Quantifying the Impact of Inter-Block Wire Delays on Processor Performance

Vivek Venkatesan, Abhishek Ranjan, Rajeev Balasubramonian
School of Computing, University of Utah
{vvenkate, abhishek, rajeev}@cs.utah.edu

Abstract

A chip's operating temperature is emerging as a major design constraint. Floorplanning is an effective technique that helps spread heat and minimize the occurrence of localized hotspots. The floorplanning process may place two communicating microarchitectural blocks (for example, the issue queue and ALU) far apart in an attempt to surround hot blocks with relatively cooler blocks. As we move to future wire-bound technologies, multiple pipeline stages may be required for the communication of signals between blocks that are placed far apart. This paper provides a detailed characterization of how these wire delays impact performance. Such data can serve as useful inputs to floorplanning algorithms so that performance-critical wire delays can be minimized. Routing and placement tools can also exploit this information by implementing low-power wires for the transmission of non-critical signals. In this work, we show that wire delays between the ALU and L1 data cache are most critical to performance, while most other wire delays degrade performance by less than 5% even if they consume up to four cycles. Our results are incorporated in a floorplanning tool and the generated floorplans perform up to 24% better than a floorplanner that does not take such information into account.

Keywords: *microarchitectural floorplanning, wire delays, deep pipelines, critical microarchitectural loops.*

1. Introduction

In recent years, power density and chip temperature have emerged as primary constraints in microprocessor design. Most modern processors today employ recovery mechanisms when thermal emergencies are triggered. A high thermal emergency threshold leads to high packaging and cooling costs, while a low thermal emergency threshold leads to frequent recovery and lower performance. The above design issues become more critical with every new process generation because of increased transistor and power densities. The move to a 3D die-stacked chip [2, 21, 25] further accelerates the climb on the power density curve.

An effective technique to overcome the temperature bottleneck is microarchitectural floorplanning. By placing relatively cool blocks around hot blocks, the rate of lateral heat spreading is improved, thereby reducing the operating temperature of hotspots. Floorplanning algorithms typically employ a simulated annealing process where blocks (such as the register file, rename unit, branch predictor, etc.) are moved around in an attempt to find a floorplan that minimizes an objective function. This objective function can be a combination of metal or silicon area, wire power dissipation, temperature, and performance. To date, no architectural study has provided a detailed characterization of how wire delays between microarchitectural blocks impact performance. Most papers on floorplanning [15, 17, 22] employ performance models that are not very detailed and even have major flaws. For example, these studies [17, 22] indicate that certain wire delays can degrade performance by as much as 65%, but as we show in this paper, simple pipeline optimizations can dramatically cut down these effects. In essence, we show that only a small subset of wires are on the critical path. We hope that our results will serve as an authoritative guideline that VLSI researchers can directly adopt in their floorplanning or routing/placement tools. We integrate our performance data into the HotFloorplan tool [22] and present the results of our case study.

The performance data in this paper is useful in another important context, not explored in this paper. Global wires can be designed in different ways: (i) latency can be minimized with optimal repeater size and spacing and with wide wire width and spacing, (ii) power can be minimized (while incurring a performance penalty) with smaller and fewer repeaters. If a VLSI routing/placement tool is fed with information about the sets of wires that are microarchitecturally non-critical, power-efficient wires can be employed. This interaction between architecture and VLSI tools can help reduce interconnect power consumption while minimally impacting performance.

The paper is organized as follows. Section 2 describes critical microarchitectural loops and how they are impacted by introducing inter-block wire delays. Section 3 provides quantitative results, including the floorplanning case study. We discuss related work in Section 4 and draw conclusions in Section 5.

2. The Interaction of Wire Delays and Loops

2.1. Floorplanning Basics

Floorplanning algorithms employ a simulated annealing process to evaluate a wide range of candidate floorplans. The objective functions for these algorithms have evolved over time. Initial algorithms [11] attempted to minimize total silicon and metal wiring area. With the recent emergence of power and temperature constraints, tools have

augmented the objective function with the activity within wires and the average/peak temperature attained by the floorplan [8, 9, 10, 12, 14, 22]. In modern microprocessors, the delays across global wires can exceed a single cycle. The Intel Pentium4 [13] has a couple of pipeline stages exclusively for signal propagation. As wire delays continue to grow, relative to logic delays and cycle times, we can expect more examples of multi-cycle wire delays within a microprocessor. A floorplanning tool must therefore also consider the performance impact of introducing multi-cycle wire delays between two communicating microarchitectural blocks. The objective function in a state-of-the-art floorplanner can be represented as follows:

$$\lambda_A \times Area\_metric + \lambda_T \times Temperature + \sum_{ij} \lambda_W \times W_{ij} \times Activity_{ij} + \sum_{ij} \lambda_I \times d_{ij} \times IPC\_penalty_{ij}$$

