs. 2 and 5, we obtain the following expression for the not busy time.

$$T_{NBZ} = N_H(t_o p + t_p) \qquad (6)$$

In most processor designs this is too crude an approximation. Each particular pipeline hazard only stalls the pipeline for some fraction of the total delay. Execution dependencies will stall the pipeline less than decode (address generation) dependencies. Different types of branch misprediction hazards will stall the pipeline for different times. We can modify Eq. 6 to reflect this fact by allowing each hazard to have its own not busy time, t_{hazard}, which is a fraction, β_h, of the total pipeline delay. Then the expression for the not busy time must be summed over the not busy time for each individual hazard. Eq. 6 then becomes

$$T_{NBZ} = \sum^{N_H} t_{hazard} = N_H(t_o p + t_p)(\frac{1}{N_H} \sum^{N_H} \beta_h) \quad (7)$$

Some comments are in order. The parameter, β_h, measures the fraction of the total pipeline delay encountered by each particular hazard. As such its value is constrained to the range 0 to 1. We have deliberately grouped the variables in Eq. 7 to form the last expression in parenthesis. This is a dimensionless parameter, which we will call $\gamma = \frac{1}{N_H}\sum \beta_h$. This parameter, γ, is the fraction of the total pipeline delay averaged over all hazards and contains all of the information about the hazards in a particular piece of code as well as details of the microarchitecture of the processor. In practice γ is difficult to calculate, but like β_h, it is constrained to have a value between 0 and 1.

The final expression for the total time to process a program is simply given by adding Eqs. 4 and 7, the busy and not busy times. We then form our performance measure by dividing the total time by the number of instructions, N_I. The final result is then:

$$T/N_I = (\frac{t_o}{\alpha} + \frac{\gamma N_H}{N_I} t_p) + \frac{t_p}{\alpha p} + \frac{\gamma N_H t_o}{N_I} p. \qquad (8)$$

The first term in this expression is independent of the pipeline depth; the second term varies inversely with p; and the last term varies linearly with p. This result depends on numerous parameters. The ratio N_H/N_I is mainly dependent on the workload being executed, but is also dependent on the microarchitecture through, for instance, the branch prediction accuracy. α and γ are mainly microarchitecture dependent, but also depend on the workload. t_o is technology dependent, while t_p is dependent on both technology and the microarchitecture.

Eq. 8 has a minimum at a particular value of p, the pipeline depth. One can solve for this optimum performance point by differentiating Eq. 8 with respect to p, setting the resulting expression equal to 0, and solving for p_{opt}. The result is

$$p_{opt}^2 = \frac{N_I t_p}{\alpha \gamma N_H t_o}. \qquad (9)$$

Some general observations about the optimum pipeline depth can be made from Eq. 9. The optimum pipeline depth increases for workloads with few hazards. As technology reduces the latch overhead, t_o, relative to the total logic path, t_p, the optimum pipeline depth increases. As the degree of superscalar processing , α, increases, the optimum pipeline depth decreases. And lastly, as the fraction of the pipeline that hazards stall, γ, decreases, the optimum pipeline depth increases. Intuitive explanations for these dependencies will be discussed in more detail in a later section.

The reader will note that in the derivation of this theory, no mention was made as to whether the instruction processing was in-order or out-of-order. That is because out-or-order processing does not change the analysis. However, it can effect the values of some of the parameters. In particular, the degree of superscalar processing will be altered with out-or-order processing. Also, the effective penalty incurred from various hazards will be reduced by allowing out-of-order processing of instructions to fill in some of the stall cycles.

It should be noted that an equivalent result was obtained by Emma and Davidson [3]. They confined their analysis to a one-at-a-time, in-order processor model. Most of their paper concerns the first principles determination of the parameter γ. The contributions of each type of hazard and the relative numbers of each type were estimated for a particular microarchitecture. We will not repeat that type of analysis here, but rather test various aspects of our theory with detailed modeling of a particular microarchitecture.

