Re-Visiting the Performance Impact of Microarchitectural Floorplanning

Anupam Chakravorty, Abhishek Ranjan, Rajeev Balasubramonian
School of Computing, University of Utah

Abstract

The placement of microarchitectural blocks on a die can significantly impact operating temperature. A floorplan that is optimized for low temperature can negatively impact performance by introducing wire delays between critical pipeline stages. In this paper, we identify subsets of wire delays that can and cannot be tolerated. These subsets are different from those identified by prior work. This paper also makes the case that floorplanning algorithms must consider the impact of floorplans on bypassing complexity and instruction replay mechanisms.

Keywords: *microprocessor operating temperature, microarchitectural floorplanning, critical loops in pipelines.*

1. Introduction

High transistor and power densities have resulted in thermal issues emerging as a major design constraint for modern-day microprocessors. Each new microprocessor generation requires greater design effort to ensure that high performance can be delivered while maintaining acceptable operating temperatures. The problem is only exacerbated by the recent introduction of vertically stacked 3D chips [13, 16]. While 3D chips help reduce on-chip communication latencies and power dissipation, they increase power density levels and can cause operating temperature to increase by tens of degrees [12].

Heat produced by a microarchitectural structure spreads laterally and vertically. Heat removal from a high temperature unit can be accelerated by surrounding it with low temperature units. Therefore, it has been suggested by various research groups [4, 5, 6, 7, 9, 10, 14] that smart microarchitectural floorplans can lower on-chip peak temperatures and trigger fewer thermal emergencies. We expect that floorplanning will continue to receive much attention in future years, especially in the realm of 3D chips.

A thermal-aware floorplan can negatively impact performance. If the inputs to a microarchitectural unit are produced by a distant unit, long wire delays are introduced. A recent paper by Sankaranarayanan *et al.* [14] demonstrates that some of these wire delays can have a significant impact on overall performance. Therefore, they argue that floorplanning algorithms must strive to co-locate certain units, thereby yielding designs with sub-optimal thermal characteristics.

In this paper, we re-visit the performance analysis of inter-unit wire delays. We show that smart pipelining can alleviate the performance impact of many long wire delays. We identify a subset of wire delays (different from that identified by Sankaranarayanan *et al.* [14]) that can degrade performance the most. The lengthening of wire delays also has a salient impact on register bypassing complexity and instruction replay complexity. These issues must also serve as inputs to any microarchitectural floorplanning tool.

Section 2 describes our simulation methodology. Section 3 describes pipeline implementations that can tolerate wire delays between pipeline stages. Section 3 also provides simulation results and conclusions are drawn in Section 4.

2. Methodology

Our performance simulator is based on an extended version of Simplescalar-3.0 [2] for the Alpha AXP ISA. The simulator models separate issue queues, register files, and reorder buffer (ROB) instead of a unified Register Update Unit (RUU). Contention for memory hierarchy resources (ports, banks, buffers, etc.) are modeled in detail. The register file and issue queue are partitioned into integer and floating-point clusters. While Simplescalar's default pipeline only has five stages, we model the effect of a deeper pipeline on branch mispredict penalty and register occupancy. Processor parameters are listed in Table 1. As a benchmark set, we use the 23 SPEC2k programs compatible with our simulator. Each program is executed for a 100 million instruction window identified by the Simpoint toolkit [15]. Detailed simulation was carried out for one million instructions to warm up various processor structures before taking measurements.

Fetch queue size	16	Branch predictor	comb. of bimodal and 2-level
Bimodal predictor size	16K	Level 1 predictor	16K entries, history 12
Level 2 predictor	16K entries	BTB size	16K sets, 2-way
Branch mispredict penalty	at least 10 cycles	Fetch, Dispatch, Commit width	4
Issue queue size	20 Int, 15 FP	Register fi le size	80 (Int and FP, each)
Integer ALUs/mult-div	4/2	FP ALUs/mult-div	2/1
L1 I-cache	32KB 2-way	Memory latency	300 cycles for the fi rst block
L1 D-cache	32KB 2-way 2-cycle	L2 unifi ed cache	2MB 8-way, 30 cycles
ROB/LSQ size	80/40	I and D TLB	128 entries, 8KB page size

Table 1. Simplescalar simulator parameters.

3. Pipelining Wire Delays

3.1. Pipelining Basics

Deep pipelines can impact IPC in many ways. Firstly, they can improve IPC because more (independent) instructions can be simultaneously operated upon in a cycle. The effect of deep pipelining on dependent instructions depends on the nature of the dependence. Consider the following two examples.