In the equation above, $\lambda_A, \lambda_T, \lambda_W$, and λ_I represent constants that tune the relative importance of each metric (area, temperature, wire power, and IPC), W_{ij} represents the metal area (length $\times$ number of wires) between microarchitectural blocks i and j, $Activity_{ij}$ captures the switching activity for the wires between blocks i and j, the metric d_{ij} represents the distance between blocks i and j in terms of cycles, while $IPC\_penalty_{ij}$ is the performance penalty when a single cycle delay is introduced between blocks i and j. The metrics W_{ij}, d_{ij}, $Temperature$, and $Area\_metric$ are computed for every floorplan being considered, while metrics $Activity_{ij}$ and $IPC\_penalty_{ij}$ are computed once with an architectural simulator and fed as inputs to the floorplanner. The design of efficient floorplanners remains an open problem and many variations to the above objective function can be found in the literature [8, 9, 10, 12, 14, 22]. One of the goals of this paper is to accurately characterize the $IPC\_penalty$ term – this data can be invaluable to VLSI researchers that are exploring floorplanning algorithms and can directly use these as inputs to their tools. For our floorplanner case study evaluation, we will consider a baseline objective function and demonstrate the effect of including an accurate $IPC\_penalty$ term.

2.2. Critical Microarchitectural Loops

Consider the superscalar out-of-order pipeline shown in Figure 1. The pipeline is decomposed into the standard microarchitectural blocks and key data transfers between blocks are indicated with solid lines. Borch et al. [3] define a microarchitectural loop as the communication of a pipeline stage's result to the input of that same pipeline stage or an earlier stage. Loops typically indicate control, data, or structural dependences. The length of the loop is the number of pipeline stages between the destination and origin of the feedback signal. If the length of the loop is increased, it takes longer to resolve the corresponding dependence, thereby increasing the gap between dependent instructions and lowering performance. If a floorplanning tool places pipeline stages far apart and introduces wire delays between them, the lengths of multiple loops may be increased. The dashed lines in Figure 1 represent important loops within an out-of-order superscalar processor.

- **Instruction Fetch Loop:** In a simple pipeline, the process of instruction fetch involves the following steps: the PC indexes into the branch predictor system to produce the next-PC, the corresponding line is fetched from the I-cache, instructions are decoded, and when the next control instruction is encountered, it is fed back to the branch predictor system. This represents a rather large loop with a few stall cycles in fetch every time a control instruction is encountered. Introducing wire delays between the branch predictor, I-cache, and decode can severely degrade performance and this pessimistic model was assumed in HotFloorplan [22]. However, it is fairly straightforward to decouple the branch predictor and I-cache so that we instead have two short loops (labeled 2a and 2b in Figure 1). Such decoupled pipelines have been proposed by academic [20] and industrial groups [16].

  In one possible implementation, the output of the branch predictor (the start of the next basic block) is fed as input back to the branch predictor. As a result, the branch predictor system is now indexed with the PC that starts the basic block, not the PC that terminates the basic block. Updates to the branch predictor system must correspondingly also use the basic block start PC. This allows the branch predictor to produce basic block start PCs every cycle. Our results show that this change in the branch predictor algorithm has a minimal impact on its accuracy. The outputs of the branch predictor can be buffered at the I-cache. Every cycle, the I-cache reads out either the line corresponding to the next queued basic block or the next sequential line if a control instruction is not encountered. Since the presence of a control instruction in the fetched line is not known until the end of decode, it may be necessary to maintain a bit for every I-cache line to indicate if the line contains a control instruction. This bit is set after the line is decoded the first time. Thus, the I-cache can produce a new line every cycle. I-cache fetch cycles are wasted only if the line is being decoded the first time and the default prediction is incorrect. The front-end pipeline now consists of two major tight loops: the branch predictor loop (2a) and the I-cache loop (2b). The front-end is also part of the branch mis-predict resolution loop (1), which feeds from the ALU stage all the way back to the front-end.

