QUANTIFYING THE IMPACT OF INTERBLOCK WIRE-DELAYS ON PROCESSOR PERFORMANCE

by

Vivek Venkatesan

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2008

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Vivek Venkatesan

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Rajeev Balasubramonian
	John B Carter
	Ganesh Gopalakrishnan

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Vivek Venkatesan in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Rajeev Balasubramonian
Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

As logic delays continue to decrease with smaller process technology, on-chip wire delays are growing exponentially and are expected to increase cross-chip communication latencies to tens of cycles. In this work, we quantify the performance impact of wire-delays in three important contexts: (i) within an aggressive out-of-order (OoO) processor pipeline on a two-dimensional (2D) plane, (ii) within a three-dimensional (3D) die-stacked processor and (iii) within coherence communication paths of a chip multiprocessor.

We perform a detailed characterization of the loops in a super-scalar pipeline and show that previous attempts to characterize the impact of wire-delays on performance over-estimate the IPC degradation for some loops. We observe that most loops tend to become less critical as more speculation and simple optimizations are introduced. three-dimensional stacking allows dies to be bonded with each other in the vertical dimension enabling further reduction in wire-length. We incorporate the data from the criticality study into a floor-planner that leverages 3D to reduce the lengths of the most critical interblock wires. The overall results argue against leveraging 3D to improve single-core performance and shows that IPC-aware 2D floor-plans perform within an acceptable range of 3D.

Coherence operations on multicore architectures necessitate frequent communication over global on-chip wires. Different coherence messages may have varying delay-tolerance levels. We quantify the sensitivity to wire delays for each type of coherence message in an OoO multiprocessor model employing directory-based cache coherence. We observe that OoO processors are able to hide latency in coherence operations adequately and hence there is potential to save considerable power over the interconnect by employing efficient power-optimization strategies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful to my advisor Prof. Rajeev Balasubramonian for being my mentor and inspiration throughout my graduate school life. All credit for opening me up to the world of computer architecture, a place where I never thought I would be, falls squarely on his shoulders. I thank my committee members Prof. John Carter and Prof. Ganesh Gopalakrishnan for their valuable guidance and comments on my work. I thank Abhishek Ranjan, Manu Awasthi and Devyani Ghosh for their collaboration in parts of this work. I also thank Niti Madan and Liqun Cheng for their valuable help with the SMT Model for SimpleScalar and the Wisconsin GEMS memory model respectively. I thank all fellow Impulse lab mates and architecture reading club members for making my years in graduate school thoroughly enjoyable. Finally I would like to thank my parents, brother and friends for their unconditional love and support through all my endeavours.

CHAPTER 1

INTRODUCTION

Advancements in silicon fabrication technology have driven improvements in processor performance because of smaller transistor sizes. The reasons for improvement are twofold: smaller sizes allow greater transistor densities and hence more complex functionality, and secondly the logic delay of a transistor also reduces because of smaller gate lengths. However, with smaller transistor sizes, wire width and height also decrease, resulting in larger wire resistances. The increased resistance coupled with faster clock rates and increasing die area will cause wire delay between micro-architectural blocks to extend beyond tens of cycles. Hence wire delay is becoming a critical component of overall performance, limiting the effectiveness of architectural features that require global communication or unduly increasing the distance between portions of a processor. This work illustrates the importance of studying the impact of these wire delays on the overall performance of a processor.

In recent years, power density and chip temperature have emerged as primary constraints in microprocessor design. They become more critical with every new process generation because of increased transistor and power densities. The move to a 3D die stacked chip [1, 2, 3] further accelerates the climb on the power density curve. An effective technique to overcome the temperature bottleneck is micro-architectural 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 a large number of arbitrary arrangements of blocks (such as the register file, rename unit, branch predictor, etc.) are evaluated based on

an objective function. The objective function attempts to find a floorplan that minimizes temperature, metal or silicon area and wire density with little regard to the performance impact of varying interblock wirelengths.

1.1 Motivation

To date, no architectural study has provided a detailed characterization of how wire delays between micro-architectural blocks impact performance. Most papers on floorplanning [4, 5, 6] employ performance models that are not very detailed or even have some flaws. For example, these studies [5, 6] indicate that certain wire delays can degrade performance by as much as 65%, but do not consider simple pipeline optimizations that can dramatically cut down these effects. We present results that can serve as an authoritative guideline that VLSI researchers can directly adopt in their floorplanning or routing/placement tools.

The performance data from this work can also be useful in another important context. The vertical stacking of dies allows microprocessor circuits to be implemented across three dimensions. This allows a reduction in distances that signals must travel. By reducing overall wire lengths, 3D implementations can help alleviate the performance and power overheads of on-chip wiring. The primary disadvantage of 3D chips is that they cause an increase in power densities and on-chip temperatures. The true potential of 3D can be estimated only with knowledge of the criticality of different interblock wires.

Many recent advances have been made in fabricating 3D chips ([7] presents a good overview). This technology can incur a nontrivial cost because of increased design effort, reduced manufacturing yield, and higher cooling capacities. Even though the technology is not yet mature, early stage architecture results are required to understand its potential. There are likely three primary avenues where 3D can provide benefits:

- 3D stacking of DRAM chips upon large scale Chip Multi Processors (CMP): Interdie vias can take advantage of the entire die surface area to implement a

high bandwidth link to DRAM, thereby addressing a key bottleneck in CMPs that incorporate over hundred cores [8, 2].

- "Snap-on" analysis engines: Chips employed by application developers can be fitted with additional stacked dies that contain units to monitor hardware activity and aid in debugging [9]. Chips employed by application users will not incorporate such functionality, thereby lowering the cost for these systems.

- Improvement in CPU performance/power: The components of a CPU (cores, cache banks, pipeline stages, individual circuits) can be implemented across multiple dies. By lowering the penalties imposed by long wires, performance and power improvements are possible.

The third approach above can itself be classified in two ways: *folding* partitions a single structure (e.g., register file) across multiple dies in order to reduce its access time; *stacking* preserves the structure of individual circuit blocks but leverages 3D to reduce interblock distances. Most recent work has focused on the *folding* approach [10, 11, 12, 13, 14, 15]. Results have shown that this can typically help reduce the delays within a pipeline stage by about 10%, which in turn can contribute to either clock speed improvements or ILP improvements (by supporting larger structures at a target cycle time). The disadvantage with the folding approach is that potential hotspots (e.g., the register file) are partitioned and placed vertically, further exacerbating the temperature problem. Much design effort will also be invested in translating well established 2D circuits into 3D.

The primary advantage of the *stacking* approach is the ability to reduce operating temperature by surrounding hotspots with relatively cool structures. It also entails less design complexity as traditional 2D circuits can be re-used. A third advantage is a reduction in wire delay/power for interconnects between various micro-architectural structures. It has, however, received relatively little attention. We intend to focus on this approach by integrating many varied aspects (loop analysis, pipeline optimizations, SMT, automated floorplanning) in determining the impact of 3D on single core performance. To understand the potential benefit of the

stacking approach, however, it is necessary that we first quantify the performance impact of wire delays between micro-architectural structures.

A study of wire delays can also be useful when applied to coherence protocols. In future microprocessors, as the number of cores scales beyond tens and hundreds, more scalable coherence protocols are needed, and directory based designs have been most popular so far [16, 17]. Several studies [18, 19, 20, 21, 22] have characterized the high frequency of cache misses in parallel workloads, and how these misses significantly hurt the total execution time. On a cache miss, a variety of protocol actions are initiated, such as request messages, invalidation messages, intervention messages, data block write backs, data block transfers, etc. Every coherence message involves on-chip communication with latencies that are projected to grow to tens of cycles in future billion transistor architectures [23]. Some of these can tolerate long latencies, whereas others are on the program critical path. Further, speculation within the core can hide the cost of some of these wire delays. For example, on a cache write miss, the requesting processor may have to wait for data from the home node (a two hop transaction), and for acknowledgments from other sharers of the block (a three hop transaction). Since acknowledgments are on critical path, every cycle of delay in acknowledgments has higher chances to hurt performance than a data block transfer delayed by a cycle. However, an in-depth analysis of the delay sensitivities of coherence messages is required to comprehend the bigger picture.

Cheng et al. [24, 25] have shown that different coherence protocol messages in a directory based protocol have varying bandwidth and latency needs. This is done by identifying message flows within the context of individual coherence operations. However, such an analysis is incognizant of the frequency of the different coherence operations and the interactions among them. For example, the study reports invalidate messages as being critical to performance. However, that may not always be the case; invalidates may not be so frequent as to make a difference in overall performance. A more detailed analysis of the delay sensitivities of coherence messages for an application as a whole will be more beneficial. Our

analysis considers the effect of wire delays in directory based coherence protocols for out-of-order execution engines.

1.2 Thesis Statement

We believe that the criticality analysis of wire delays in the context of uniprocessors or shared memory multiprocessors will prove as an effective tool to selectively optimize a system for performance or power. This thesis corroborates this statement by applying such an analysis in two contexts, for micro-architectural loops within a pipeline and for coherence communication paths in a shared memory multiprocessor system.

The thesis incorporates the results of the loop analysis into a floorplanning algorithm to produce performance optimized floorplans. The criticality data are also used to identify the effectiveness of the 3D *stacking* approach. The data from the wire delay study within coherence protocols will be used to isolate coherence messages that can be optimized for latency and for power. Other intuitive techniques, not within the scope of this thesis, to utilize the criticality information are discussed.

1.3 Contributions

In this thesis, we expose the impact of wire delays on processor performance and highlight the significance of criticality awareness on processor designs. The primary contributions of this thesis are:

- Carries out the most comprehensive analysis of the impact of wire delays on critical micro-architectural loops of a traditional out-of-order processor pipeline that takes into consideration several common pipeline optimizations. The analysis also identifies and addresses areas where past work is inaccurate.

- Designs and implements a criticality aware floorplanner based on input from the wire delay study. The detailed models of the wire delay analysis help in accurately characterizing the IPC penalty of each critical loop. Our floorplan-

ners are able to automate the process of identifying layouts that minimize the lengths of these critical wires.

- Extends the methodology stated above to generate wire delay optimal 3D floorplans that utilize the extra dimension to minimize critical wire lengths between different units. This enables researchers to draw a conclusion about the effectiveness of the 3D stacking approach.

- Identifies the delay tolerance levels exhibited by coherence messages in a shared memory multiprocessor environment where coherence is maintained with the help of a directory based protocol. We explore variations in sensitivity to wire delay for different consistency models. We propose a list of hardware techniques that incorporate criticality awareness in coherence protocols for improved performance or power efficiency.

1.4 Outline

Before we begin with the criticality analysis of micro-architectural loops, it is essential to identify the factors affecting performance for each of these loops. Chapter 2 qualitatively describes the relationships between wire delays and critical micro-architectural loops, and quantifies these relationships for single and multi-threaded superscalar cores. In Chapter 3, these data are then fed into floorplanning algorithms to derive layouts for 2D and 3D chips that optimize a combination of metrics. We show that 2D layouts are able to minimize the impact of critical wire delays effectively. This result is more optimistic about 2D layouts than some prior work in the area and there is little room for improvement with a 3D implementation for traditional simple superscalar cores. The motivation for performing a criticality analysis on coherence protocols and a few possible techniques to utilize the information are discussed in Chapter 4. We also present the methodology for our evaluation and detail the results of the coherence message analysis. We discuss related work in Chapter 5 and summarize our conclusions and possible future work in Chapter 6.

CHAPTER 2

WIRE DELAYS

An out-of-order superscalar processor has a number of communication paths between microarchitectural structures. For a large enough processor operating at a high frequency, some of these paths may incur multicycle delays. For example, the Pentium4 has a few pipeline stages dedicated for wire delay [26]. A state-of-the-art floorplanning algorithm must attempt to place microarchitectural blocks in a manner that minimizes delays for interblock communication paths, but even the best algorithms cannot completely avoid these delays. As examples, consider the following wire delays that are encountered between pipeline stages in the Pentium4. The floating point, integer, and load/store units cannot all be colocated: this causes the load-to-use latency for floating point operands to be higher than that for integer operands. A recent paper by Black et al. [27] indicates that multicycle wire delays are encountered between the extreme ends of the L1 data cache and integer execution units. Similarly, the paper mentions that wire delays are introduced between the FP register file and FP execution units because the SIMD unit is placed closest to the FP register file (SIMD access to the register file is considered more critical). By introducing a third dimension, we can help reduce on-chip distances and the overall performance penalty of interblock wire delays.

To understand this benefit of 3D, we must first quantify the impact of interblock wire delays on performance and evaluate if 2D floorplanning algorithms yield processors that incur significant IPC penalties from wire delays. In addition to serving as the foundation for our 3D layout study, the data produced here can serve as useful inputs for groups researching state-of-the-art floorplanning tools. It should

be noted that while similar analyses exist in the literature, a few papers report inaccurate results because of simplified models for the pipeline.