Figure 1 shows the behavior of a branch mis-predict in a shallow and deep pipeline. In either case, a certain number of gate delays (say, N FO4 gate delays) must be navigated to determine the outcome of the branch. In the shallow pipeline, it takes four pipeline stages to execute the branch, so the penalty for a branch mis-predict equals $N + 4V$, where V equals the latch and skew overhead per pipeline stage. In the deep pipeline, it takes eight pipeline stages to execute the branch, so the penalty for a branch mis-predict equals $N + 8V$. In other words, the deep pipeline has increased the gap between two control-dependent instructions.

Figure 2 shows the behavior of a 64-bit integer add instruction in shallow and deep pipelines. In the shallow pipeline (Figure 2(a)), the add is completed in a single pipeline stage and the gap between two dependent add operations is $M + V$, where M represents the FO4 gate delays required to complete the 64-bit add. In the deep pipelines, the add operation is broken into 4 pipeline stages. If the second dependent instruction can begin execution only after the first instruction completes (shown in Figure 2(b)), the gap between the instructions is $M + 4V$. However, the pipeline may be such that the first stage operates on the 16 least-significant bits, the second stage on the next 16 least-significant bits, and so on. In such a pipeline (Figure 2(c)), the second dependent instruction can start operating on its 16 least-significant bits as soon as the first instruction leaves the first stage. A similar pipeline organization has been implemented in Intel's Netburst microarchitecture [8]. With this pipeline, the gap between dependent instructions is $M/4 + V$. Therefore, a deep pipeline does not necessarily lengthen the gap between dependent instructions. The pipeline implementation and the nature of the dependence determine the performance effect of adding stages to a pipeline. In other words, a microarchitectural loop is formed when the output of one pipeline stage is required as an input to an earlier stage [1]. If the introduction of additional pipeline stages does not lengthen critical microarchitectural loops, IPC will not be significantly degraded.

The microarchitectural units within a typical out-of-order superscalar processor are shown in Figure 3(a). Each of these units can represent a pipeline stage. In order to achieve high clock speeds, each stage can itself be partitioned into multiple pipeline stages. During the process of microarchitectural floorplanning, some of the units may be placed far apart. The wire delay for communication between the units may be long enough that multiple pipeline stages have to be introduced between the units just for data transfer. Figure 3(b) shows an example where three new pipeline stages are introduced between the branch predictor and I-cache, and between the integer issue queue and the integer execution units. Sankaranarayanan *et al.* [14] claim that separating these units has the most significant impact on performance. They report that for these sets of units, the introduction of four additional pipeline stages results in an execution time increase of 50% and 65%. Next, we show that the pipeline can be designed such that these wire delays can be tolerated.

3.2. Back-End Pipelines

Consider the pipeline shown in Figure 4. The wakeup and select operation happens within the issue queue in a single pipeline stage. At the end of the stage, it is known that instruction A is ready to execute. Control information is sent to the ALU through long wires that constitute four pipeline stages (in this example). In the meantime, the register tags for the input operands are sent to the register file, the values are read in a single pipeline stage, and these values then arrive at the ALU. The ALU can begin its computation once the control information and the register operands arrive at the ALU. If the wakeup and select operation completes at the end of cycle t, the computation begins in cycle $t + D$, where delay D is determined by the length of the critical path (either for the register access or for the control

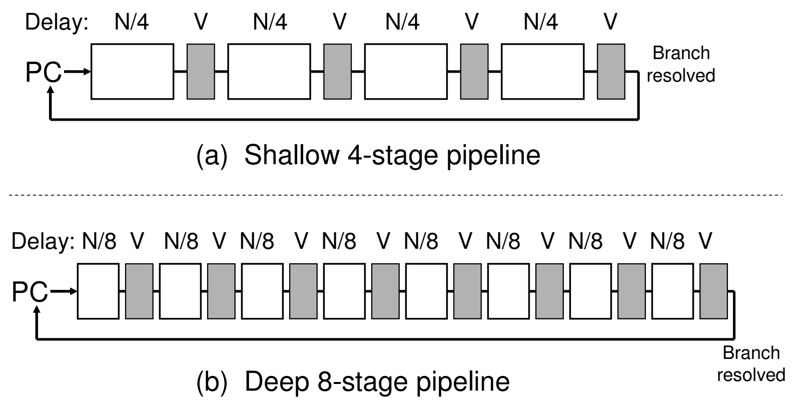

Delay: N/4 V N/4 V N/4 V N/4 V
PC
Branch resolved
(a) Shallow 4-stage pipeline
Delay: N/8 V N/8 V N/8 V N/8 V N/8 V N/8 V N/8 V N/8 V
PC
Branch resolved
(b) Deep 8-stage pipeline