  The effect of introducing wire delays between front-end pipeline stages can be summarized as follows. Delays between the branch predictor and I-cache will only impact branch mispredict penalty. In addition to impacting

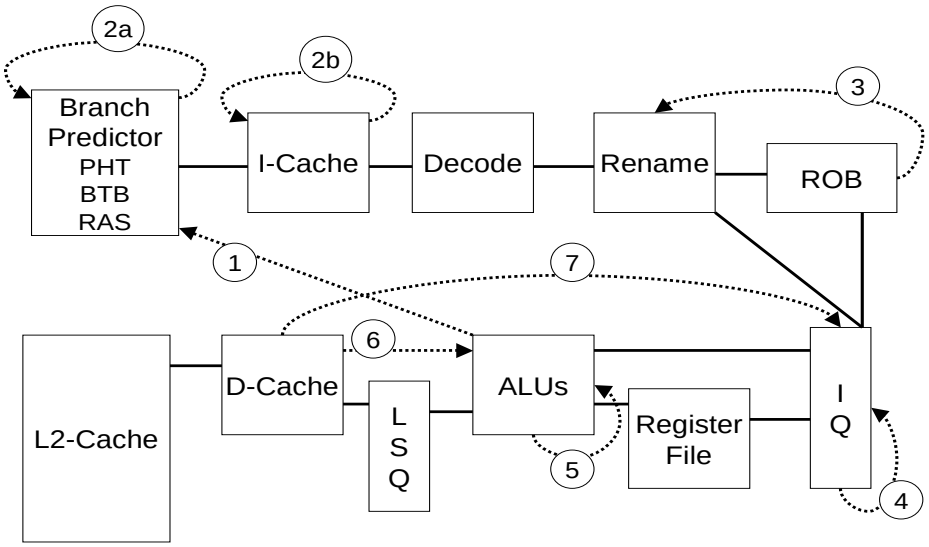

Figure 1. **Critical microarchitectural loops in an out-of-order superscalar pipeline.**

branch mispredict penalty, delays between the I-cache and decode will also introduce fetch stall cycles the first time a line is decoded and the default prediction about the existence of a control instruction is incorrect. As we will subsequently see, the above delays have a minor impact on IPC. Our relative results will hold true even if a different pipeline implementation is adopted, as long as the critical loops are short. For example, consider the decoupled front-end implementation of the Alpha 21264 [16]. The I-cache is augmented with a next-line-and-set (NLS) predictor [6] that allows the I-cache to produce a new line every cycle. A more accurate branch predictor over-rides the NLS prediction in case of a disagreement. The fetch loop is therefore tight in the common case. Wire delays between the branch predictor and I-cache will impact IPC only in the uncommon case where the branch and NLS predictors disagree.

- **Rename Loops:** The register rename stage is not part of any critical loop. The introduction of wire delays either between the decode and rename stages or between the rename and issue queue stages will lengthen the penalty for a branch mispredict (loop 1). Since registers are allocated during rename, wire delays between the rename stage and the issue queue will increase the duration that a register entry remains allocated (loop 3). This increases the pressure on the register file and leads to slightly smaller in-flight instruction windows, on average.

- **Wakeup and Bypass Loops:** There is a common misconception that wire delays between the issue queue and ALUs will lead to stall cycles between dependent instructions [17, 22]. This is not true because the pipeline can be easily decomposed into two tight loops – one for wakeup (loop 4) and one for bypass (loop 5). When an instruction is selected for issue in cycle N, it first fetches operands from the register file, potentially traverses long wires, and then finally reaches the ALU. Because of these delays, the instruction may not begin execution at the ALU until the start of cycle $N + D$. If the ALU operation takes a single cycle, the result is bypassed to the inputs of the ALU so that a dependent instruction can execute on that ALU as early as the start of cycle $N + D + 1$. For this to happen, the dependent instruction must leave the issue queue in cycle $N + 1$. Therefore, as soon as the first instruction leaves the issue queue, its output register tag is broadcast to the issue queue so that dependents can leave the issue queue in the next cycle. Thus, operations within the issue queue must only be aware of the ALU latency, and not the time it takes for the instruction to reach the ALU (delay D). The gap between dependent instructions is therefore not determined by delay D, but by the time taken for the wakeup loop and by the time taken for the bypass loop (both of these loops were assumed to be 1 cycle in the above example). The introduction of wire delays between the issue queue and ALU because of floorplanning will not impact either of these loops (for now, we assume that the ALUs are not partitioned into multiple blocks).

  However, wire delays between the issue queue and ALU will impact another critical loop that has been disregarded by every floorplanning tool to date. This is the load hit speculation loop (loop 7 in Figure 1). The issue queue schedules dependent instructions based on the expected latency of the producing instruction. While most instructions have fixed latencies, a load instruction's latency depends on the location of data in the cache hierarchy. In modern processors, such as the Pentium4 [13], the issue queue optimistically assumes that the load will hit in the L1 data cache and accordingly schedules dependents. If the load latency is any more than this minimum latency, dependent instructions that have already left the issue queue are squashed. These instructions