2.1 Critical Microarchitecture Loops

Consider the superscalar out-of-order pipeline shown in Figure 2.1. The pipeline is decomposed into the standard microarchitectural blocks and key data transfers between blocks are indicated with solid lines. Borch et al. [28] 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 two microarchitectural structures far apart and introduces wire delays between them (in the form of additional pipeline stages for signal transmission), the lengths of multiple loops may be increased. Hence, to understand the IPC impact of wire delays, we must understand how the length of a loop impacts IPC. Similar, but less detailed evaluations have also been carried out in prior work (such as [28, 29, 6, 30]). The dashed lines in Figure 2.1 represent important loops

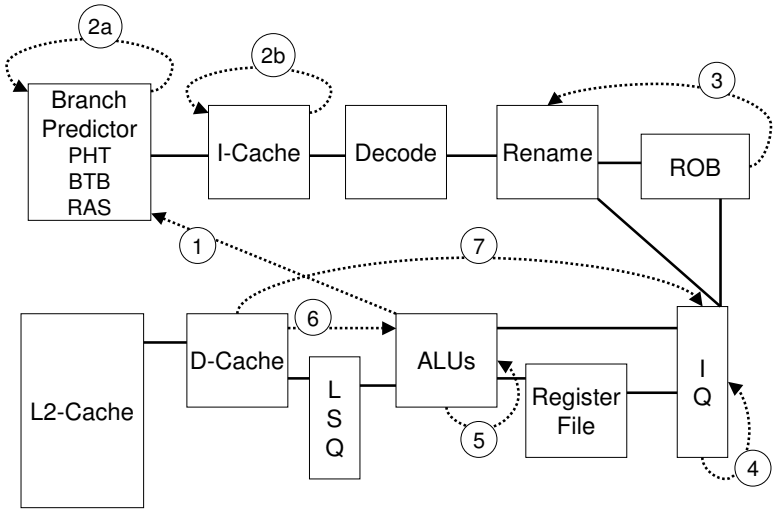

Figure 2.1. Critical microarchitectural loops in an O-o-O superscalar pipeline

within an out-of-order superscalar processor and each loop is discussed next.

2.1.1 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 Instruction cache (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 [6]. 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 2.1). Such decoupled pipelines have been proposed by academic [31] and industrial groups [32].

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 independent of the rest of the front end. 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, a bit is maintained 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 mispredict resolution loop (1), which feeds from the ALU stage all the way back to the front end. Thus, the primary impact of introducing a wire delay between front end pipeline stages is an increase in branch mispredict penalty. Our relative results will hold true even if a different front end pipeline implementation (such as the next-line-and-set predictor in the Alpha 21264 I-cache [32]) is adopted, as long as the critical loops are short. Prior studies [5, 33, 6] have overstated the IPC impact of this wire delay because the branch predictor and I-cache were assumed to not be decoupled.

2.1.2 Rename Loops

The introduction of wire delays either between the decode and rename stages or between the rename and issue queue stages lengthens the penalty for a branch mispredict (loop 1). Since registers are allocated during rename, wire delays between the rename stage and the issue queue increase the duration that a register entry remains allocated (loop 3). This increases the pressure on the register file and leads to smaller in-flight instruction windows.

2.1.3 Wakeup and Bypass Loops

There is a common misconception that wire delays between the issue queue and ALUs lead to stall cycles between dependent instructions [5, 6]. 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 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.

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 2.1). 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 [26], 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 and subsequently replayed. To facilitate this replay, instructions can be kept in the issue queue until the load latency is known. Thus, load-hit speculation negatively impacts performance in two ways: (i) replayed instructions contend twice for resources, (ii) issue queue occupancy increases, thereby supporting a smaller instruction window, on average. The first factor comes into play on a load-hit misspeculation, whereas the second factor impacts performance even for correct speculations.

If any wire delays are introduced in the pipeline between the issue queue and ALU, 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 [34], 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 branch mispredict penalty and register occupancy. They also increase the L1 miss penalty as it takes

longer to restart the pipeline after an L1 miss.

2.1.4 Bypassing Loops Between Groups of Functional Units

For this discussion, we assume 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. The introduction of wire delays between the integer cluster and memory unit impacts the load-to-use latency (loop 6 in Figure 2.1) and the penalties for load-hit speculation.

If a single cycle delay is introduced between the memory and integer units, the load-to-use latency increases by two cycles. Similarly, wire delays between levels of the cache hierarchy will increase the cache miss penalties. 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 [32] micro-architecture. 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 intercluster data transfers. An increase in intercluster wire delays will therefore increase the gap between every pair of dependent instructions that are assigned to different clusters.

2.1.5 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 2.1 summarizes the different ways that wire delays can impact performance.

2.2 IPC Impact of Wire Delays

2.2.1 Methodology

The simulator used in this study is based on SimpleScalar-3.0 [35], a cycle-accurate simulator for the Alpha AXP architecture. It is extended to not only model multiple threads and separate issue queues, register files, and reorder buffer, but also the microarchitectural loops and features discussed in Section 2.1. For each of the critical sets of pipeline stages listed in Table 2.1, we introduce additional wire delays of 2, 4, 6, and 8 cycles. The single thread benchmark suite includes 23 SPEC-2k programs, executed for 100 million instruction windows identified by the Simpoint tool [36]. Table 2.2 lists the processor parameters for the base configuration.

Table 2.1. Effect of wire delays on critical loops

Pipeline stages	**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

Table 2.2. SimpleScalar simulator parameters

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 width	4
Dispatch width	4	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

We also repeat our experiments for a core that supports the execution of two threads in SMT fashion and a traditional in-order core. For the multithreaded workload, we form a benchmark set that consists of 10 different pairs of programs. Programs are paired to generate a good mix of high IPC, low IPC, FP, and Integer workloads. Table 2.3 shows our benchmark pairs. Each of the multithreaded workloads are executed until the first thread commits 100 million instructions. Our SMT model employs the ICOUNT [37] fetch policy and all resources (except the ROB) are dynamically shared by the two threads.

2.2.2 Behavior of Single Threaded Workloads

The resulting IPC degradation curves (averaged across the benchmark suite), relative to the baseline processor (that imposes zero interblock wire delay penalties), are charted in Figure 2.2 for single threaded workloads. For the single threaded workloads, 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 four cycle delay. Integer programs are impacted more than FP programs, with some benchmarks exhibiting slowdowns of greater than 40%. As shown in Table 2.1, delays between the ALU and data cache affect multiple critical loops. The load-to-

Table 2.3. Benchmark pairs for the multithreaded workload

Benchmark Set	Set #	IPC Pairing	Benchmark Set	Set #	IPC Pairing
swim-applu	1	FP/FP/Low/High	bzip-fma3d	2	Int/FP/Low/High
bzip-vortex	3	Int/Int/Low/Low	eon-art	4	Int/FP/High/Low
eon-vpr	5	Int/Int/High/High	gzip-mgrid	6	Int/FP/Low/Low
mesa-equake	7	FP/FP/High/High	swim-lucas	8	FP/FP/Low/Low
twolf-equake	9	Int/FP/High/High	vpr-gzip	10	Int/Int/High/Low

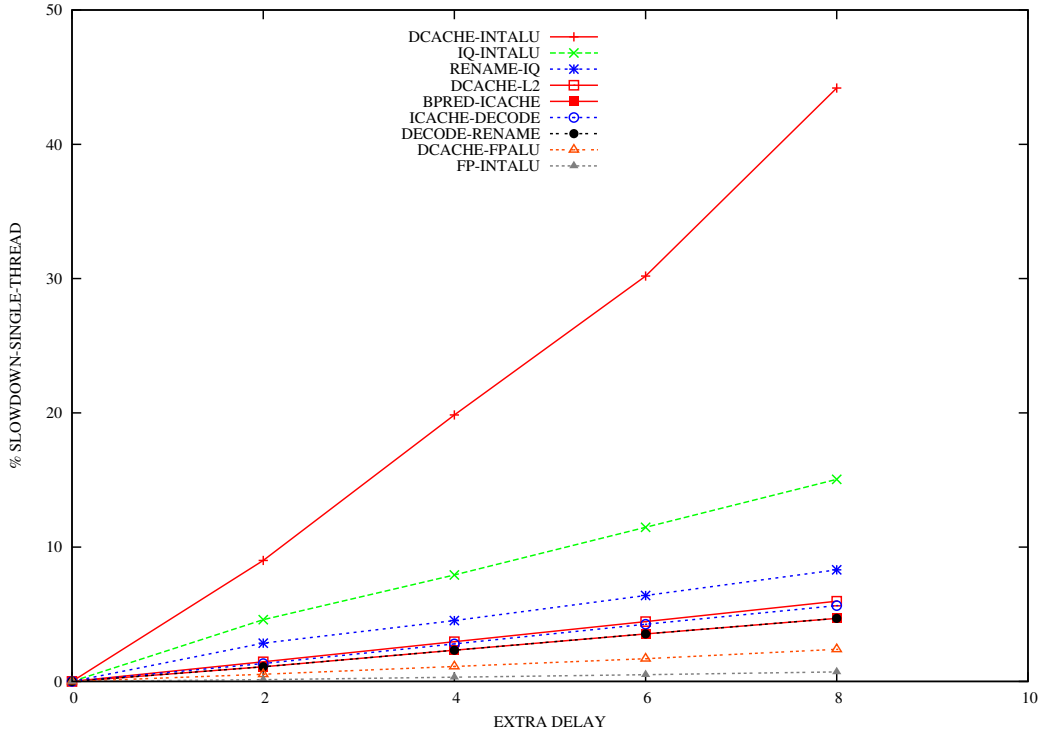

Figure 2.2. IPC slowdown curves for single threaded workloads

use loop contributes nearly three-fourth of the 20% observed slowdown, with the remaining attributed to the load-hit speculation loop (1.7%), and L1 miss penalty loop (2.6%).

The load-hit speculation loop also contributes to the second most critical wire delay, that between the issue queue and ALUs. Since the wakeup and bypass loops are decoupled, a four cycle wire delay between the issue queue and ALU only causes a performance degradation of 8%, much lower than the pessimistic 65% degradation reported in [6]. Similarly, because of the decoupled front end, a four 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 [6]).

To establish confidence in our simulation infrastructure, we modeled the coupled IQ-ALU and front-end in an attempt to reproduce the results in [48]: we observed slowdowns of 68% and 41%, respectively, quite similar to the numbers reported in [48]. The new branch predictor algorithm (indexing with basic block start address

instead of basic block end address) affects accuracy by 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 noncritical and cause slowdowns of less than 5% (for a four cycle delay).

2.2.3 Comparison with Multithreaded Workloads

Figure 2.3 shows results for multithreaded workloads. Since the multithreaded workloads only include a subset of all programs, we normalize the multithread slowdowns against the single thread slowdowns observed for those programs. Hence, it is a reasonable approximation to directly compare the data in the two graphs of Figure 2.2 and Figure 2.3. For almost all loops, the multithreaded processor is slightly less sensitive to wire delays because it can find useful work in other threads during stall cycles. The only exception is the IQ-ALU loop. Wire delays in the IQ-ALU loop increase the load-hit speculation penalty. An increase in this delay causes the thread to issue more speculative instructions: hence wire delays are an

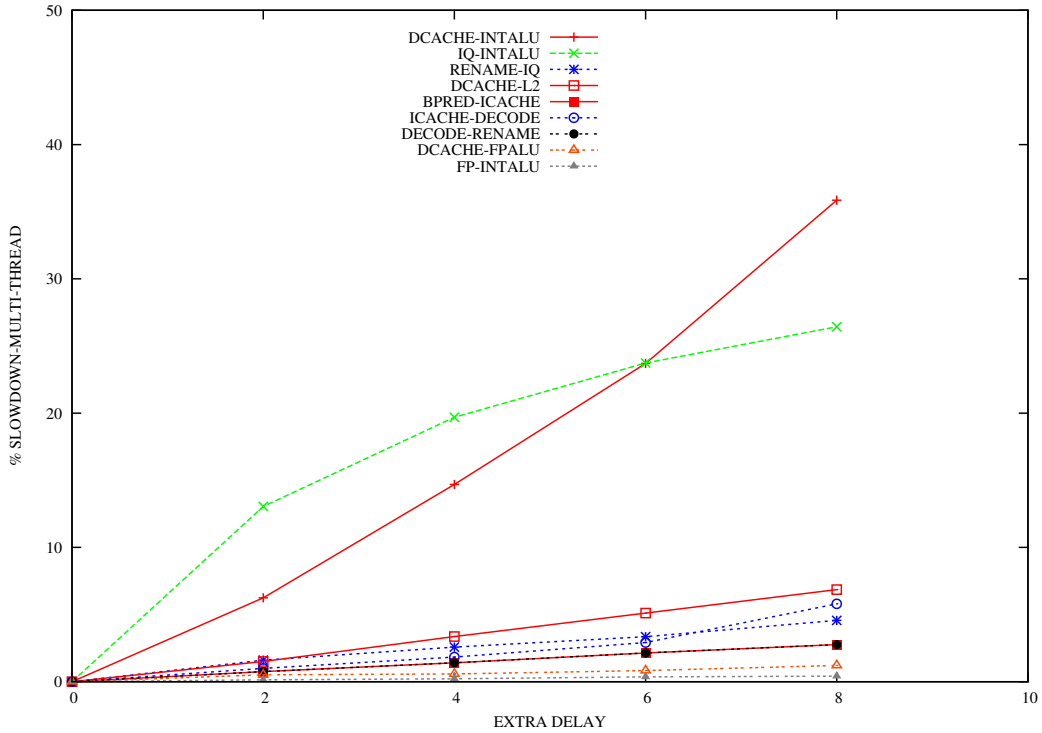

Figure 2.3. IPC slowdown curves for multithreaded workloads

impediment to the execution of the co-scheduled thread, not an opportunity to steal idle slots. Further, as this wire delay increases, issue queue occupancy increases and since this is a shared resource, it further inhibits the co-scheduled thread.