Figure 1. **Branch mispredict loop in a (a) shallow and (b) deep pipeline. Grey boxes represent pipeline latches.**

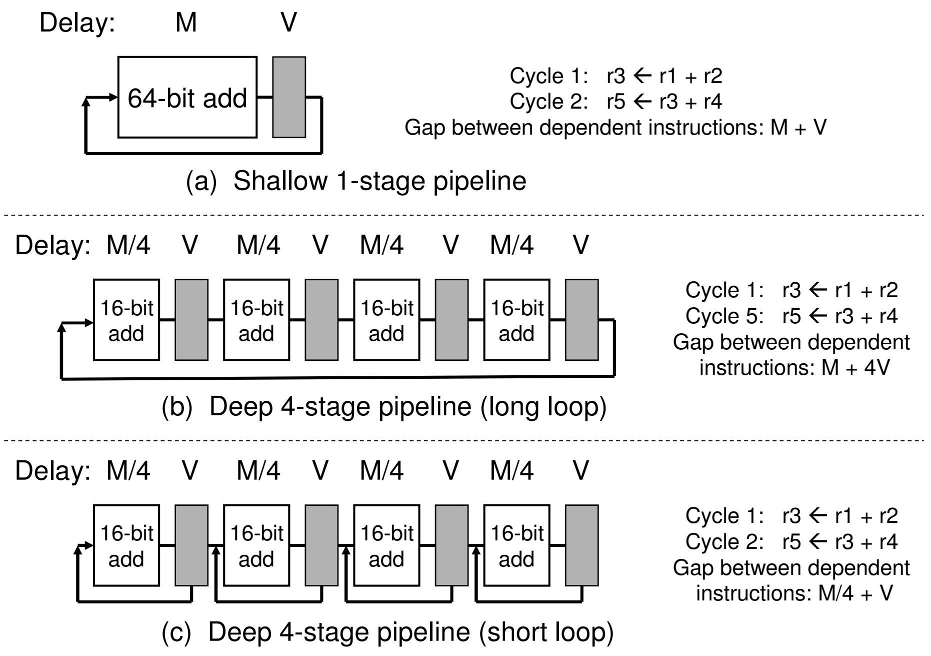

Delay: M V
64-bit add
Cycle 1: r3 ← r1 + r2
Cycle 2: r5 ← r3 + r4
Gap between dependent instructions: M + V
(a) Shallow 1-stage pipeline
Delay: M/4 V M/4 V M/4 V M/4 V
16-bit add
Cycle 1: r3 ← r1 + r2
Cycle 5: r5 ← r3 + r4
Gap between dependent instructions: M + 4V
(b) Deep 4-stage pipeline (long loop)
Delay: M/4 V M/4 V M/4 V M/4 V
16-bit add
Cycle 1: r3 ← r1 + r2
Cycle 2: r5 ← r3 + r4
Gap between dependent instructions: M/4 + V
(c) Deep 4-stage pipeline (short loop)

Figure 2. **The ALU-bypass loop in shallow and deep pipelines.**

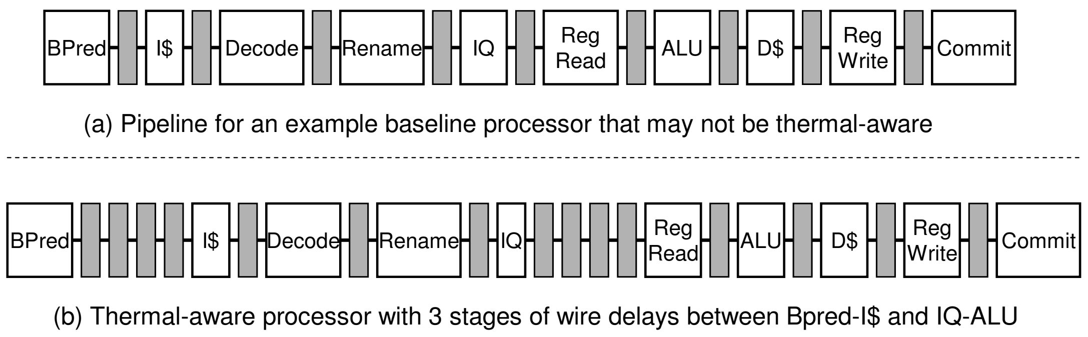
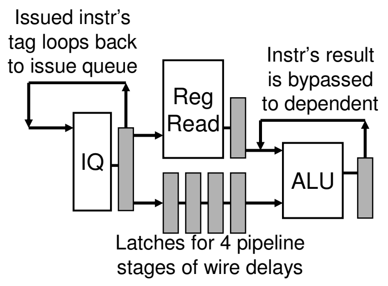