3. Simulation Methodology

In order to test the limits of our optimum pipeline depth theory and to explore additional ramifications of the theory, we have used a proprietary simulator. The simulator uses as input design parameters that describe the organization of the processor and a trace tape. It produces a very flexible cycle accurate model of the processor. With this tool we are able to model numerous pipeline designs, variable issue width superscalar designs, and either in-order or out-of-order execution processing. We also had the availability of 35 traces, encompassing traditional (legacy) workloads, "modern" workloads, SPEC95 and SPEC2000 workloads. The traditional workloads include both database and on-line transaction processing (OLTP) applications. These applications were originally written in S/360 assembler language over 30 years ago. They have continued to evolve and now run on S/390 (zSeries) processors. The modern workloads were written in either C++ or Java. These traces were carefully selected because they accurately reflect the instruction mix, module mix and branch prediction characteristics of the entire application, from which they were derived. This tool has mainly been used for work on S/390 processors.

In order to build a model to test the above theory we need to be able to expand the processor pipeline in a uniform manner. The theory assumes that the processor logic can be uniformly divided into p stages. In modeling, it is not practical to repartition the pipeline for each new design. Instead we have made use the pipeline shown in Fig. 4.

This is the pipeline model of a 4 issue superscalar, out-or-order execution machine. The model can handle S/390 code, so that register only instructions (RR), as well as register / memory access instructions (RX), must be executed efficiently. The pipeline has 2 major instruction flow paths. The RR instructions go sequentially through Decode - Register Rename - Execution Queue - Pipelined E-Unit - Completion - Retire. The RX instructions, including loads and stores, add to this pipeline the sequence: Address Queue - Pipelined Address Generation - Cache Access, between the register rename and execution queue stages.

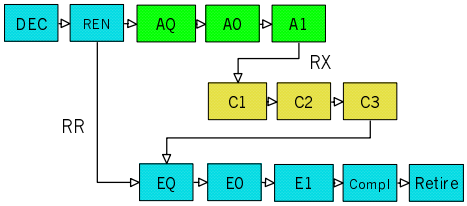

Fig. 4 shows the pipeline modeled in this study. The stages include: Decode, Rename, Agen Q, Agen, Cache Access, Execute Q, Execute, Completion and Retire.

This is the base pipeline we modeled. Address generation, cache access and execution are independently out-of-order processes. In testing the theory we utilize the flexibility of our simulator model. In particular we expand the pipeline depth by adding stages "uniformly". We insert extra stages in Decode, Cache Access and E-Unit pipe, simultaneously. For example, as the Cache Access pipe is increased (in Fig. 4) the Decode and E-Unit pipe stages are increased proportionally. This allows all hazards to see pipeline increases. Hazards, whose stalls cover a larger fraction of the pipeline, see larger increases due to the increased pipeline depth.

We have also utilized other aspects of our model flexibility in our studies. In one experiment the branch prediction accuracy was varied in order to alter the number of hazards encountered for a particular workload. In still another experiment we varied the superscalar issue width, to study the predicted dependence of the optimum pipeline depth on the average degree of superscalar processing. We have also run both in-order and out-of-order models to show that both show qualitatively the same results.

4. Simulator Results

Each particular workload was simulated in this way with pipeline depths ranging from 10 stages to 34 stages. We count the stages between decode and execution, rather than the entire pipeline. We determine the entire logic delay time in the same way. To compare with the theory, we use the detailed statistics obtained from each simulator run to determine the parameters in Eq. 8. Two of the parameters, N_I and N_H, are simply enumerated, but α and γ require more extensive analysis of the details of the pipeline and the particular distribution of instructions and hazards in each simulation run. The parameters, t_p and t_o, were chosen from a particular technology; but only the ratio $t_p/t_o = 55$ is important in determining the optimum pipeline depth.

Fig. 5 shows the results of the simulation, for a particular traditional workload, along with the corresponding theoretical curve. We plot TPI (Time per Instruction) as a function of the number of pipeline stages. The predicted

minimum in the curve is clearly seen. It is fairly flat around the minimum, which leaves considerable latitude in choosing the best design point for a particular processor. The fact that the optimum pipeline depth is so large is due mainly to the small latch delay in our technology model.