There are two reasons that could be attributed to this phenomenon. (i) The IQ-ALU loop is slightly different from the other loops. An increase in other delays causes a thread to sit idle allowing another thread to fill in the slack. For load-hit-speculation, increasing the loop length causes the thread to continue issueing dependents, so regardless of the presence of another thread, some penalty is being imposed. (ii) The IQ-ALU loop increases the Issue queue occupancy, which in turn could inhibit instructions from other threads from being issued.

2.2.4 Dependence of Criticality on Processor Configuration

Next, we examine various processor design points to verify if the criticality of wire delays is a function of the processor model and configuration.

We first consider a processor with strict in-order execution and also plot its slowdown. Figure 2.4 shows the corresponding curves. The parameters of the simulated processor are similar to the base out-of-order parameters of Table 2.2. We observe that the DCACHE-ALU loop stands out as the only critical loop with an average slowdown of 77% for all benchmarks for an eight cycle delay. Wire delay between the DCACHE and the ALUs adversely affects performance because not only do they affect instructions in the load's dependence chain, as in the case of the out-of-order pipeline, but they could possibly stall all instructions following the first consumer as they wait for it to proceed to the next pipe stage. Also, the ALU to IQ loop is not as critical relative to the out-of-order processor configuration. It experiences a slowdown of only 3.4% compared to 8% slowdown in performance for the OoO core. This is because, at any point in time the number of uncommitted in-flight instructions in the window is few compared to an out-of-order model. Hence, there is little pressure on the register file and causes no stalls related to register unavailability. All other loops experience an insignificant slowdown due to wire delays.

The properties for four out-of-order processors (ranging from "Poor" to "Su-

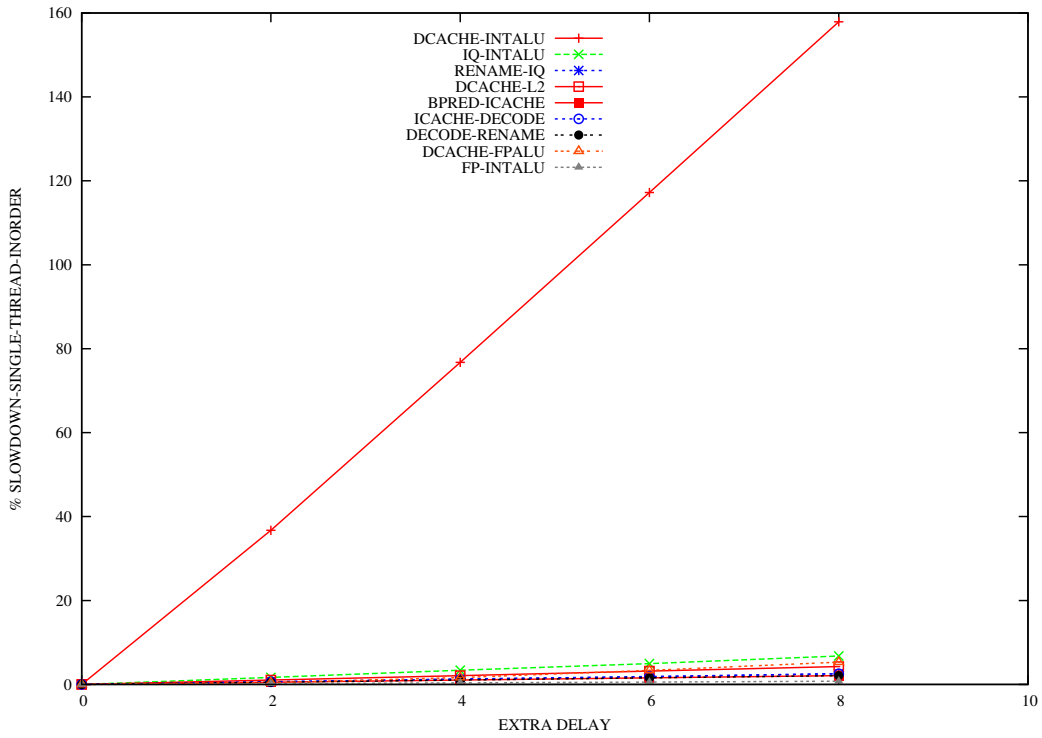

Figure 2.4. IPC slowdown curves for an in-order processor configuration

per") and the in-order configuration are summarized in Table 2.4. Figure 2.5 demonstrates the IPC degradation when eight cycles of wire delay 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. And as we saw, for the in-order model the ALU-Dcache delay stands out, yielding a 157% 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.

Table 2.4. Parameters for five different processor configurations

	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

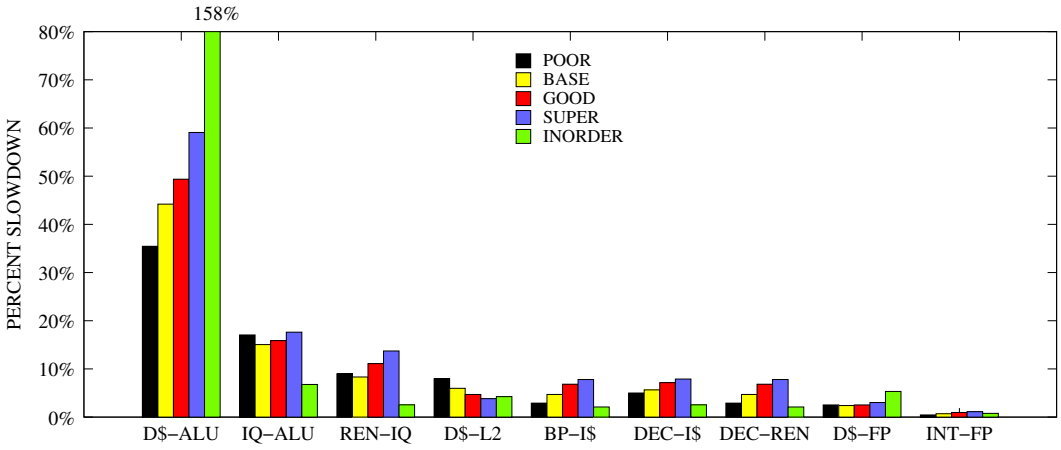

Figure 2.5. IPC slowdowns for various processor configurations

For our study in Chapter 3 and results in Section 3.3, we assume single threaded workloads and only consider the single threaded slow-down curves of Figure 2.2. However, a sensitivity analysis of our floorplanning algorithms to in-order and multithreaded workloads is presented in Section 3.3.3

CHAPTER 3

CRITICALITY AWARE FLOORPLANNING

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. An intelligent floorplanning algorithm should also strive to place blocks communicating over a critical path close to each other. The slowdown curves in Chapter 2 are indicative of the criticality of each interblock wire. In this section, we will discuss how a floorplanning algorithm can efficiently incorporate this data to produce floorplans tuned for performance.

3.1 IPC-aware 2D Floorplanning

Floorplanning algorithms [29, 38, 39, 6] typically employ a simulated annealing process to evaluate a wide range of candidate floorplans. The objective functions for these algorithms attempt to minimize some combination of silicon/metal area, wire power, and chip temperature. In modern microprocessors, since delays across global wires can exceed a single cycle, a floorplanning tool must also consider the performance impact of introducing multicycle wire delays between two communicating microarchitectural blocks. The objective function in a state-of-the-art floorplanner can be represented as follows [29, 38, 39, 6]:

$$\lambda_A \times Area\_metric \; + \; \lambda_T \times Temperature$$

$$+ \sum_{ij} \lambda_W \times W_{ij} \times Activity_{ij} \; + \; \sum_{ij} \lambda_I \times d_{ij} \times IPC\_penalty_{ij}$$

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.

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 predict the IPC of a configuration with arbitrary wire delays between pipeline stages. The predicted theoretical IPC slowdown (relative to the baseline processor with zero interblock wire delays) 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 2.2. If this hypothesis is true, detailed architectural simulations can be avoided for every floorplan that is considered. To verify this hypothesis, we simulated 10 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 3.1. Figure 3.1 compares the experimental IPC slowdowns against the theoretical slowdown computed according to the slopes of the curves in Figure 2.2. The maximum and average errors were 4% and 2.1%, respectively. This minor discrepancy is partially because the slowdown curve is being represented

Table 3.1. Critical path latencies for 10 random configurations

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

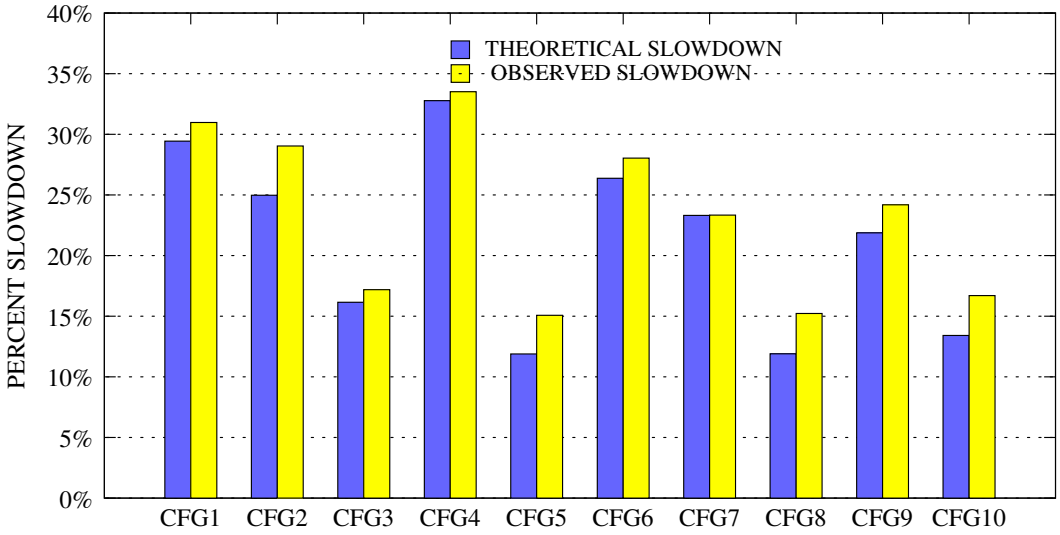

Figure 3.1. Additive nature of IPC degradation

as a straight line by a single slope value. We also repeated our experiments for out-of-order processor models with a range of resources and found little difference in the relative slopes of each slowdown curve.

By applying the floorplanning algorithm described, we are able to generate floorplans that reduce delays due to on-chip wires. Comparing such an IPC optimized floorplan with an unrealistic perfect floorplan with zero wire delays gives an estimate of the penalty due to wire delays in a processor. In the next section, we will see how much 3D layouts help in eliminating this penalty.

3.2 Optimizing 3D Floorplans

Three dimensional integrated circuits (3D ICs) [40, 41] offer an attractive opportunity to overcome the barriers of interconnect scaling. In a 3D circuit, several device layers are stacked together either in a face-to-face, face-to-back or back-to-back bonding. Vertical interconnects called as *interdie vias* or *d2d vias* are tunneled through silicon to provide direct communication paths between the various dies. An important benefit of 3D chips over a conventional 2D design is reduction of interconnect lengths. Other advantages are higher package density, smaller footprint, higher performance and lower interconnect power due to shorter overall wire lengths, and support for mixed technology chips.

The HotFloorplan [6] tool from Virginia is used to generate 2D floorplans. For each floorplan, the tool is allowed to move/rotate blocks and vary their aspect ratios, while attempting to minimize the objective function. We also extended the tool to generate 3D floorplans with a two phase approach similar to that described in [42]. The floorplan is represented as a series of *units/operands* (blocks in the floorplan) and *cuts/operators* (relative arrangement of blocks), referred to as a Normalized Polish Expression (NPE) [43]. Wong et al. [43] prove that a floorplan with n basic blocks can be represented as a unique NPE of size $2n - 1$. The balloting property of an NPE ensures that any point in the NPE there are at least as many units as the number of cuts and therefore allows for the reconstruction of a valid floorplan. The design space can be explored by applying the following three operations. As long as the balloting property holds, these operations will result in a valid floorplan: (i) swap adjacent operands, (ii) change the relative arrangement of blocks (i.e., complement the operators in NPE), and (iii) swap adjacent operator and operand. This is repeatedly performed as part of a simulated annealing process until a satisfactory value for the cost function is obtained.