BPred
I$
Decode
Rename
IQ
Reg Read
ALU
D$
Reg Write
Commit
(a) Pipeline for an example baseline processor that may not be thermal-aware
BPred
I$
Decode
Rename
IQ
Reg Read
ALU
D$
Reg Write
Commit
(b) Thermal-aware processor with 3 stages of wire delays between Bpred-I$ and IQ-ALU

Figure 3. **Pipelines for a baseline processor and for a thermal-aware processor that places BPred and ICache far apart, and IQ and ALU far apart.**


Issued instr's tag loops back to issue queue
Reg Read
Instr's result is bypassed to dependent
IQ
ALU
Latches for 4 pipeline stages of wire delays

Instr A: r3 ← r1 + r2
Instr B: r5 ← r3 + r4

Cycle at which each instr leaves each pipeline stage:

	A	B
IQ	t	t+1
ALU	t+5	t+6

Figure 4. Tight loops within a pipeline that places IQ and ALU far apart.

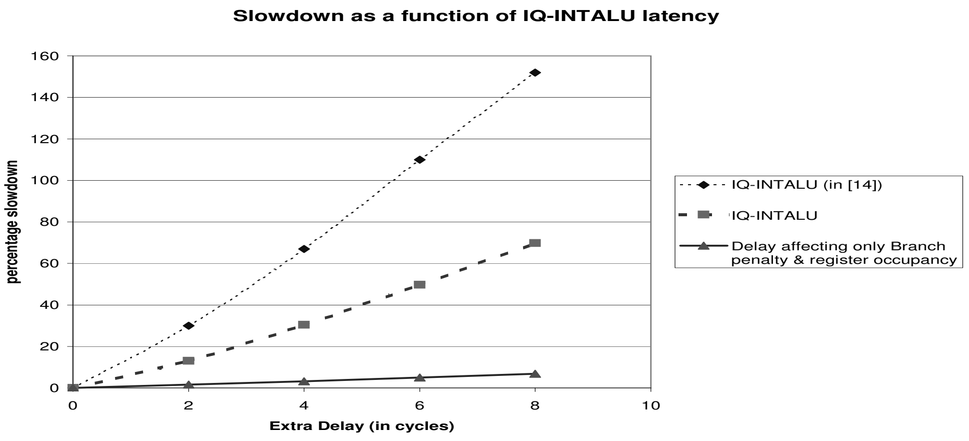


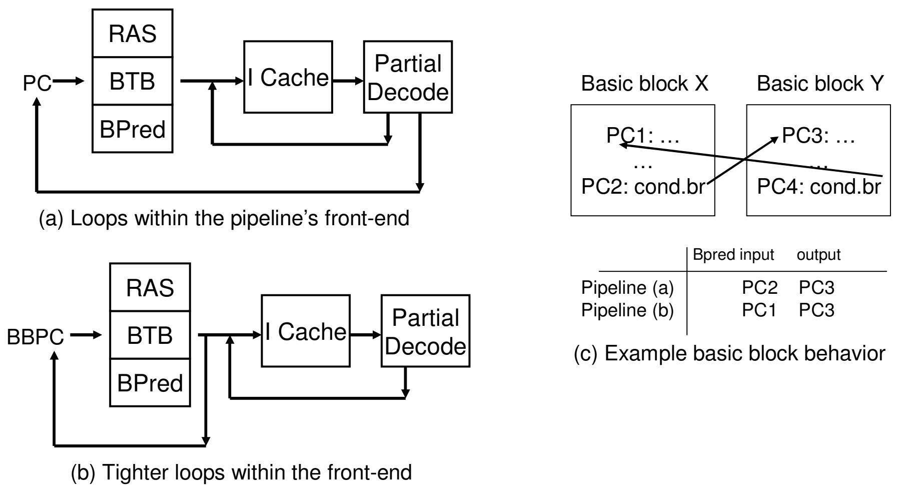
Figure 5. Percentage slowdown as a function of IQ - IntALU wire delay for various pipelines.

information). In a compact floorplan with a single-cycle register file access, D can be as small as 2. In the floorplan in Figure 4, where the wire delays for the control information are on the critical path, D is 5.