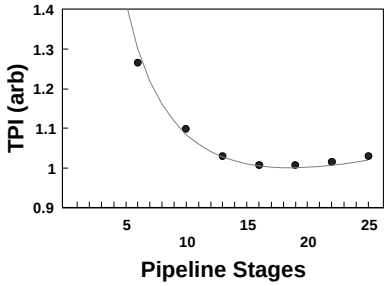

Fig. 5 is a direct comparison of the simulation (data points) and the theory (solid curve) for a particular workload. The TPI numbers are scaled to the minimum point.

As can be seen the agreement between theory and experiment is extraordinarily good. Because of the difficulty of determining the parameter, γ even from our detailed statistics, the fit between theory and experiment should be considered as a one parameter fit. We effectively chose a value of γ, consistent with our estimates from the statistics, which caused the theory and experiment to correspond at one point. This in no way diminishes the value of the comparison, which clearly shows that the theory adequately describes the functioning of the pipeline. This comparison could have been shown for any of the 35 workloads run, with equally good results, although some workloads show somewhat more scatter in the data.

Eq. 9 gives the minimum point in Fig. 5, and from the fit of theory and simulation, it clearly agrees with the simulation. To further explore the applicability of the theory one can construct experiments to test the various dependencies in Eq. 9. The first one we will test is the dependence on α, the degree of superscalar processing. In order to do this we first obtain the results for a particular workload (that has significant superscalar parallelism), and then obtain new results limiting the issue width to 1, i.e.. non-superscalar but still out-of-order processing. In this way the primary change in parameters will be to reduce α. Some secondary effect on γ may result, but will be small.

Fig. 6 shows these 2 simulation results for the IJPEG workload from the SPEC95 suite. The shift in the optimum pipeline depth point with decreased superscalar issue width is clearly evident. As expected, moving to a single issue E-Unit, decreases α, and shifts the optimum point to larger values of pipeline stages. Even the magnitude of the shift can be accurately accounted for by the theory. In passing we should note that reducing the issue width, reduces the overall performance, as well.

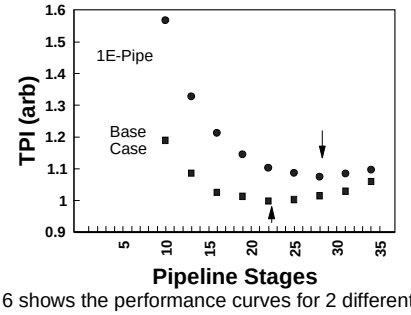

Fig. 6 shows the performance curves for 2 different superscalar issue widths: single and 4-issue for the IJPEG workload. The optimum performance positions are indicated by arrows.

We can test the dependence of Eq. 9 on γ by varying the branch prediction efficiency by reducing the size of the branch prediction table. If we reduce the branch prediction capability, we both increase the number of branch hazards contained in N_H; and since conditional branches cause pipeline stalls over the entire pipeline, we increase the value of γ. Both of these effects should shift the optimum pipeline point to lower values of p. Fig. 7 shows this effect for a particular modern workload. Also note that reducing branch prediction efficiency severely impacts the performance.

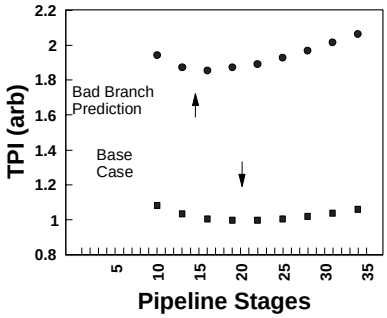

Fig. 7 shows the effect of changing branch prediction on the optimal pipeline depth.

5. Workload Specificity

No one doubts that there are significant differences between different types of workloads run on processors. Maynard, et. al. [6] have explored the workload differences between multi-user commercial workloads and technical workloads. Differences arise from branch prediction accuracy, the degree of operating system calls, I/O content and dispatch characteristics. Gee et.al. [7] and Charney et. al. [8] have addressed the cache performance of the SPEC

workloads compared to other workloads. The SPEC workloads tend to have small instruction footprints, and can achieve "good" performance, even for small caches. In this work we have employed both SPEC benchmark traces and workloads with much larger cache footprints. We have used both traditional S/390 legacy workloads and some modern workloads, written in Java and C++.

We have simulated all 35 of our workloads, obtained the TPI as a function of pipeline depth for each of them, and determined the optimum pipeline depth for each. In Fig. 8 we show the distribution of these optimum pipeline depths for different workloads. As one can see there is essentially a Gaussian distribution of the optimum pipeline depth over the workloads studied.