For the 3D floorplan, the above basic algorithm is extended with additional moves proposed by Hung et al. [42] and is implemented as a two phase algorithm. In the first phase, two *move* functions are introduced in addition to the three described in [43] – interlayer move (move a block from one die to another) and

interlayer swap (swap two blocks between dies) – while still maintaining NPEs and satisfying the balloting property. The purpose of the first phase is two fold: (i) minimize the area footprint ($area_{tot}$) of both the layers and the difference in the areas of each layer ($area_{diff}$), and (ii) move the delay sensitive blocks between layers to reduce wire delays between them. The cost function used for this phase is (the equation parameters are clarified in Table 3.2):

$$cost_{phaseI} = \alpha_A \times area_{tot} + \alpha_{wl} \times \sum_i l_i.w_i + \alpha_d \times area_{diff}$$

The first phase results in two die floorplans having similar dimensions that serve as inputs to the second phase. In the second phase, no interdie moves or swaps are allowed. This phase tries to minimize (i) lateral heat dissipation among units, (ii) total power density of all pairs of overlapping units, (iii) wire delays among units, and (iv) total area of each die using the three basic within-die moves as described in [43]. The cost function used for this stage is:

$$cost_{phaseII} = \alpha_A \times area_{tot} + \alpha_{wl} \times \sum_i l_i.w_i + \alpha_d \times area_{diff} +$$
$$\alpha_{vh} \times \sum_{i,j} A_{olap}(i,j) \times (pd_i + pd_j) + \alpha_{lh} \times \sum_{i_1,i_2} sharedlen(i_1,i_2) \times (pd_{i_1} + pd_{i_2})$$

At the end of the second phase, we obtain floorplans for two layers with favorable thermal and wire delay properties. Finally, the L2 is wrapped around the two dies in a proportion that equalizes their area.

Table 3.2. 3D floorplanner cost function parameters

Parameter	**Description**	**Associated Weight**	**Value of Weight**
$area_{tot}$	Total area of both dies	α_{tot}	0.05
$area_{diff}$	Area difference between dies	α_{diff}	4e5
$\sum_i l_i.w_i$	Total Wire length/delay l_i - length of wire i w_i - number of bits being transferred on wire i	α_{wl}	0.4
$\sum_{i,j} A_{olap}(i,j) \times (pd_i + pd_j)$	Power density of overlapping units $A_{olap}(i,j)$ - overlapping area between units i and j, pd_i - power density of unit i	α_{vh}	0.5
$\sum_{i_1,i_2} sharedlen(i_1,i_2) \times (pd_{i_1} + pd_{i_2})$	Lateral heat dissipation factor $sharedlen(i_1,i_2)$ - shared length between units i_1 and i_2	α_{lh}	$5e-5$

3.3 Floorplanning Results

In this section we will discuss the methodology of our floorplanning algorithms and present detailed results for the optimal 2D and 3D floorplans obtained. Throughout this section we only deal with optimal single core floorplans and estimate the impact of wire delays within an uniprocessor core.

3.3.1 Methodology

HotSpot-3.0's grid model is used to determine the transient temperatures of the 2D and 3D layouts. The power traces to HotSpot are obtained using the Wattch power model for 90nm technology. Heat dissipation on interconnects is also modeled by attributing interconnect power to its connected units in the ratio of their areas, As discussed earlier, we also extend HotFloorplan [6] to evaluate 3D layouts and to include the $IPC\_penalty$ factor while generating optimal floorplans. The microprocessor model fed to HotFloorplan is very similar to the Alpha 21264 [32] – the same microarchitectural blocks and relative sizes are assumed. The slopes of the slowdown curves in Figure 2.2 are used to compute the $IPC\_penalty$ weights for each set of wires. These weights are listed in Table 3.3 along with the corresponding cycles required for each communication (for two different processor models).

To estimate the performance of each floorplan, we determine the distances between the centers of interacting blocks and compute wire latencies for two different types of wires – fast global wires on the 8X metal plane and semiglobal wires on the 4X plane. The processor is assumed to have four types of metal layers, 1X, 2X, 4X, and 8X, with the notation denoting the relative dimensions of minimum width wires [44]. 1X and 2X planes are used for local wiring within circuit blocks. These latencies are converted to cycles for two clock speed assumptions – 2 GHz and 4 GHz. The data in Table 3.3 show the corresponding cycles required for each communication in the most wire constrained model (wires are implemented on the slower 4X metal plane and a fast clock speed of 4 GHz is assumed) and the least wire constrained model (wires are implemented on the faster 8X plane and a clock speed of 2 GHz is assumed) for the optimal 2D floorplan. If the wire delay between blocks is less than 1 FO4, we assume that the delay can be somehow absorbed in

Table 3.3. Weights for the different pairs of blocks and the corresponding wire delays for the least constrained and most constrained models

Critical loop	Weight	Delay for optimal 2D floorplan 4X wires (4 GHz)/ 8X wires (2 GHz)	Delay for optimal 3D floorplan 4X wires (4 GHz) / 8X wires (2 GHz)
DCACHE-INTALU	18	1/1	0/0
DCACHE-FPALU	1	3/1	1/1
BPRED-ICACHE	2	1/1	1/1
IQ-INTALU	6	1/1	1/1
FP-INTALU	1	2/1	1/1
DECODE-RENAME	2	1/1	1/1
RENAME-IQ	4	1/1	1/1
DCACHE-L2	2	1/1	1/1
DECODE - ICACHE	2	2/1	1/1

the previous pipeline stage and no additional cycles of wire delay are introduced. The L2 latency is determined by adding the wire delay between the L1 cache and the nearest L2 bank to the 30 cycle L2 access time. This way, we are able to obtain optimal floorplans for both the 2D and 3D case and evaluate their performance relative to a baseline that has no wire delays.

The average power values for each microarchitectural block are derived from the Wattch power model [45] for 90 nm technology and this is used by HotFloorplan to estimate temperatures within each candidate floorplan. Wattch's default leakage model is employed, where a certain fraction of a structure's peak power is dissipated in every idle cycle. The leakage value is not a function of the operating temperature, thus under underestimating the power consumed by hot units. As we later show, even with this advantage, the hotter 3D architectures are unable to significantly outperform the cooler 2D architectures. Since we are preserving the 2D implementation for each circuit block and not folding them across multiple dies, Wattch's default power models for each block can be employed. HotFloorplan uses Hotspot-3.0's [46] grid model with a 50×50 grid resolution. Hotspot's default heat sink model and a starting ambient temperature of 45°C is assumed for all temperature experiments throughout the paper.

For 3D floorplans, each die is modeled as two layers – the active silicon and the bulk silicon. The dies are bonded face-to-face (F2F) and the heat sink is placed

below the bottom die. A layer of thermal interface material (TIM) is modeled between the bulk silicon of the bottom die and the heat spreader [47]. The thermal parameters for the various layers of the 3D chip are listed in Table 3.4. The power consumed by data wires between pipeline stages at 90 nm is also considered [48]. Hotspot does not consider interconnect power for thermal modeling. Hence, consistent with other recent evaluations [49], interconnect power is attributed to the units that they connect in proportion to their respective areas. Similar to the methodology in [27], the reduction in area footprint from 3D is assumed to cause a proportional reduction in clock distribution power.

3.3.2 Comparison of Optimal 2D and 3D Floorplans

Figure 3.2 and Figure 3.3 illustrate the optimal 2D and 3D floorplans derived from our methodology respectively. The 2D floorplan obtained is based only on weights derived from the single thread (OoO) slowdown curve. The performance of these floorplans along with that of a floorplan generated by a basic wire delay incognizant floorplanner is shown in Figure 3.4. The figure plots the average slowdown of SPEC-INT and SPEC-FP benchmarks with respect to a perfect floorplan with no wire delays for the most and least wire constrained models.

Table 3.4. Thermal model parameters

Bulk Si Thickness die1(next to heatsink)	750μm
Bulk Si Thickness die2 (stacked die)	20μm
Active Layer Thickness	1μm
Cu Metal Layer Thickness	12μm
D2D via Thickness	5μm
Si Resistivity	0.01 $(mK)/W$
Cu Resistivity	0.0833$(mK)/W$
D2D via Resistivity (accounts for air cavities and die to die interconnect density)	0.0166 $(mK)/W$
HotSpot Grid Resolution	50x50
Ambient temperature	45°C

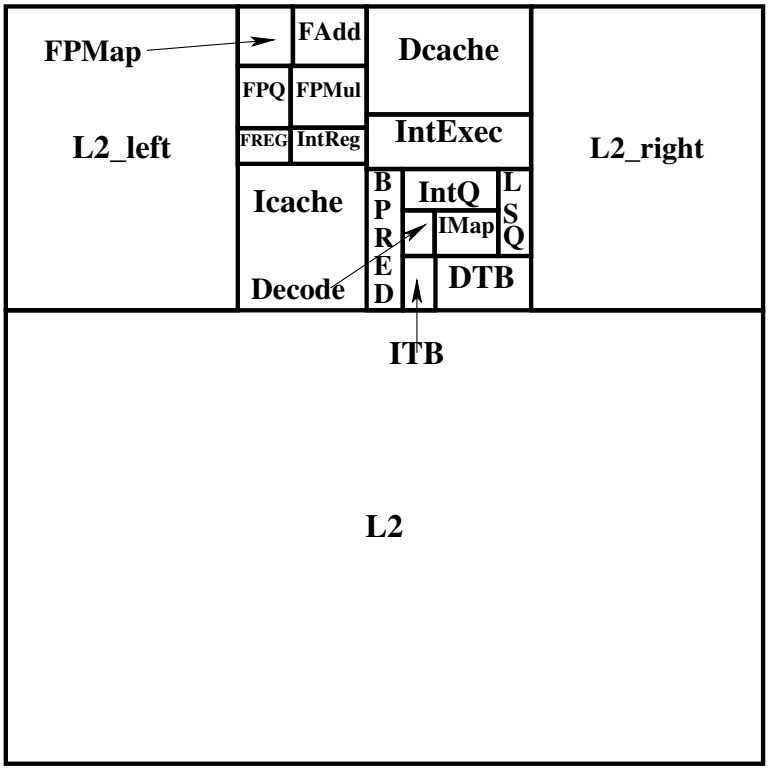

Figure 3.2. Optimal 2D floorplan

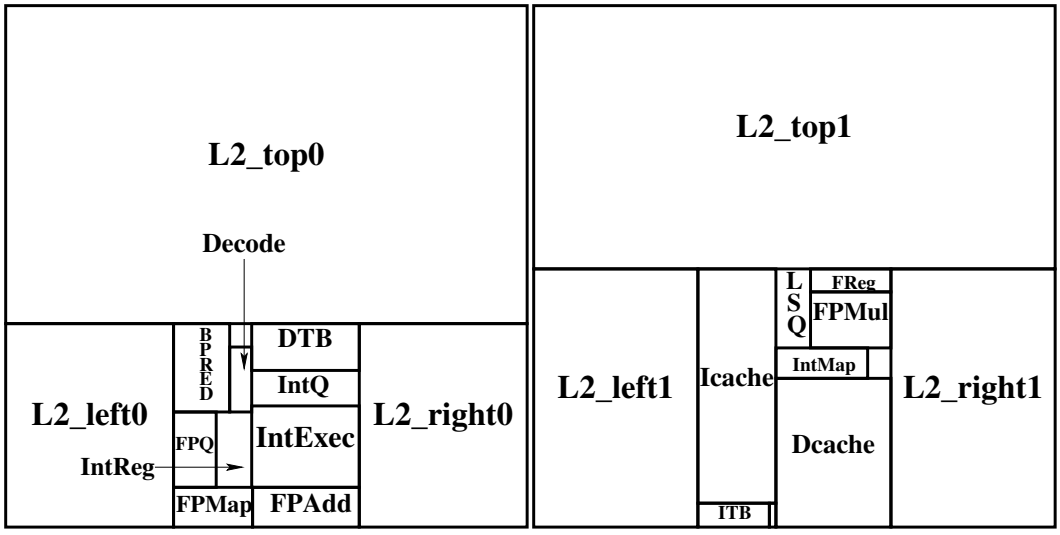

Figure 3.3. Optimal 3D floorplan

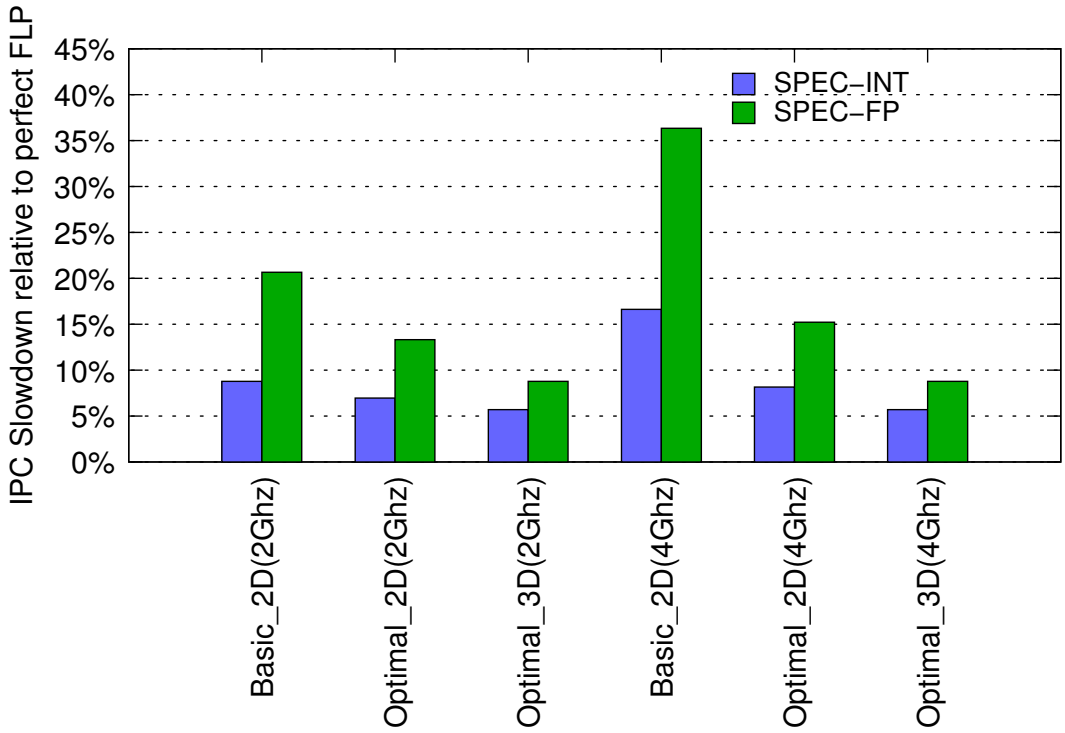

Figure 3.4. Comparison of basic, optimal 2D and optimal 3D floorplans

Based on these floorplans, it was observed that the optimal 2D floorplan enabled a performance improvement of 16.5% over the basic 2D floorplan. The IPC-aware 3D floorplan further achieves a performance improvement of 4%. Hence 3D stacking does not dramatically reduce critical latencies for an aggressive OoO processor.