However, the gap between read-after-write (RAW) dependent instructions is not a function of D. The wakeup and select operation completes in cycle t and it is known that instruction A will produce its output at the end of cycle $t + D$ (assuming that the ALU operation completes in a single cycle). Accordingly, in cycle $t + 1$, the output register for instruction A will be broadcast to the issue queue and at the end of cycle $t + 1$, wakeup and select will produce the dependent instruction B of instruction A. Instruction B's control information and register values arrive at the ALU at the end of cycle $t + D$. Instruction B begins its execution in cycle $t + D + 1$, replacing one of its register operands with the result of A that is read off of the bypass bus (note that the result of A is ready by the end of cycle $t + D$). Thus, regardless of the value of D, the gap between dependent instructions remains a single cycle (max(latency for the ALU+bypass operation, latency for wakeup+select)). In essence, the gap is determined by the maximum length of a dependent loop. There is one loop within the issue queue, where the result of wakeup+select feeds back into the wakeup+select for the next cycle. There is a second loop within the ALU and bypass bus, where the output of an ALU operation is bypassed back to the inputs of the ALU for use in the next cycle. However, neither of these loops involve the delay D. The delay D will influence the loop that re-directs fetch on discovering a branch mis-predict. It also affects the loop that releases registers on commit and makes these registers available to the dispatch stage. There are other effects on less critical loops, such as the loop for branch predictor update.

In Figure 5, we show the impact of delay D on overall average IPC for SPEC2k programs. The thin dashed line in Figure 5 reproduces the result in [14] and depicts the IPC slowdown as a function of the wire delay D between the integer issue queue and the integer execution units. The bold dashed line represents our simulation where the gap between dependent instructions is varied as a function of D. Finally, the solid line in Figure 5 represents the simulation where D only impacts the branch mis-predict penalty, the delay in releasing registers, and not the gap between RAW-dependent instructions. We see that the wire delay D has a negligible impact on overall performance, unlike the conclusion drawn in [14].

The above analysis does not take two important effects into account. Firstly, as D increases, the likelihood of reading an invalid operand from the register file increases. Hence, more bypass registers are required at the ALU to buffer the results of instructions executed in the last R cycles. R is a function of register file read and write latency and the wire delays between the ALU and register file. Cor-



(a) Loops within the pipeline's front-end

(b) Tighter loops within the front-end

(c) Example basic block behavior

Figure 6. Loops within front-end pipelines.

respondingly, the complexity of the multiplexor before the ALU also increases [11]. It is possible that an increase in D (and correspondingly in R) may result in an increase in bypass delay, thereby introducing stall cycles between RAW-dependent instructions. It is unlikely that a processor will be designed to allow such stalls between every pair of RAW-dependent instructions. We expect that the following design methodology is more realistic: circuit complexity analysis will yield the maximum number of bypass registers that can be accommodated in a cycle; correspondingly, the maximum allowed delay D is fed as a constraint to the floorplanning algorithm. Palacharla *et al.* [11] quantify bypassing complexity as a function of issue width and window size. A similar analysis can help quantify bypassing complexity as a function of bypass registers and this is left as future work.

The second important effect of increasing D is the cost of instruction replay mechanisms. Processors such as the Pentium4 [8] implement load-hit speculation with selective replay within the issue queue. When an instruction leaves the issue queue, based on the expected latency of the instruction, a subsequent wakeup operation is scheduled. A load is a variable-latency instruction and a wakeup is scheduled, assuming that the load will hit in the cache with no bank conflicts. As soon as it is known that the load latency does not match the best-case latency, dependent instructions that have already left the issue queue are squashed. These instructions will subsequently be re-issued after the load latency is known. Also, dependents of loads are kept in the issue queue until the load latency is known. This simplifies the recovery from the load-hit mis-speculation and facilitates replay. Thus, load-hit speculation can negatively impact performance in two ways: (i) replayed instructions contend twice for resources, (ii) issue queue occupancy increases, thereby supporting a smaller in-flight window, on average. A deeper pipeline can exacerbate this performance impact because it takes longer to detect if the load is a hit or a miss. Borch *et al.* [1] conduct an experiment (Figure 5 in [1]) to determine the impact of a deep pipeline on load-hit speculation. While keeping the branch mispredict penalty a constant, they show that reducing six pipeline stages from the load-hit speculation loop causes speedups ranging from 1.02 to 1.15, with an average of 1.07. Therefore, the addition of pipeline stages between the issue queue and the execution units will impact performance, but not as drastically as that indicated by [14].