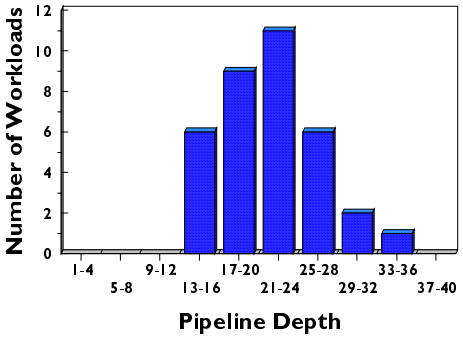

Fig. 8 shows the distribution of pipeline depth optima for all workloads.

It is perhaps more instructive to divide the distribution into the separate types of workloads. Fig. 9 shows this analysis, divided into SPEC workloads (programmed in C), traditional workloads (programmed largely in assembler), and "modern" workloads (programmed in C++ or Java). It is clear that there is a significant difference between the SPEC workloads and the other workloads in terms of the optimal pipeline depth. A pipeline optimized for real workloads, either traditional or modern, should be considerably deeper than a pipeline optimized for the SPEC workloads. This is true for both SPEC95 and SPEC2000

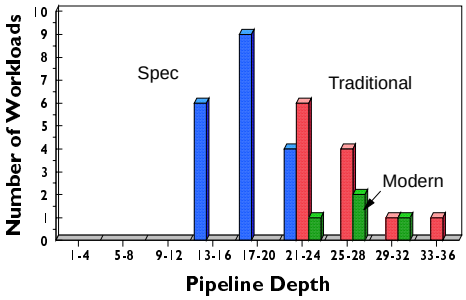

Fig. 9 shows the distribution of pipeline depth optima for different workload classes. Blue are the Spec (C) workloads, red are the traditional workloads, and green are the modern (C++ and Java) workloads.

workloads. Fortuitously, as we have seen, the dependence of the performance on the number of pipeline stages, leading to the optimum pipeline depth, is quite broad. Optimizing for the wrong type of workload will not incur too large a penalty.

6. Discussion

Now let us explore the consequences of this work in a more intuitive way. Looking back at Eq. 8, we see that this expression for the performance consists of four terms. The first two have no dependence on the pipeline depth. The first term is the constant performance degradation due to the latch overhead. The second term is an overhead from pipeline hazards. The next two terms are more interesting.

The third term is inversely proportional to the pipeline depth. This is the term that drives deeper pipelines to have better performance. Basically, what it says is that the total logic delay of the processor is divided up into separate parts for each pipeline stage. One then gets a larger throughput by "storing" instructions in the pipeline. The performance is given by the rate at which instructions finish the pipeline, not the time it takes an instruction in the pipeline. In fact if there were no hazards, as in Eq. 4, the only performance limitation would arise from the latch overhead.

The last term in Eq. 8 is linearly proportional to the pipeline depth. This term arises from the dependence of pipeline stalls on pipeline depth. Deeper pipelines suffer a larger penalty from hazards. The term is proportional to the fraction of the pipeline that each stall covers, as well as the number of hazards. It is also important to note that the term is proportional to the latch overhead; each additional hazard induced pipeline stall causes a performance degradation from that latch overhead. On the other hand the hazard related term which is proportional to the total logic depth, the second term, is independent of both pipeline depth and latch overhead.

There is a competition between greater pipeline throughput and pipeline hazards. If no hazards were present, the optimum performance would occur for an infinitely deep pipeline. One simply loads all of the instructions into the pipeline, and the performance, the rate at which the instructions complete, is optimized. On the other hand, hazards disrupt the pipeline flow by periodically draining the pipeline or a portion of the pipeline. This competition between increasing pipeline throughput, by "storing" instructions in the pipeline, and reducing pipeline hazard stalls, by minimizing pipeline depth, accounts for the observed behavior.