We observe that the optimal 2D floorplan (shown in Figure 3.2) co-locates the units that are involved in the most critical wire delays (DCache-IntALU, IQ-IntALU). Because the critical wire delays are minimized, the IPC slowdown incurred by all the introduced wire delays is only 12% in the most wire constrained model and 10% in the least wire constrained model. It must be noted that this optimal 2D floorplan has a peak temperature that is only 5.5°C higher than a floorplan that is optimized for low temperature (produced by increasing the weight for the temperature term in the objective function). So it is not the case that wire delays are being reduced at the cost of high temperature. This result indicates that it is fairly easy to minimize wire delays between critical units even in two dimensions.

For the optimal 2D floorplan above, wire delays impose a performance penalty of 12% at most and this represents an upper bound on the performance improvement that 3D can provide. Figure 3.3 and Table 3.3 also show the optimal 3D floorplan and its corresponding communication latencies. The wire delays impose a performance penalty of 8% at most. Hence, for a traditional out-of-order superscalar processor, the stacking of microarchitectural structures in three dimensions enables a performance improvement of at most 4%.

According to our floorplan models, the peak temperatures for the 3D floorplan are on an average 12.7°C and 6.1°C higher than the 2D floorplan for the most and least wire constrained models, respectively (Table 3.5). We estimated the power dissipated by interblock wires in 2D and 3D based on the number of maximum bits being transferred between blocks, the distance traveled, power per unit length for 4X and 8X wires at 90 nm technology, and an activity factor of 0.5. In addition to the 50% reduction in clock distribution power, we observed a 39% reduction in power for interblock communication. We acknowledge that it is difficult to capture interblock control signals in such a quantification, so this is a rough estimate at best.

3.3.3 IPC-optimal Floorplanning for In-order and SMT Cores

In Sections 2.2.3 and 2.2.4 respectively, we saw that workloads for Simultaneous Multithreading cores and In-order cores have varied magnitudes of slowdown to wire delay. In this section, we perform sensitivity analysis of our floorplanning algorithm for In-order and SMT processors. Similar to our analysis for single threaded Out-of-order cores, we use the slowdown curves of Figure 2.4 and Figure 2.3 to

Table 3.5. Optimal floorplan temperatures in °C

Model	2D	3D	Difference
Most-constrained (Peak)	81.4	94.1	12.7
Least-constrained (Peak)	69.2	75.3	6.1
Most-constrained (Avg)	75.7	83.5	7.8
Least-constrained (Avg)	66.7	70.5	3.8

calculate the weights of each loop for In-order and SMT cores respectively. However, we restrict our analysis here to 2D floorplans alone. For a 100nm target process technology, we evaluate the performance of the benchmarks for a wire constrained model (Intermediate Wires/4Ghz Frequency) as well as a wire unconstrained model (Global Wires/2Ghz Frequency). For the In-order floorplan, we account for its simple structure by halving the areas of all units from the default Alpha 21264 floorplan.

Figure 3.5 shows the CPI of a basic floorplan and the IPC-optimal floorplan for an In-order core running the chosen benchmarks, normalized with respect to a perfect floorplan with no wire delays. For the least wire constrained model, the area of the in-order floorplan is small enough such that all interblock wires are within a cycle's reach for both the basic and the optimized floorplans. Both experience a slowdown of 20% with respect to the perfect floorplan. Our criticality aware floorplan experiences a slowdown of 23% on average compared to the perfect floorplan for the most wire constrained model. For the same cycle length and metal layer constraints, it outperforms the basic floorplan by 13%. This indicates that

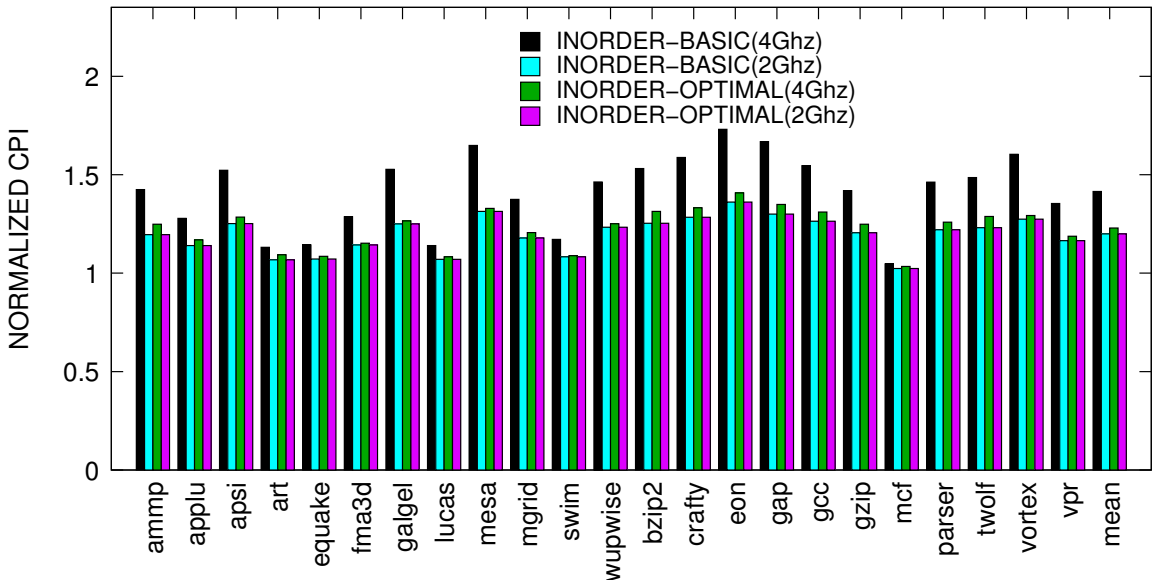

Figure 3.5. Normalized CPI of basic and criticality aware floorplans for in-order processors

even though the area of In-order cores are small, any delay in the DCACHE-ALU loop would hinder performance.

The normalized CPIs for an SMT processor simultaneously executing two SPEC benchmarks are shown in Figure 3.6. Once again our IPC-aware floorplanner outperforms a basic floorplanner by 6% and 17% respectively for the least and most wire constrained models. For the most wire constrained model, the average slowdown due to wire delays for the optimal floorplan is 14%.

3.3.4 Comparison with Pentium4 Study

Our conclusions differ from those drawn in the study by Black et al. [27]. The study characterizes the performance and power effect of 3D on an Intel Pentium4 implementation. In that work too, 3D is primarily exploited to reduce delays between microarchitectural structures (pipeline stages). Wire delay reduction in two parts of the pipeline contribute 3% and 4% IPC improvements and many other

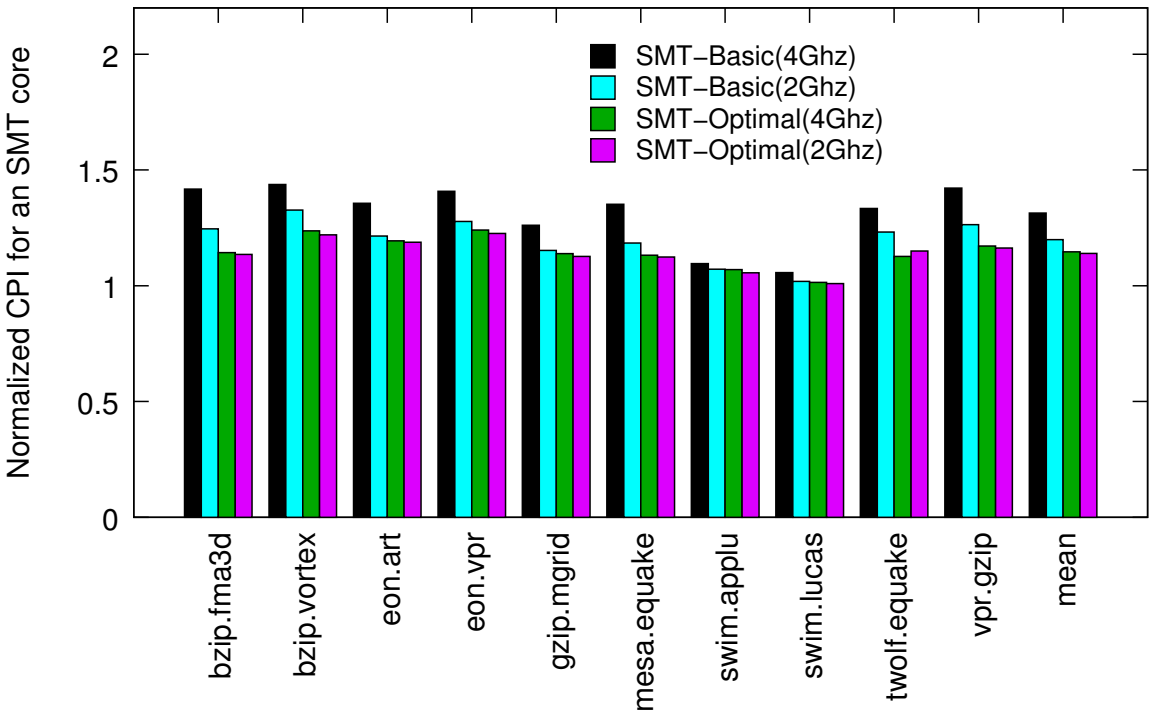

Figure 3.6. Normalized CPI of basic and wire delay aware floorplans for SMT processors

stages contribute improvements of about 1%. The combined effect is a 15% increase in IPC in moving to 3D. While that data serve as an excellent reference point, they are specific to the Pentium4 pipeline and the cause for performance improvement in each stage is not identified. We performed experiments to help fill gaps and provide more insight on the performance improvements possible by eliminating intracore wire delays.

That study reports a 15% performance improvement by implementing a Pentium4 core in 3D. This difference can be attributed to two sources:

1. **FP instruction latency:** The Pentium4 2D layout places the SIMD unit between the FP register file and FP ALUs. This introduces a 1-cycle delay between the FP register file and FP ALUs, modeled as a two cycle increase in the latency of all FP instructions. The move to 3D eliminates this wire delay and improves performance by 4%. If the latency of all FP instructions is reduced by two cycles in our simulation infrastructure, we too observe a similar 3% performance improvement (for the SPEC2k FP benchmarks). However, this aspect is left out of the results shown above. Firstly, as explained in Section 2.1, a delay between the register file and ALU should not introduce stall cycles between dependent instructions if full bypassing is provided. Secondly, the FP ALUs and FP register file are in proximity in our 2D layout. These observations do highlight the point that a 2D industrial implementation may have some inefficiencies (for example, inability to provide full bypassing or inability to co-locate certain structures) that can be elided with a 3D layout.

2. **Store life time:** The Pentium4 has a small store queue and is sensitive to postretirement pipeline stages involving the store instruction. By eliminating postretirement wire delay and releasing the store queue entry sooner, a 3% performance improvement is reported in [6]. In our simulation environment, if we implement a 14-entry store queue and release an entry 30 cycles after the store retires, we observe a 3% improvement if the postretirement delay

is reduced by 30%. This phenomenon disappears if the store queue size is eliminated as a bottleneck (by increasing its size to at least 24). We assume that the store queue size will not be a bottleneck and exclude this aspect from our simulation results.

Hence, we believe that these specific features of the Pentium4 may have contributed to greater improvements from 3D. These improvements are perhaps not indicative of what we can expect from other processors as we have shown. Overall, our study is more pessimistic about the potential of 3D because our pipeline model has balanced resources, is simpler and shorter (perhaps more indicative of future cores), and takes a rosy view of the efficiency of a 2D layout.

CHAPTER 4

COHERENCE MESSAGE CRITICALITY

The coherence operations on multicore architectures necessitate frequent communication over global on-chip wires. A closer look at the coherence messages can yield strategies to tolerate network delays and reduce significant power dissipation over the interconnects. Different coherence protocol messages have different delay tolerance levels, and result in varying impacts on overall processor performance. Our goal is to quantify the impact on performance when a coherence message gets delayed by an arbitrary number of cycles. A key benefit of such a study can be to determine if power optimization can be applied to parts of the network, or to specific messages.