3.3. Front-End Pipelines

The results in [14] show that wire delays between the branch predictor and I-cache have the second largest impact on performance. The branch/jump/return PC serves as an input to the branch predictor, branch target buffer (BTB), and return address stack (RAS). Correspondingly, the next target address is determined. This target address is then sent to the I-Cache to fetch the next line. After partial decoding, it is known if the line contains a branch/jump/return instruction. If the next line does not contain a control instruction, the next sequential cache line is fetched from the I-Cache. When a branch/jump/return instruction is encountered, the PC is sent to the branch predictor to determine the next fetch address. In the pipeline just described (and depicted in Figure 6(a)), there are two loops. The longer loop determines the next input to the branch predictor/BTB/RAS after fetching and decoding the instruction. If wire delays are introduced between the branch predictor/BTB/RAS and the I-Cache, this loop becomes longer and the rate of instruction fetch is severely crippled. The thin dashed line in Figure 7 reproduces the results in [14] and indicates the performance degradation effect of introducing wire delays between the branch predictor and I-Cache. The bold dashed line in Figure 7 attempts to model a similar effect in our

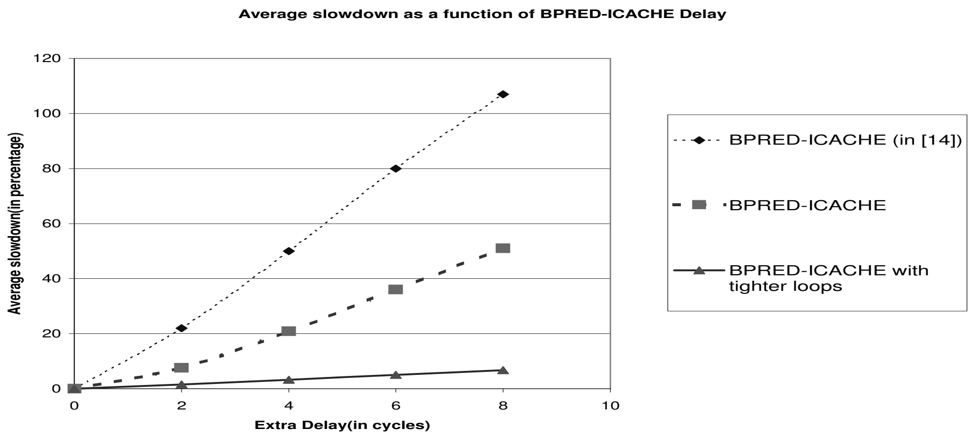


Figure 7. **Percentage slowdown for BPred - ICache wire delays.**

simulator. Given a loop length of D cycles, our simulator stalls fetch for D cycles every time a control instruction is encountered.

Fortunately, the branch predictor and I-Cache can be easily decoupled. Instead of identifying a branch by its PC, the branch can be identified by the PC of the instruction that starts the basic block. As shown in Figure 6(c), $PC1$ is the start of basic block X and $PC3$ is the start of basic block Y. $PC2$ is the branch that terminates basic block X and $PC4$ is the branch that terminates basic block Y. In the pipeline described before, $PC2$ is used to look up the branch predictor and the output is $PC3$. A few cycles later, it is determined that $PC4$ is the next branch and this is used as input to the branch predictor to determine that the next instruction is $PC1$, and so on. In the proposed pipeline (shown in Figure 6(b)), $PC1$ is used to index into the branch predictor and the output is $PC3$. This is fed to the I-Cache, where it is queued. The I-Cache adopts this PC as soon as it encounters the next control instruction. In the meantime, $PC3$ is fed to the branch predictor and the output is $PC1$. In essence, when a PC is input to the branch predictor, the predicted address represents control information for the branch that terminates the basic block for that PC.

Clearly, the same policy must be adopted during update. Once the outcome of $PC2$ is known, $PC1$ is used as the index to update the branch predictor state. When $PC2$ is encountered during fetch, the index of the previous branch prediction ($PC1$) can be saved in its reorder buffer (ROB) entry to facilitate such an update.