Pipeline stages involved in wire delay	Critical loops affected
Branch predictor and L1I-Cache	Branch mispredict penalty
I-Cache and Decode	Branch mispredict penalty, penalty to detect control instruction
Decode and Rename	Branch mispredict penalty
Rename and Issue queue	Branch mispredict penalty and register occupancy
Issue queue and ALUs	Branch mispredict penalty, register occupancy, L1 miss penalty, load-hit speculation penalty
Integer ALU and L1D-Cache	load-to-use latency, L1 miss penalty, load-hit speculation penalty
FP ALU and L1D-Cache	load-to-use latency for floating-point operations
Integer ALU and FP ALU	dependences between integer and FP operations
L1 caches and L2 cache	L1 miss penalty
Clusters in a clustered microarchitecture	inter-cluster dependences

Table 1. Effect of wire delays on critical loops.

will subsequently be re-issued after the load miss returns. To facilitate this replay, instructions are kept in the issue queue until the load latency is known. 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 instruction window, on average. The first factor comes into play on a load-hit mis-speculation, while the second factor impacts performance even for correct speculations. If any wire delays are introduced in the pipeline between the issue queue and ALU[1], or between the ALU and data cache, it takes longer to determine if a load is a hit or a miss. Correspondingly, the penalties for correct and incorrect load-hit speculations increase. We also model the Tornado effect [18], where an entire chain of instructions dependent on the load are issued, squashed, and replayed on a load miss.

Delays between the issue queue and ALUs also impact the branch mispredict penalty and register occupancy. Further, once the L1 miss is resolved, it takes longer to re-start the pipeline and for the dependent instructions to begin execution on the ALUs. For example, the gap between a load (that hits in L2) and its dependent instruction may be 20 cycles in a baseline processor. If two wire delay stages are introduced between the issue queue and ALU, the gap between the load and its dependent grows to 22 cycles. In essence, the cost of an L1 miss grows as a function of the wire delays introduced between the issue queue and ALU.

- **Bypassing Loops between Groups of Functional Units:** We begin this discussion by assuming that the functional units are organized as three clusters: integer ALUs, floating-point ALUs, and memory unit. The memory unit is composed of the load-store queue (LSQ) and L1 data cache. The ALUs that compute the effective addresses for loads and stores are part of the integer cluster. Bypassing within a cluster does not cost additional cycles. If wire delays are introduced between the integer and floating-point clusters, performance will be impacted for those integer operations that are data-dependent on a floating-point result, and vice versa. As discussed in the previous item, the introduction of wire delays between the integer cluster and memory unit will increase the penalties for load-hit speculations and pipeline re-start after an L1 miss. More importantly, this delay impacts the load-to-use latency (loop 6 in Figure 1). If a single cycle delay is introduced, it takes one more cycle to communicate the effective address to the cache and an additional cycle to forward the value back to dependent instructions in the integer cluster. Even if the ALU that generates the effective address is part of the memory unit, the penalty is similar because a register value produced in the integer cluster will likely have to be forwarded as input to the ALU. As a result, the gap between a load and a dependent integer operation increases by two cycles. Similarly, if a single cycle wire delay is introduced between the memory unit and floating-point cluster, the gap between a load and a dependent floating-point operation increases by one cycle.

  Now consider the case where the integer ALUs are themselves distributed across multiple clusters, similar to the Alpha 21264 [16] microarchitecture. Bypassing an operand within a cluster imposes little wire delay penalty, but bypassing an operand between clusters is more expensive. An instruction steering heuristic attempts to balance load across clusters and steer dependent instructions to the same cluster. However, it is impossible to localize all dependent instructions and typically, there exist numerous critical inter-cluster data transfers. An increase in inter-cluster wire delays will therefore increase the gap between every pair of dependent instructions that are assigned to different clusters.

- **Cache Hierarchy Loops:** The wire delay between the L1 data cache controller and the L2 cache controller directly impacts the latency for an L1 data cache miss. Similarly, the delay between the L1 instruction cache controller and the L2 cache controller impacts the latency for an L1 instruction cache miss.

Table 1 summarizes the different ways that wire delays can impact performance.

[1]The ALU is used to compute the effective address of the load instruction.

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, 20 FP	Register file 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 first block
L1 D-cache	32KB 2-way 2-cycle	L2 unified cache	2MB 8-way, 30 cycles
ROB/LSQ size	80/40	I and D TLB	128 entries, 8KB page size

Table 2. Simplescalar simulator parameters.