4.1 Coherence Message Taxonomy

Directory based protocols were introduced to address the lack of scalability in snoopy bus based protocols. A cache miss at any node produces a coherence request which is sent to the home directory of the cache line. The home node consults its corresponding directory entry and replies with one of several coherence response messages. The different types of coherence messages in a MOESI based directory protocol are identified below.

- On a read/write cache miss, the requesting node sends either a *GETS* or *GETX* message to the home directory depending on its intention to modify the block. The home directory consults the entry for the block and either responds with *DATA* or *DATA_EXCLUSIVE* if there is no exclusive copy of the block elsewhere. In case there exists an exclusive copy elsewhere, the

GET request is forwarded to that owner. In our base protocol, the directory does not send a speculative reply back to the requester.

- The directory on receiving a read exclusive (GETX) request consults its list of sharers and advances *INV* (invalidate) requests to all the sharers. The sharers invalidate their cache lines and send *ACK* messages back to the requester.

- When an L2 replacement is encountered, the L2 cache sends a writeback request (PUT) message to the directory. There are two variations of the PUT message: *PUTX* and *PUTO* depending on whether an exclusive copy or a (shared) owner copy is being written. In case of an L1 replacement, in addition to the above messages the L1 can also send a *PUTS* message to the L2 on eviction of a shared block.

- When a PUT is received at the directory (or the L2), either a *WB_ACK* request or *WB_NACK* response is sent back. If a WB_ACK is received the requester responds with either a *Clean_Writeback* message (no data) or a *Dirty_Writeback* message (with data).

- The last set of control messages are the *Unblock* messages, which are used to inform either the L2 or a directory that it can commit its state transition. For example, on receiving a exclusive read request for a block in shared state, the directory sends invalidations and transitions to an intermediate state (say MM indicating "Blocked, going to modified"). The requester on receiving all the ACKs sends an Unblock message to the directory forcing it to commit its transition.

Given such a classification of coherence messages, we compare their latency needs, and estimate their sensitivity to link delays. Several factors, such as network load, topology, etc., may contribute to message latencies, but the physical delay in global wires, is most challenging to computer architects today. Hence, it is worth understanding how wire delays on an interconnection network can impact the overall performance of applications.

4.2 Motivation

Each of the coherence messages introduced in section 2.1 can have varying delay sensitivities. On an exclusive read request (GETX), the request for invalidations by the directory and the transmission of acknowledgments back to the requester necessitates a three hop transaction while the actual data is transmitted with just two hops. For such a scenario, the data are not in the critical path but the invalidations and acknowledgments are. However, such a scenario is specific to the MESI protocol, an assigned "Owner" node is responsible for providing the data block in the MOESI protocol. Moreover, bandwidth limitations may delay the propagation of data flits while smaller control messages are likely to be transmitted without much delay. There are other scenarios where DATA messages can become more critical, such as a node waiting on a read request(GETS) for a block in a shared state. In this case an extra delay in the delivery of the data might cause the load instruction that issued the GETS to stall for a prolonged period of time. Whether this is harmful or not to overall processor performance depends on whether the processor adopts a blocking in-order pipeline or an aggressive out-of-order engine which is perhaps more indicative of current and future generation of processors.

As another example, unblock control messages are critical for a directory to commit the state transition of a particular cache line so that it becomes available to service other requests. In order to avoid delaying the service of other requests to the same cache line, it is important that the unblock messages do not get delayed unduly. Moreover, the home directory often NACKs requests when it is busy making a transition. This could lead to more retries and introduce additional network traffic. Although, it should be noted that this would depend on the frequency of concurrent requests to the directory. If there are not many simultaneous requests for the same line, then perhaps "unblock" would represent a potentially noncritical message occupying a nontrivial portion of network bandwidth and power.

It is also possible that certain delays can instead be benign. Although we saw that invalidates are critical for a particular write request, it may reduce the lifetime of several other shared blocks resulting in more cache misses at their end. Hence,

deferred invalidates might increase reuse of shared blocks thereby increasing their utility. A performance improvement is also possible if a NACK sent by a busy directory gets delayed. If this happens often, delayed NACKs reduce the number of unsuccessful retries and wasted coherence traffic.

These are only a few examples where certain coherence messages appear more critical than others. However, their actual impact on performance can depend on several factors, such as the processor model, sharing pattern of applications, frequency of the coherence message, its interaction with other coherence operations, etc. Thus, detailed simulations are required to truly understand the overall performance and latency dependencies. Our work attempts to address this need by quantifying the tolerance levels that can be obtained by selectively changing the latency of specific coherence messages. For each cache coherence protocol, there exists a variety of coherence operations with different latency needs. Because of this diversity, there are many opportunities where results of such criticality analysis can play an important role. Rather than searching for specific optimizations as in [24], we only concern ourselves with automating the process of identifying critical and noncritical paths within the coherence network.

4.3 Making Use of Criticality

Criticality based control policies proposed in the past [50, 51, 52] require that the hardware evaluate criticality on a per instruction basis or aggregated over intervals during program execution. We propose to apply optimizations at a finer granularity by looking at criticality at a per coherence message basis. Provided that such capabilities are available, the following sections show some potential advantages of criticality based policies in a multiprocessor system. However, evaluating these techniques is beyond the scope of this thesis.

4.3.1 Resource Utilization

Resources (e.g., buffers, network bandwidth) can be better utilized by prioritizing allocations and accesses based on coherence message criticality. On-chip networks [53, 54] are replacing shared buses and dedicated wires as the *de facto*

interconnection fabric for chip multiprocessors. Packets compete for resources on a hop-by-hop basis while going through a complex router pipeline before traversing the output link at each intermediate node along their path. At each hop flits compete for buffer space, virtual channels and for switch allocation. For a head flit to proceed, at each stage the router has to take several decisions to allocate these resources. While allocating buffer space or virtual channel at the next router, the decision can be made to take into account the criticality of the messages to make sure the more critical messages do not get unduly starved. Kumar et al. [55] also propose a priority-based flow control mechanism called Express Virtual Channels that gives preference over resources to flits traveling long-distances. Message-criticality can also be extended to make EVC allocation-decisions at the EVC source/sink nodes.

4.3.2 Power Efficiency

Resources consuming power on less critical coherence traffic can often be made more power-efficient at the cost of performance. Cheng et al. [24] propose the use of heterogenous interconnects having varying latency, band-width and power-consumption properties by trading-off wire parameters such as wire-width and spacing, and repeater size and spacing. For instance, they observe that power-efficient wires can be implemented by reducing the repeater size and increasing its spacing on long global wires. However, the latency of such a wire is affected negatively. Banerjee et al. [48] show that a fivefold reduction in power can be achieved by trading-off a twofold increase in latency. Similarly, low-latency wires can also be implemented by increasing wire-width and spacing at the expense of decreased bandwidth. Hence, an intelligent router can be made to compute wire-mapping decisions that optimize for power and latency based on the criticality of the coherence message. Dally [56] first proposed the *Express Cube* topology which tries to reduce the average hop count that a coherence message takes to its destination. The idea is to use special express-nodes that have extra channels between nonadjacent nodes for long-distance traversals. Express nodes consume

more power than a local node as they have twice the number of ports, buffer space and a larger crossbar. Clock-gating is often applied to power-hungry components in order to save leakage power. Message criticality information can help decide if a particular express router needs to be woken up from its sleep state.

4.3.3 Misspeculation Reduction

Selectively applying speculation techniques based on coherence message criticality can reduce the effect and number of misspeculations. Using coherence prediction to accelerate less critical messages does not help improve overall performance even if the prediction is correct. There exists past work that uses prediction mechanisms to accelerate coherence protocols for example by guessing where the message will be sent or where the data will be used next. Kaxiras et al. [57] produce a taxonomy of past shared-memory coherence prediction schemes. The Cosmos [58] coherence message predictor predicts the source and type of the next coherence message to a cache block. Chang et al. [59] speculatively initiate coherence messages (e.g. invalidates or upgrades) in order to accelerate shared accesses. Lebeck et al. propose Dynamic Self-invalidation [60] in which processors speculatively invalidate their own blocks to speed-up future invalidations. With such speculation on coherence protocols, a misspeculation might cause unnecessary re-tries, invalidations and wasted coherence traffic. Avoiding speculation for noncritical messages would help avoid these undesirable consequences.

4.4 Coherence Message Analysis

4.4.1 Methodology

Virtutech Simics [61] is a full system functional simulator that accurately models the SPARC architecture. In conjunction with Simics we use the Wisconsin GEMS-1.4 [62] timing-infrastructure to model a four core chip multiprocessor with out-of-order issue and directory-based cache coherence. Each processor has a private L1 and all cores share a multi-bank noninclusive L2. The L1 caches, L2 banks and their corresponding directories are inter-connected with a hierarchical switch topology.

The Ruby module in GEMS provides a detailed memory timing-model that implements the MOESI directory-based cache coherence protocol with block migration. It is configured to model the timing of user-mode data only and ignore supervisor-mode instructions in order to filter out the effect of long-latency page table misses and page faults. In order to realize our goal of quantifying cache-coherence message criticality, we modify Ruby to allow network switches to identify coherence messages based on their type. We map each type of message, as classified by Section 4.1, onto links with some specified extra cycles of latency. Switches are also modified to allow out-of-order delivery of coherence messages in order to accommodate the nonuniform latency of links. We also model a directory cache in order to minimize directory response time. Table 4.1 lists the different coherence message groupings along with the constituent messages evaluated for this analysis.

Opal is an out-of-order timing-first processor simulator module that drives Ruby. It runs ahead of Simics' in-order functional simulation and determines when and what instructions should be fetched, decoded and executed. Opal consults Simics during its instruction retirement stage and advances Simics by one instruction should they agree on the correctness of the execution. This checking for correctness is necessary as Opal employs a relaxed consistency model and does not guarantee sequentially consistent execution by itself. On the event that Opal and Simics disagree, Opal replays the offending instructions. The Opal and Ruby simulation parameters used for the analysis are shown in Table 4.2 and Table 4.3, respectively.

Table 4.1. Coherence message groupings

Message Group	**Constituent Messages**
GETS	Shared Read Requests
GETX	Exclusive Read Requests
PUT	Writeback Requests (PUTS, PUTX, PUTO)
INV	Invalidate Requests
ACK	Acknowledge Replies
DATA	Shared DATA Replies
DATAX	Exclusive DATA Replies
WBDATA	Clean/Dirty Writeback Data Replies

Table 4.2. Opal parameters

Parameter	Value
System configuration	1 chip, 4 cores per chip
Clock frequency	2GHz
Fetch/decode/execute width	4/4/4
Pipeline stages	11
ROB/IWin size	80/64
BPred	1KB YAGS,128-entry cascaded indirect,64-entry RAS

Table 4.3. Ruby parameters

Parameter	Value
Cache block size	32 Bytes
Split L1 I & D cache	32KB each, 2-way
Shared L2 size	2MB/8-way
L2 configuration	4-banks/noninclusive NUCA
N/w topology	Hierarchical switch, 4 VCs per switch
Link bandwidth	10 bytes/cycle
Baseline link latency	1 cycle (one-way)
Mem controller/dir-cache latency	40/6 cycles

4.4.2 Workload Description

For our evaluation, we use a workload consisting of a self-built kernel-like synthetic macro-benchmark that accepts configurable parameters. The benchmark is compiled on Solaris with optimizations turned off. In essence, the benchmark simulates simultaneous random accesses by multiple threads to a shared global data structure without using any locking primitives. By not employing any locking primitives there will be increased contention for shared blocks and higher coherence traffic. Evaluating the performance impact for such a workload for an 8x increase in base link latency will reflect the most pessimistic effect of wire delays within coherence paths.

We provide the ability to tune the characteristics of the synthetic benchmark by having it accept input parameters similar to the SPLASH2 kernel workloads. The size of the global data structure, the number of threads, the total number of accesses in the primary random access stage, the percentage of writes among those memory accesses and the level of ILP available in the main loop are all configurable. Table 4.4 describes the input parameters to the program along with their default

Table 4.4. Synthetic benchmark input parameters

Parameter	Description
N [Default = 1024]	Even integer that determines the total number of data pts (N*N)
P [Default = 4]	Number of processors/threads
I [Default = 256K]	Number of iterations of the main loop
W [Default = 50%]	Percentage of writes in the main loop
L [Default = 0]	Level of ILP available per iteration. Each level corresponds to 2-3 additional FLOPs per iteration

values. A FLOP is a complex FP computation (in C code) that breaks down to multiple FP operations at the machine level. For each level of ILP, one dummy FLOP is added to the main loop. The benchmark characteristics for the default input parameters are listed in Table 4.5. The main loop refers to the compute loop within the macro-benchmark for which statistics are tracked. Total loads/stores also includes memory operations that occur outside the main loop, such as memory initialization and cache warm-up. The Appendix contains the pseudo-code for the synthetic benchmark.

In order to avoid cold start misses and to randomly distribute ownership of cache blocks, each thread touches an equal share of the global memory after initialization. Profiling of execution statistics begins after this warm-up stage. In every iteration of the main loop, each thread also reads a local per-thread data structure that is initialized with indices for the shared random access. This engenders competition for cache space between local and shared data. We choose to perform most of our evaluations with such a benchmark in order to circumvent the prohibitively long simulation periods encountered with Opal, without losing credibility of the results. Moreover, we are better able to understand the interaction between wire-delays in coherence paths and factors such as write set size and available ILP.