If a basic block has multiple entry points, note that multiple branch predictor entries are created for the branch that terminates the basic block. This may even improve the quality of the branch predictor. As shown in Figure 6(b), the advantage of the above approach is that the microarchitectural loop involving the branch predictor does not include the I-Cache, decode, or the wire delays to reach the I-Cache. For the pipelines in Figure 6 (a) and (b), there still remains the loop (not shown in the figure) where the branch predictor and fetch are re-directed because of a branch/target mis-predict. The solid line in Figure 7 shows the performance impact of the above approach. The increase in wire delay between branch predictor and I-Cache only increases the cost of a branch mis-predict and not the ability of the branch predictor to produce a new target every cycle. The change in the branch predictor algorithm (indexing with basic block start address) has a minimal impact on branch predictor accuracy. For nearly all applications, the branch direction and target accuracy are effected by much less than a single percentage point. In *eon*, the branch direction prediction accuracy actually improved from 92% to 99% and in *mcf*, the accuracy improved from 92% to 96%. The new branch predictor algorithm improves IPC by 1% on average, with almost all of this improvement being attributed to a 19% increase in the IPC for *eon*. As wire delay between branch predictor and I-Cache increases, the IPC degradation for this new pipeline is marginal.

The above result shows that by decoupling branch predictor look-up from the I-Cache look-up, wire delays between the two stages can be tolerated, allowing greater flexibility when floorplanning. The proposed pipeline is not the only possible alternative. The I-Cache can implement next line and set (NLS) prediction [3] and the branch predictor can be used for more accurate prediction and can over-ride the NLS prediction in case of a disagreement. In the common case where the NLS prediction agrees with the branch prediction, the wire delay between I-Cache and branch predictor is not part of any critical loop. Such a decoupled pipeline will also be tolerant of I-Cache to branch predictor wire delays.

3.4. Delays in Executing Dependent Instructions

In Section 3.2, we noted that result bypassing (forwarding) can allow RAW-dependent instructions to execute in consecutive cycles even if there exist wire delays between the issue queue and the ALUs. If the dependent instructions

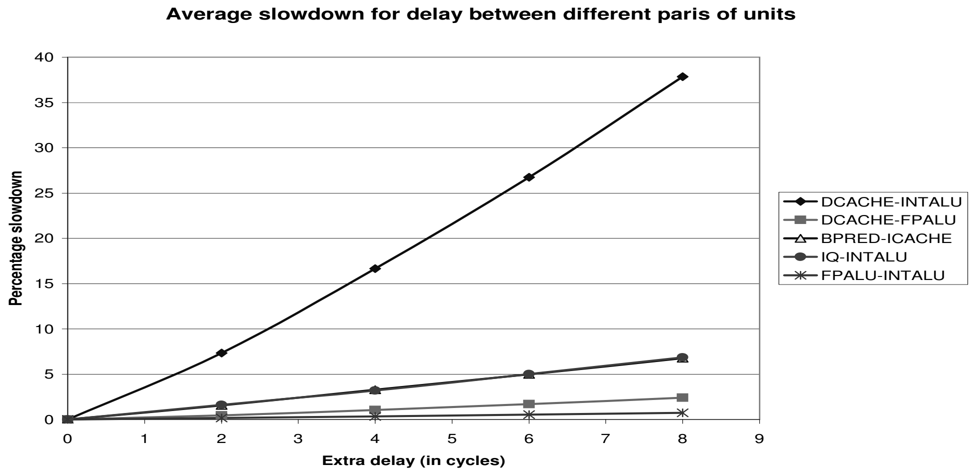


Figure 8. Effect of wire delays between various execution clusters.

execute on different ALUs that are placed far apart, wire delays for bypassing can increase the gap between the execution of these dependent instructions. These delays were not considered in the analysis in [14] (or may have been attributed to other aspects of the pipeline). We assume that there exist three clusters of execution units: integer ALUs, floating-point ALUs, and the load/store unit. The load store unit contains the L1 data cache and load/store queue (LSQ). The integer and FP execution units also contain the corresponding issue queues and register files. Consider the following ways in which these clusters communicate with each other.

When executing a load or store operation, the effective address is computed in the integer execution cluster and communicated to the LSQ. After memory disambiguation is completed, the load issues. If the two clusters are placed far apart, the introduced wire delays cause an increase in load latency. Even if the effective address is computed at an ALU within the load/store unit, the communication of integer register values from the integer execution cluster will add to load latency. After a value is fetched from the data cache, it is written into either the integer or floating-point register file. The result may also be bypassed to dependent instructions executing on either the integer or FP ALUs. In all of the above cases, the wire delays between the data cache and the ALUs will determine the gap between dependent instructions. If an integer value is being fetched from the data cache, the load-to-use latency involves two transfers on wires between the integer and load/store units. If a floating-point value is being fetched from the data cache, the load-to-use latency involves one transfer on wires between the integer and load/store unit and one transfer on wires between the load/store unit and FP unit.