It is instructive to compute the portion of the cycle time allocated to latch overhead for the optimum pipeline depths. For the ratio of total logical delay of the processor to latch overhead, $t_p/t_o = 55$, latch overhead consumes 25% to

50% of the cycle time. Even though this may seem like a large portion of the cycle time, our results clearly show that this gives the optimal pipeline performance. It would be a difficult challenge to actually design a processor, which is so highly pipelined. Designing a processor, where latch overhead can account for ¼ to ½ of the time allocated to each cycle, offers significant new challenges in partitioning the logic and controlling the power.

Moving on to Eq. 9 we can understand the optimum pipeline depth dependencies in light of the above arguments. As the number of hazards per instruction increases, the optimum pipeline depth is pushed to lower values. As the fraction of the pipeline stalled by any one hazard increases, the optimum pipeline depth decreases. As the superscalar processing parallelism increases, the throughput increases without the need for a pipeline depth increase, and therefore the optimum pipeline depth decreases. As the latch overhead increases relative to the total logic delay, pipelining becomes more of a burden, and the optimum pipeline depth decreases.

The competing nature of these effects is inherent in Fig. 9. One might expect the SPEC workloads to have a longer optimal pipeline depth than the traditional legacy workloads, due to the fact that the SPEC applications have fewer pipeline stalls than the legacy applications. However, the opposing effect that the SPEC applications contain a higher degree of instruction level parallelism, that can be exploited by a superscalar processor, is even more important. The net result is that the SPEC workloads optimize for a shorter pipeline.

In all of the preceding, we have been implicitly considering an infinite cache model. We could have constructed a similar variable pipeline depth model, which included the multiple level cache access paths. As the pipeline depth increased, the latency of cache or memory accesses would have shown an increased number of cycles. However, the cache or memory access time would remain constant. Therefore, although the finite cache contribution to the CPI would increase, the finite cache contribution to TPI stays constant. The finite cache equivalent to Figs. 5-7 simply have a constant finite cache TPI adder. This in no way changes the optimum pipeline depth or any of the dependencies that have been discussed.

7. Summary

A theory has been presented of the optimum pipeline depth for a microprocessor. The theory has been tested by simulating a variable depth pipeline model, and the two are found to be in excellent agreement. It is found that the competition between "storing" instructions in a deeper pipeline to increase throughput, and limiting the number of pipeline stalls from various pipeline hazards, results in an optimum pipeline depth. That depth depends in a complex but understandable way on the detailed microarchitecture of the processor, details of the underlying technology used to build the processor, and certain characteristics of the workloads run on the processor. Our analysis clearly shows a difference between the optimum pipeline length for SPEC applications and the optimum pipeline length for both legacy applications and modern workloads. It would be instructive to gain a deeper understanding of the factors that produce similar optimal pipeline depths for the modern workloads and legacy applications.

8. Acknowledgments

The authors would like to thank P. G. Emma and E. S. Davidson for many stimulating discussions and helpful comments on this work.

9. References

[1] S. R. Kunkel and J. E. Smith. "Optimal pipelining in supercomputers", *Proc. of the 13th Annual International Symposium on Computer Architectures*, pp. 404 - 411, 1986.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler and D. Burger. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures", *Proc. of the 27th Annual International Symposium on Computer Architectures*, pp. 248 - 259, 2000.

[3] P. G. Emma and E. S. Davidson. "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance", *IEEE Transactions on Computers* **C-36**, pp. 859 - 875, 1987.

[4] M. H. Macdougal. "Instruction-Level Program and Processor Modeling", *Computer*, pp. 14 - 24, 1984.

[5] P. Emma, J Knight, J Pomerene, T Puzak, R Rechschaffen. "Simulation and Analysis of a Pipeline Processor", *8th Winter Simulation Conference*, pp. 1047 - 1057, 1989.

[6] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. "Contrasting Characteristics and cache performance of technical and multi-user commercial workloads", *ASPLOS VI*, pp. 145 - 156, 1994.

[7] J. D. Gee, M. D. Hill, D. N Pnevmatikatos, and A. J. Smith. "Cache Performance of the SPEC Benchmark Suite", *Technical Report 1049, Computer Sciences Department, University of Wisconsin*, 1991.

[8] M. J. Charney and T. R. Puzak. "Prefetching and Memory System Behavior of the SPEC95 benchmark Suite" *IBM Journal of Research and Development* **41**, pp. 265 - 286, 1997.