Table 4.5. Breakdown of instructions and memory operations per CPU

Total Instructions	884 M	Instructions within main loop	122 M
Total Loads	100 M	Loads within main loop	38 M
Total Stores	41 M	Stores within main loop	7 M

4.4.3 Performance Impact of Wire Delays on Coherence Protocols

To obtain an insight into the delay sensitivity of coherence messages we quantify the slowdown in performance of the benchmark when a specified number of cycles are added to coherence messages of each target group. This helps us clearly distinguish their criticality, and determine the level of tolerance each message can provide to network delays. Figure 4.1 shows the normalized execution time relative to a baseline of the macro-benchmark when 4, 8 and 16 cycles of additional link latency are added to each target message group.

In general we observe that, with the exception of GETS, most control messages are tolerant to network delay. Read requests are affected the most causing an average slowdown of 36% for eight cycles of extra delay. This is primarily because each L1 miss for a read request will trigger the delay and since the probability of an instruction waiting on a load instruction is near 100%, delays on GETS messages critically impact performance. Table 4.6 shows the percentage of bandwidth consumed by each type of coherence message. Given that GETS messages only occupy 5% of the consumed bandwidth, an efficient strategy might be to map them on to very low-latency wires perhaps at the cost of increased area and power

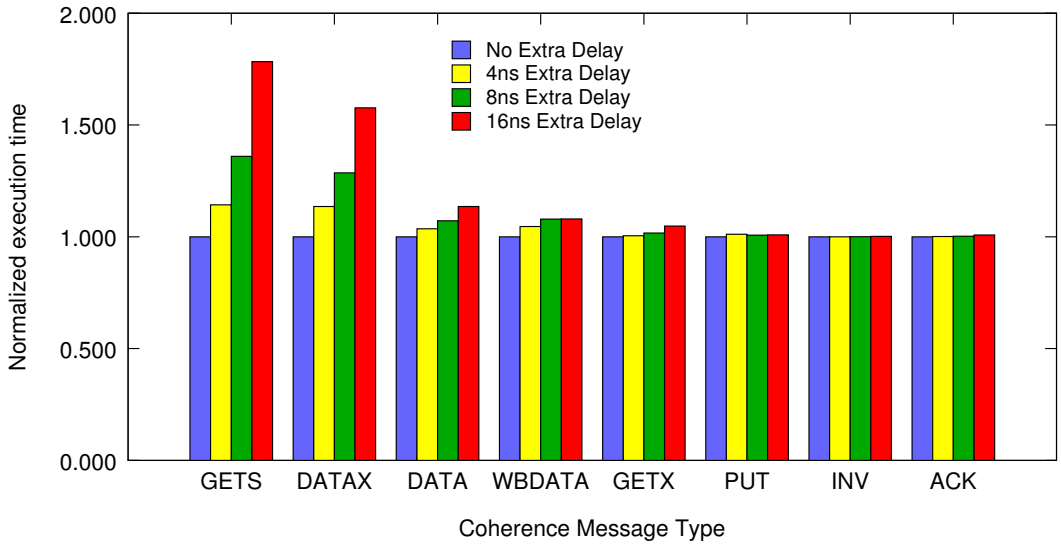

Figure 4.1. Execution time impact of wire delay on the synthetic benchmark

Table 4.6. Bandwidth consumption of different message types

ACK :0.12%	DATA :10.45%	DATAX :36.88%	GETS :4.97%	WBDATA :28.54%
INV :0.01%	PUT :7.61%	WBACK :6.04%	GETX :0.37%	OTHER :4.99%

dissipation. The second most critical kind of messages are Exclusive Data messages that encounter a degradation of 29% for an eight cycle overhead in link latency. They also consume the most bandwidth, this is perhaps indicative of the fact that only a minority of the read requests are made for blocks that exist in shared state in other caches at the time of request. The only other message types that offer any kind of hindrance to performance are DATA and WBDATA which cause 7% and 8% slowdown, respectively. All other messages are noncritical and at most cause a 2% reduction in performance.

4.4.4 Impact of Write-set Size and Available ILP

We evaluate different workloads by varying the write-set size of the macro-benchmark in order to see whether it alters the relative order of criticality of the coherence messages. Figure 4.2 shows the impact of the write set size on the execution time of the main loop. We observe that an increased write-set size has almost a linear effect on execution time even with no additional delays. On the baseline interconnect network which has no additional delays, the workload with a 100% write-set size takes 26% more time to execute compared to a write-set size of 0%. This is because fetching the permissions for writes in general consumes more time compared to reads, as there is a compulsory overhead of both internal and external acknowledgments for each access. Also since Opal by default employs a relaxed memory consistency model that relies on the use of locks, an increase in the write-set size will cause more sequential consistency violations on average. The resulting instruction replays could also have contributed to the increased execution time. However, the relative impact of coherence messages does not change by much, GETS and DATAX remain the most critical message groups regardless of the write-set size.

Figure 4.3 shows the deviation in slowdowns corresponding to two workloads

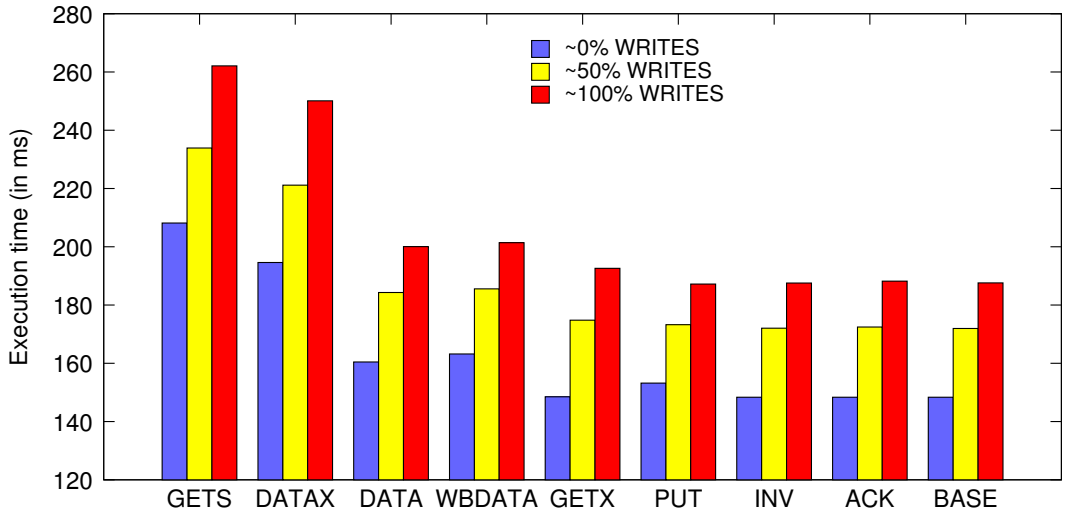

Figure 4.2. Variation with different write-set sizes

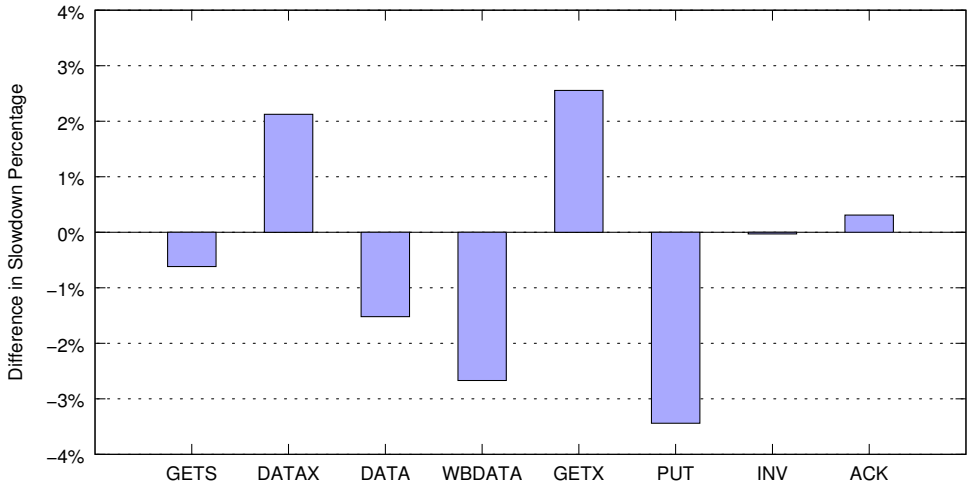

Figure 4.3. Difference in slowdown between 0% and 100% write-set size

with 0% (*workload-w0*) and 100% (*workload-w100*) write-set sizes, respectively. There is only a 1.7% average deviation in slowdowns of all coherence messages corresponding to a 100% increase in write-set size. The maximum deviation is for PUT for which *workload-w0* incurs 3.4% additional performance penalty compared to *workload-w100*. In general, write-backs become more critical as the percentage of loads increases. WBDATA, and PUT messages are issued during a write-back transaction and together contribute to an aggregate slowdown that is 6.1% more

for *workload-w0* than *workload-w100*. DATAX and GETX only become marginally more critical with increased write-set size. This perhaps can be attributed to the fact that for the synthetic workload writes rarely occur on the critical path. Since the benchmark represents a random access pattern, it lacks both spatial and temporal locality.

The synthetic benchmark is tailored to be memory-intensive with minimum available ILP. However, slightly more computation-intensive workloads can better hide latency in coherence paths with other coherence independent instructions. Figure 4.4 illustrates the effect of available ILP on coherence protocol performance. The performance of workloads with a specified ILP level of 0 (*workload-l0*), 4 (*workload-l4*) and 8 (*workload-l8*) additional floating point computations are plotted in the figure for a link overhead of eight cycles for each target message group. Our baseline workload with no additional computations has an aggregate IPC of 1.05, while *workload-l4* and *workload-l8* have IPCs of 3.02 and 3.41, respectively. In Section 4.4.3 we saw that GETS and DATAX are most delay sensitive. The availability of more ILP is proportionally able to offset 13% and 12% of the slowdown for these messages with an addition of eight floating point computations per iteration.

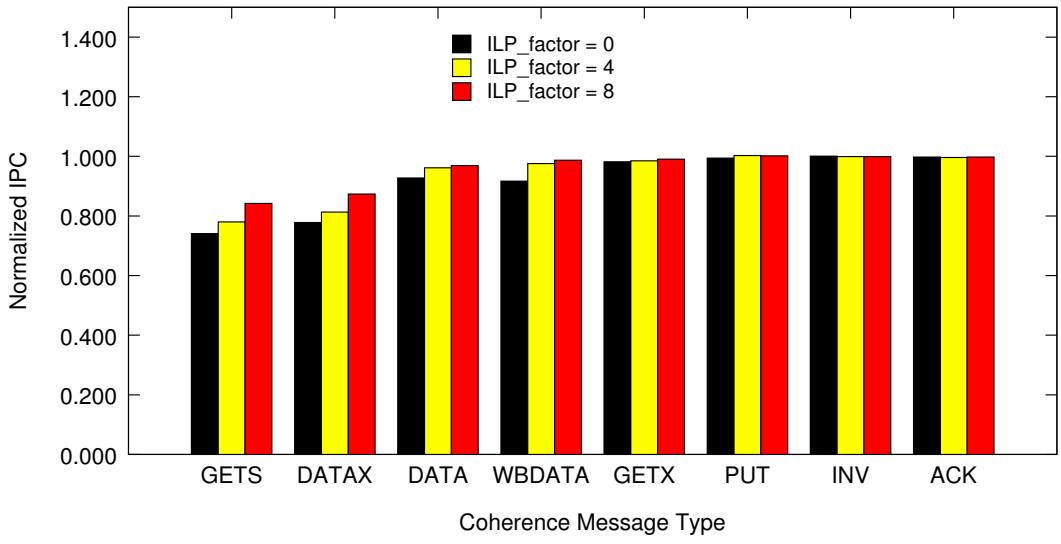

Figure 4.4. Performance impact under different levels of available ILP

4.4.5 Memory Consistency Model

Memory consistency models also play a role in determining the ability of a processor system to absorb delays. Stricter consistency models have constraints on the amount of parallelism that can be exploited in memory requests. More relaxed consistency models are more optimistic about the correctness of operations and overlap memory requests often, such models would be better able to hide latency in servicing the requests. However, some form of additional book-keeping or programming constructs are normally required to ensure correctness. We evaluate two models of consistency in this section: Sequential Consistency and an Alpha-like Relaxed Consistency model. As mentioned earlier Opal by default models the latter: memory operations are allowed to execute out of order and only load-store dependences are checked in the LSQ. In order to model a more strict sequentially consistent memory model, we modify Opal to execute memory operations in order but allow flexibility in reordering independent nonmemory operations.

Figure 4.5 shows the performance slowdown observed for 4, 8 and 16 cycles of additional link latency for both consistency models under consideration. As surmised, out of order execution when coupled with a relaxed consistency model provide further tolerance to wire delays. For an eight cycle overhead in network link latency, compared to a sequential consistency model, a relaxed consistency model is able to absorb 18% and 21% of the total slowdown for the two most critical messages: GETS and DATAX.