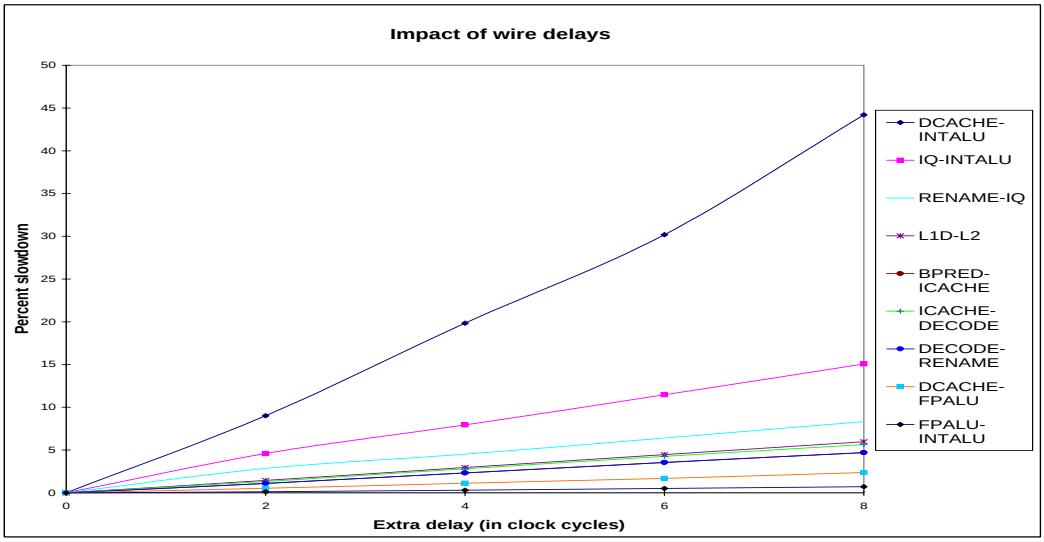

Figure 2. IPC slow-down curves

3. Results

3.1. Methodology

The simulator used in this study is based on SimpleScalar-3.0 [5], a cycle-accurate simulator for the Alpha AXP architecture. It is extended to not only model separate issue queues, register files, and reorder buffer, but also the micro-architectural loops and architectural features discussed in Section 2.2. The benchmark suite includes 23 SPEC-2k programs, executed for 100 million instruction windows identified by the Simpoint tool [23]. The processor parameters for the base configuration are listed in Table 2.

3.2. Impact of Inter-Unit Delays

For each of the critical set of pipeline stages listed in Table 1, additional wire delays of 2, 4, 6, and 8 cycles are introduced. The resulting IPC degradation curves, relative to the baseline, are charted in Figure 2. It is evident that wire delays between the ALU and data cache have the greatest impact on performance, causing an average slowdown of 20% for a 4-cycle delay. Integer programs are impacted more than FP programs, with *gap*, *gzip*, and *bzip2* exhibiting slowdowns of greater than 40%. As shown in Table 1, delays between the ALU and data cache affect multiple critical loops. The load-to-use loop contributes nearly three-fourth of the 20% observed slowdown, with the remaining attributed to the load-hit speculation loop and L1 miss penalty loop. The load-hit speculation loop also contributes to the second-most critical wire delay, that between the issue queue and ALUs. For a 4-cycle wire delay, the performance degradation is 8%, much lower than the pessimistic 65% degradation reported in [22]. Similarly, because of the decoupled front-end, a 4-cycle wire delay between the branch predictor and I-cache only causes a 2.3% performance loss (instead of the 50% performance loss reported in [22]). The new branch predictor algorithm (indexing with basic block start address instead of basic block end address) affects accuracy by only 0.55%. Only 1.14% of all fetched branches introduce stalls in fetch because the line is new in the I-cache and the bit indicating the presence of a control instruction is not set. All other wire delays are non-critical and cause slowdowns of less than 5% (for a 4-cycle delay).

We also evaluate the effect of inter-ALU wire latency by distributing processor resources (register file, issue queue, ALUs) across two clusters. A state-of-the-art instruction steering mechanism [1] is used to assign instructions to clusters. An additional 4-cycle delay for inter-cluster communication results in a significant slowdown of 13.1% (not shown in Figure 2). For the rest of this paper, we only consider a monolithic architecture and lump all ALUs as a single block.

When a floorplanning algorithm evaluates a floorplan with various wire delays between pipeline stages, it must predict the expected overall IPC. If the effects of different wire delays are roughly additive, it is fairly straightforward to pre-

dict the IPC of a configuration with arbitrary wire delays between pipeline stages. The IPC slowdown (relative to the baseline) for such a processor equals $\sum_i d_i.\mu_i$, where d_i represents each wire delay and μ_i represents the slope of the corresponding slowdown curve in Figure 2. If this hypothesis is true, detailed architectural simulations can be avoided for every floorplan that is considered. To verify this hypothesis, we simulated ten processor configurations with random wire delays (between 0 and 4 cycles) between every pair of pipeline stages. The wire delays for these configurations are shown in Table 3. Figure 3 compares the experimental IPC slowdowns against the theoretical slowdown computed according to the slopes of the slowdown curves. The experimental slowdown closely matches the theoretical slowdown, with an average IPC error of 2.1% and a maximum error of 4%. This minor discrepancy is partially because the slowdown curve is being represented as a straight line by a single slope value.