Finally, some instructions move results between the integer and floating-point register files. Wire delays between these two units will increase the gap between dependent instructions that execute on different units.

Figure 8 shows the effect of increasing wire delays between each pair of units. The wire delay between integer and FP units has a minor impact on performance. The wire delay between the data cache and integer unit impacts performance for most benchmark programs (overall 38% performance degradation for an 8-cycle delay). The wire delay between the data cache and FP unit similarly impacts performance for most FP programs, although to a much lesser degree. Comparing Figures 5, 7, and 8, we see that a 4-cycle wire delay between units can cause the following performance penalties:

- Int cluster $\leftrightarrow$ Ld/St cluster : 16.7%
- Bpred $\rightarrow$ ICache : 3.3%
- IQ $\rightarrow$ Int-ALU : 3.2% (not including the effect on instruction replay)
- Ld/St cluster $\rightarrow$ FP cluster : 1%

Based on these observations, we claim that floorplans must primarily strive to reduce the bypass delays between the Int and Ld/St clusters. The delays between IQ and Int-ALU must also be reduced to minimize bypassing and instruction replay complexity.

4. Conclusions

The microarchitectural loops within a pipeline determine the gap between dependent instructions. In this preliminary study, we show that wire delays between specific pipeline stages (issue queue and ALU; branch predictor and I-Cache) are not part of critical loops. Instead, bypass delays between the data cache and integer ALUs impact overall IPC the most. If wire delays are introduced within the execution back-end, bypassing complexity and the negative effect of instruction replay can increase. These are important phenomena that have been ignored by floorplanning algorithms to date. For future work, we will carry out a more detailed characterization of various wire delays on different pipelines and include the effect of instruction replay in our simulator. We also plan to extend floorplanning algorithms such as HotFloorplan [14] to take our performance results into account.

References

[1] E. Borch, E. Tune, B. Manne, and J. Emer. Loose Loops Sink Chips. In *Proceedings of HPCA*, February 2002.

[2] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[3] B. Calder and D. Grunwald. Next Cache Line and Set Prediction. In *Proceedings of ISCA-22*, June 1995.

[4] J. Cong, A. Jagannathan, G. Reinman, and M. Romesis. Microarchitecture Evaluation with Physical Planning. In *Proceedings of DAC-40*, June 2003.

[5] M. Ekpanyapong, M. Healy, C. Ballapuram, S. Lim, H. Lee, and G. Loh. Thermal-Aware 3D Microarchitectural Floorplanning. Technical Report GIT-CERCS-04-37, Georgia Institute of Technology Center for Experimental Research in Computer Systems, 2004.

[6] M. Ekpanyapong, J. Minz, T. Watewai, H. Lee, and S. Lim. Profile-Guided Microarchitectural Floorplanning for Deep Submicron Processor Design. In *Proceedings of DAC-41*, June 2004.

[7] Y. Han, I. Koren, and C. Moritz. Temperature Aware Floorplanning. In *Proceedings of TACS-2 (held in conjunction with ISCA-32)*, June 2005.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1, 2001.

[9] W. Hung, Y. Xie, N. Vijaykrishnan, C. Addo-Quaye, T. Theocharides, and M. Irwin. Thermal-Aware Floorplanning using Genetic Algorithms. In *Proceedings of ISQED*, March 2005.

[10] V. Nookala, D. Lilja, S. Sapatnekar, and Y. Chen. Comparing Simulation Techniques for Microarchitecture-Aware Floorplanning. In *Proceedings of ISPASS*, March 2006.

[11] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA-24*, pages 206–218, June 1997.

[12] K. Puttaswamy and G. Loh. Thermal Analysis of a 3D Die-Stacked High-Performance Microprocessor. In *Proceedings of GLSVLSI*, April 2006.

[13] Samsung Electronics Corporation. Samsung Electronics Develops World's First Eight-Die Multi-Chip Package for Multimedia Cell Phones, 2005. (Press release from `http://www.samsung.com`).

[14] K. Sankaranarayanan, S. Velusamy, M. Stan, and K. Skadron. A Case for Thermal-Aware Floorplanning at the Microarchitectural Level. *Journal of ILP*, 7, October 2005.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of ASPLOS-X*, October 2002.

[16] Tezzaron Semiconductor. (`http://www.tezzaron.com`).