	cfg1	cfg2	cfg3	cfg4	cfg5
Dcache-IntALU	3	2	1	4	0
Dcache-DFPALU	3	0	4	3	1
FP-IntALU	1	1	3	4	4
Bpred-Icache	0	2	0	0	1
Decode-Rename	2	2	1	1	4
Rename-IQ	3	4	0	3	0
Dcache-L2	4	1	2	2	3
Decode-Icache	3	4	2	2	1
IQ-IntALU	1	2	3	1	3
	cfg6	cfg7	cfg8	cfg9	cfg10
Dcache-IntALU	2	3	0	1	0
Dcache-DFPALU	3	2	3	4	0
FP-IntALU	3	1	3	0	0
Bpred-Icache	4	2	1	0	0
Decode-Rename	0	1	4	2	4
Rename-IQ	2	0	0	4	2
Dcache-L2	3	1	0	3	2
Decode-Icache	0	1	1	1	1
IQ-IntALU	4	1	4	4	4

Table 3. Critical path latencies for 10 random configurations

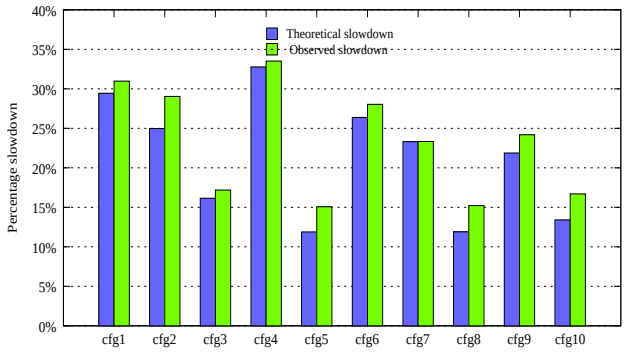

Figure 3. Additive nature of IPC degradation

Next, we examine various processor design points to verify if the criticality of wire delays is a function of the processor configuration. Table 4 summarizes the properties for four out-of-order processors (ranging from "Poor" to "Super") and one in-order configuration. Figure 4 demonstrates the IPC degradation when 8-cycle wire delays are introduced between each set of pipeline stages for all five processor configurations. For all the out-of-order configurations, our broad conclusions hold true: the ALU-Dcache delay is most critical, followed by the IQ-ALU delay. However, we can see that the magnitude of the slowdown due to the ALU-Dcache delay increases as we go from a poor configuration to a good configuration. The rationale behind this is that in a high-IPC model, any available ILP is quickly mined. Long-latency operations tend to be on the critical path and any additional delays to these instructions will almost certainly increase overall execution time. For the in-order model, the ALU-Dcache delay stands out, yielding a 162% performance slowdown. Note that a single cycle wire delay between the ALU and Dcache increases load latency by two cycles, effectively stalling all subsequent instructions in the in-order processor by two additional cycles.

	Poor	Base	Good	Super	Inorder
Issue	o-o-o	o-o-o	o-o-o	o-o-o	in-order
Dec/Iss/Comm width	4	4	8	8	4
ROB size	56	80	128	256	80
L1-Dcache	16K	32K	64K	128K	32K
L1-Icache	16K	32K	64K	128K	32K
L2-cache	1MB	2MB	4MB	4MB	2MB
Mem. Ports	2	2	2	4	2
IntALU/IntMul	2/1	4/2	6/2	8/4	4/2
FPALU/FPMul	2/1	2/1	4/1	8/2	2/1

Table 4. Parameters for five different processor configurations.

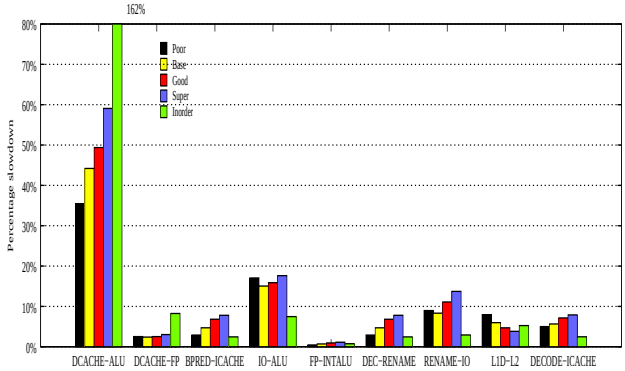

Figure 4. IPC slowdowns for each 8-cycle wire delay for various processor configurations.

3.3. Behavior of Floorplanning Algorithms

As shown in Section 2, the objective function for the floorplanner can include a term for IPC penalty. The slopes of the slowdown curves in Figure 2 are used to compute the $IPC\_penalty$ weights for each set of wires. These weights are listed in Table 5. The average power values for each microarchitectural block are derived from the Wattch power model [4] and incorporated into the HotFloorplan tool [22]. It is assumed that all blocks can be rotated and the maximum aspect ratio for blocks range from 3 to 6. We generate two floorplans:

Critical loop	Weight
DCACHE-INTALU	18
DCACHE-FPALU	1
BPRED-ICACHE	2
IQ-INTALU	6
FP-INTALU	1
DECODE-RENAME	2
RENAME-IQ	4
DCACHE-L2	2
DECODE - ICACHE	2

Table 5. Weights for the different pairs of blocks

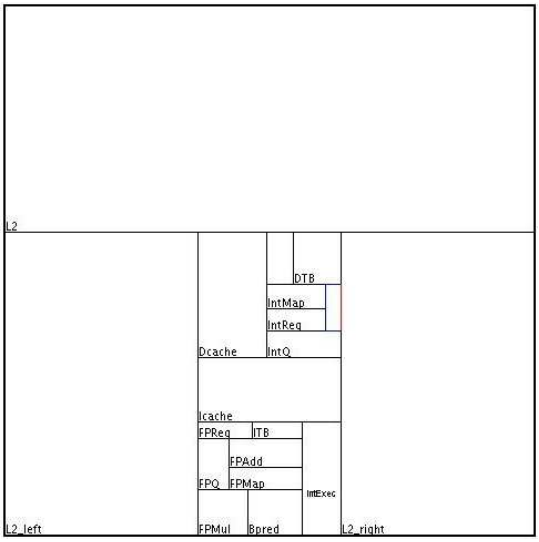

Figure 5. Basic floorplan

- *Basic floorplan:* This is generated with no knowledge regarding the criticality of various wire delays (the weight for every set of wires is set to 1). This floorplan is shown in Figure 5.

- *Performance-optimized floorplan:* This floorplan assumes weights for each set of wires as shown in Table 5. The floorplan is shown in Figure 6.

The characteristics of each floorplan are summarized in Table 6. The total wire length for the performance-optimized floorplan is much higher than that of the basic floorplan, but crucial wires are kept short even if it compromises the wire lengths of other non-critical wires. For example, the data cache and integer execution units are placed far apart in the basic floorplan, but are adjacent in the performance-optimized floorplan. The peak temperature of the performance-optimized floorplan is only slightly higher than that of the basic floorplan.

To compare the performance of these floorplans, we determined the distances between interacting blocks and computed wire latencies for two different types of wires – faster global wires on the 8X metal plane and semi-global wires on the 4X plane. These latencies were converted to cycles for three different clock speed assumptions – 3 GHz, 5 GHz, and 10 GHz. Figure 7 shows the IPC improvement achieved with the performance-optimized floorplan for the six different combinations of clock frequency and metal plane. The

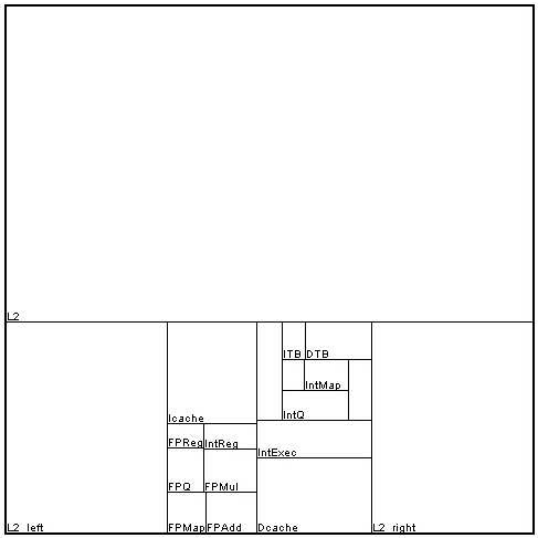

Figure 6. Critical-path optimized floorplan

	Basic	Performance-Optimized
Total area	$2.56960cm^2$	$2.56942cm^2$
Wire-length metric	0.081	0.220
Peak temperature	336.631K	337.872K
Avg. temperature	326.004K	325.49K

Table 6. Properties of the floorplans

improvement ranges from 5 to 25%. The performance improvement is a direct result of a defining feature of the optimized floorplan; the three most delay-sensitive wires - the Dcache-IntALU loop, the IQ-IntALU loop and the Rename-IQ loop are found to be 3.72X, 4.74X and 1.33X longer in the basic floorplan, when compared with the optimized floorplan.

4. Related Work

Numerous automated floorplanning tools and algorithms exist in the literature [8, 9, 10, 11, 12, 14, 19, 22]. The objective function for some of these tools is purely performance [8], while for others it is some combination of temperature and wire communication [9, 10, 12, 14, 19, 22]. To date, there is no architectural evaluation that comprehensively quantifies the effect of wire delays between var-

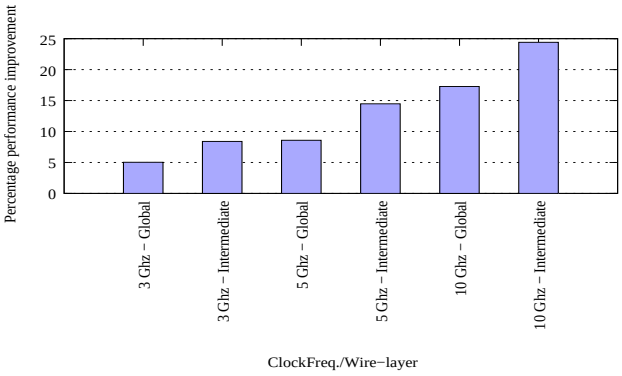

Figure 7. Percentage performance improvement with critical-path sensitive floorplanning

ious microarchitectural blocks. Some of this data can be found scattered in the literature. For example, Sprangle and Carmean [24] quantify the effect of increasing the number of cycles for ALU bypass, L1/L2 cache access, and branch mispredict penalty. Borch et al. [3] quantify the effect of increasing the number of cycles in the branch prediction and load-hit speculation loops.

The MEVA floorplanner [7, 15] adopts the methodology of Sprangle and Carmean [24]. Other floorplanning tools [17, 19, 22] over-estimate the IPC effect of wire delays because they do not consider simple pipeline optimizations. For example, (i) Long et al. [19] report IPC differences of up to 60% for various floorplans, (ii) Sankaranarayanan et al. [22] report an IPC penalty of 50% and 65% when four cycles of delay are introduced between branch predictor and I-Cache, and between issue queue and integer execution units, respectively. As we show in this paper, inter-block wire delays are not as critical as they have been made out to be.

5. Conclusions and Future Work

This paper presents a methodology to determine the effect of inter-block wire delays on performance. We observed that the relative importance of wires was constant across a range of out-of-order processor configurations. The computed weights may therefore be directly input to state-of-the-art floorplanning algorithms. We also observed that the IPC effects of various wire delays are roughly additive, obviating the need for detailed architectural simulations during the floorplanning process. We show that the wire delay between the ALU and data cache is most critical to performance because of its impact on three critical loops. The performance-optimized floorplan can out-perform a basic floorplanner by up to 25%, while incurring a minor increase in peak temperature. As future work, we will extend our floorplanning algorithm to 3D. We will also investigate if power savings and temperature reductions can be achieved by routing non-critical signals on wires that are optimized for low power.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-Off in Future Clustered Processors. In *Proceedings of ISCA-30*, pages 275–286, June 2003.

[2] B. Black, D. Nelson, C. Webb, and N. Samra. 3D Processing Technology and its Impact on IA32 Microprocessors. In *Proceedings of ICCD*, October 2004.

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

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of ISCA-27*, pages 83–94, June 2000.

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

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

[7] J. Cong, A. Jagannathan, Y. Ma, G. Reinman, J. Wei, and Y. Zhang. An Automated Design Flow for 3D Microarchitecture Evaluation. In *Proceedings of ASP-DAC*, January 2006.

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

[9] 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.

[10] 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.

[11] S. Gerez. *Algorithms for VLSI Design Automation*. John Wiley & Sons, Inc., 1999.

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

[13] 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.

[14] 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.

[15] A. Jagannathan, H. Yang, K. Konigsfeld, D. Milliron, M. Mohan, M. Romesis, G. Reinman, and J. Cong. Microarchitecture Evaluation with Floorplanning and Interconnect Pipelining. In *Proceedings of ASP-DAC*, January 2005.

[16] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[17] W. Liao and L. He. Full-Chip Interconnect Power Estimation and Simulation Considering Concurrent Repeater and Flip-Flop Insertion. In *Proceedings of ICCAD*, 2003.

[18] Y. Liu, A. Shayesteh, G. Memik, and G. Reinman. Tornado Warning: The Perils of Selective Replay in Multithreaded Processors. In *Proceedings of ICS*, June 2005.

[19] C. Long, L. Simonson, W. Liao, and L. He. Floorplanning Optimization with Trajectory Piecewise-Linear Model for Pipelined Interconnects. In *Proceedings of DAC*, 2004.

[20] G. Reinman, T. Austin, and B. Calder. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of ISCA-26*, May 1999.

[21] 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`).

[22] 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.

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

[24] E. Sprangle and D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proceedings of ISCA-29*, May 2002.

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