4.4.6 Case Study: SPLASH-2

Figure 4.6 shows the percentage slowdown for four applications: *barnes, fmm, ocean, water* and three kernel programs: *cholesky, fft* and *lu* from the Splash-2 [63] suite of parallel benchmarks for eight cycles of extra link latency. The figure also shows the mean slowdown for the chosen programs.

Execution statistics are tracked for the parallel sections of the benchmark after an initial warm-up period of 1 million instructions. All programs are compiled for a processor with four cores and simulations are run till the end of the parallel section or until any of the four cores commits 100 million instructions.

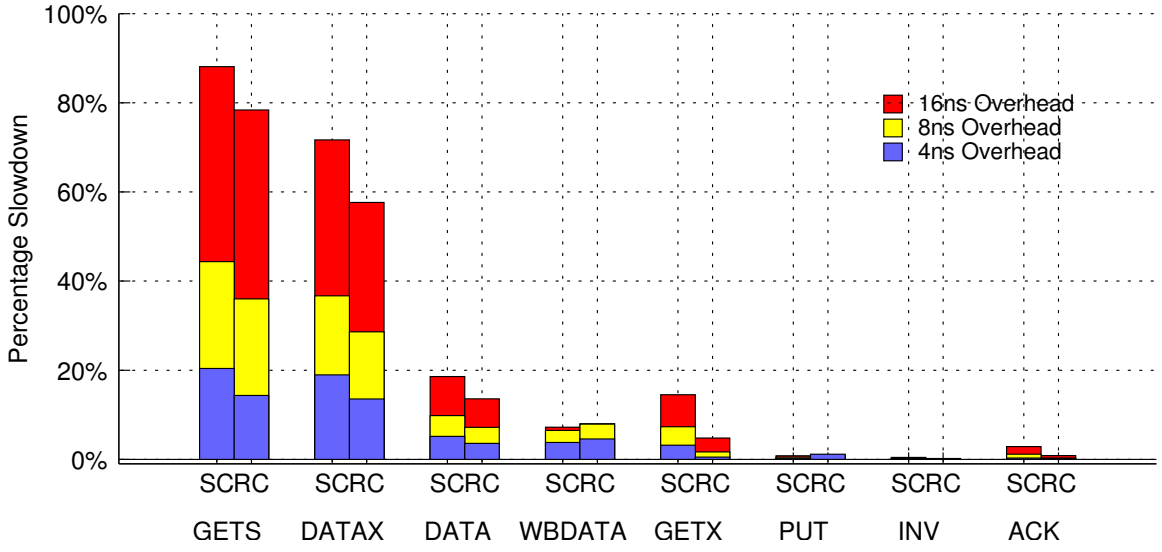

Figure 4.5. Slowdown percentage for sequential and relaxed consistency models

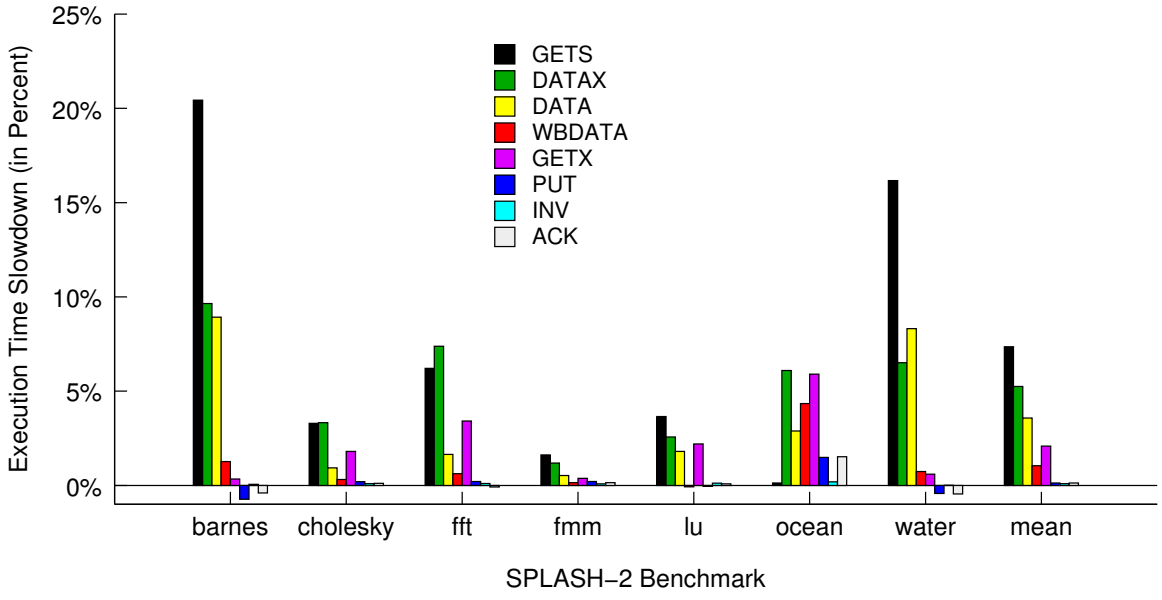

Figure 4.6. Percentage slowdown for SPLASH-2 benchmarks

GETS has the highest mean slowdown of 7.4% for all benchmarks, followed by DATAX and DATA with slowdowns of 5.3% and 3.6%, respectively. These numbers in general agree with our previous results based on our synthetic macro-benchmark which argue that read requests and data occur the most on the critical path. However, the magnitude of the slowdowns are low compared to our synthetic benchmark as they are likely to have sufficient ILP and spatial/temporal locality in their cache accesses. A few benchmarks such as *ocean* and *fft* experience a nontrivial slowdown for GETX also, which may be due to the fact that these benchmarks are more cooperative in their operation than our random workload. With such an access pattern, a value written to a shared memory location has a high probability of being read by another thread. Hence GETX often occurs on the critical path.

CHAPTER 5

RELATED WORK

Numerous automated floorplanning tools and algorithms exist in the literature [64, 65, 38, 66, 67, 39, 33, 6]. The objective function for some of these tools is purely performance [64], while for others it is some combination of temperature and wire communication [65, 38, 67, 39, 33, 6]. To date, there is no architectural evaluation that comprehensively quantifies the effect of wire delays between various micro-architectural blocks. Some of these data can be found scattered in the literature. For example, Sprangle et al. [30] quantify the effect of increasing the number of cycles for ALU bypass, L1/L2 cache access, and branch mispredict penalty. Borch et al. [28] quantify the effect of increasing the number of cycles in the branch prediction and load-hit speculation loops.

The MEVA floorplanner [29, 4] adopts the methodology of Sprangle et al. [30]. Other floorplanning tools [5, 33, 6] over estimate the IPC effect of wire delays because they do not consider simple pipeline optimizations. For example, (i) Long et al. [33] report IPC differences of up to 60% for various floorplans, (ii) Sankaranarayanan et al. [6] 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. We show interblock wire delays are not as critical as they have been made out to be.

Early stage computer architecture results for 3D technology have appeared in the last two years (examples: [27, 29, 68, 69, 9, 47, 7]). A study by Loi et al. [69] evaluate an architecture where cache and DRAM are stacked upon a planar CPU implementation. Li et al. [68] quantify the effect of the 3D stacking approach on a chip multiprocessor and thread level parallelism. This work focuses on the effect

of 3D stacking on a single core and instruction level parallelism. A recent paper by Black et al. [27] evaluates stacking for a Pentium4 processor implementation. However, that evaluation does not provide the details necessary to understand if the stated approach is profitable for other processor models. We attempt to address that gap in Section 3.3.4. Li et al. [68] employ a 3D on-chip network to connect cache banks and cores in a CMP. We have also done a comparison of our results with that of [27]. Cong et al. [29] quantify the impact of a few critical loops on performance and 2D and 3D layouts, but we present the most comprehensive analysis to date of the effect of wire delays on critical processor loops.

Cheng and Muralimanohar et al. [24] propose the use of heterogeneous interconnects for cache coherence traffic and large L2 caches [70]. They map wires of varying latency and power dissipation characteristics to coherence protocol messages by examining their bandwidth and latency needs in a directory based cache coherence system. However, their study does not include a detailed characterization of the impact of coherence message delays on performance. Lebeck et al. [71] extend the uniprocessor DAG (Directed acyclic graph) model to quantify instruction criticality and slack in shared memory multiprocessor systems and study how the choice of coherence protocols may affect the slack distribution. To our knowledge, no other work characterizes the impact of wire delays on cache coherence message paths for chip multiprocessor systems.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis presents a methodology to determine the effect of interblock 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. An IPC-optimized floorplan can outperform a basic floorplanner by up to 17% on average, while incurring a minor increase in peak temperature. A floorplanning algorithm that can exploit the reduced signal distances provided by 3D bonding can further improve performance by 4% albeit with a nontrivial increase in temperature.

We also perform a wire delay study in the context of coherence protocols for a directory based shared memory multiprocessor system. We show that read requests, exclusive data and shared data occur the most on the critical path causing slowdowns up to 20% for a set of SPLASH-2 programs. However, we find that for an out-of-order processor employing a fairly relaxed memory consistency model, most other wire delays are noncritical and at most cause an average slowdown of 2%. This presents the opportunity to perform aggressive interconnect power optimizations and priority based resource allocations.

The key contributions of the work can be summarized as follows:

- We present the most comprehensive analysis of the impact of wire delays on critical processor loops (including various pipeline optimizations and SMT

cores).

- Based on the wire delay study we present two IPC aware floorplanning algorithms for 2D and 3D integrated circuits.
- This work is the first to integrate many varied aspects (loop analysis, automated floorplanning, etc.) in determining the benefit of 3D for single core performance.
- This work also identifies the impact of wire delay on coherence messages for a shared memory out-of-order CMP.

Recently transactional memory systems are showing great promise as the preferred programming approach for effectively using the multithreaded environment offered by future chips with multiple cores. Our future work will attempt to recognize the criticality of coherence messages within a hardware log based transactional memory system, such as LogTM. It will also be worthwhile quantifying the benefits of the proposed techniques (section 4.3) that exploit coherence message criticality while exploring other applications.

APPENDIX

SYNTHETIC BENCHMARK PSEUDOCODE

```
SYNTHETIC_BENCH(N,P,I,W,L)
   gl_array[N][N] <- init_global_matrix(N,N,double)
   thread_create(P, SLAVE())
   thread_wait()
   free_global_matrix()
   return

SLAVE()
   //init indices for warm-up
   lo_warm_index[N*N/P][2] <- init_random_indices(N)
   //init indices for compute
   lo_main_index[I][2] <- init_random_indices(N)
   //init write flags for compute
   lo_main_write_flag[I] <- init_random_flag(W)
   //warm-up
   for i <- 1 to N*N/P
      x <- lo_warm_index[i][0]
      y <- lo_warm_index[i][1]
      read(gl_array[x][y])
   barrier
   //compute
   for i <- 1 to I
```

```
   x <- lo_main_index[i][0]
   y <- lo_main_index[i][1]
if lo_main_write_flag[i]
   write(gl_array[x][y])
else
   read(gl_array[x][y])
for j <- 1 to L
   dummy_flop()
return
```

REFERENCES

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

[2] 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`).

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

[4] 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.

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

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

[7] Y. Xie, G. Loh, B. Black, and K. Bernstein, "Design Space Exploration for 3D Architectures," *ACM Journal of Emerging Technologies in Computing Systems*, vol. 2(2), pp. 65–103, April 2006.

[8] J. Rattner, "Predicting the Future," 2005. Keynote at Intel Developer Forum.

[9] S. Mysore, B. Agrawal, N. Srivastava, S. Lin, K. Banerjee, and T. Sherwood, "Introspective 3D Chips," in *Proceedings of ASPLOS-XII*, October 2006.

[10] K. Puttaswamy and G. Loh, "Implementing Caches in a 3D Technology for High Performance Processors," in *Proceedings of ICCD*, October 2005.

[11] K. Puttaswamy and G. Loh, "Dynamic Instruction Schedulers in a 3-Dimensional Integration Technology," in *Proceedings of GLSVLSI*, April 2006.

[12] K. Puttaswamy and G. Loh, "Implementing Register Files for High-Performance Microprocessors in a Die-Stacked (3D) Technology," in *Proceedings of ISVLSI*, March 2006.

[13] K. Puttaswamy and G. Loh, "The Impact of 3-Dimensional Integration on the Design of Arithmetic Units," in *Proceedings of ISCAS*, May 2006.

[14] P. Reed, G. Yeung, and B. Black, "Design Aspects of a Microprocessor Data Cache using 3D Die Interconnect Technology," in *Proceedings of International Conference on Integrated Circuit Design and Technology*, May 2005.

[15] Y.-F. Tsai, Y. Xie, N. Vijaykrishnan, and M. Irwin, "Three-Dimensional Cache Design Using 3DCacti," in *Proceedings of ICCD*, October 2005.

[16] Corporate Institute of Electrical and Electronics Engineers, Inc. Staff, *IEEE Standard for Scalable Coherent Interface, Science: IEEE Std. 1596-1992*. 1993.

[17] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of ISCA-24*, pp. 241–251, June 1997.

[18] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato, "The Use of Prediction for Accelerating Upgrade Misses in CC-NUMA Multiprocessors," in *Proceedings of PACT-11*, 2002.

[19] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood, "Multicast Snooping: A New Coherence Method Using a Multicast Address Network," *SIGARCH Comput. Archit. News*, pp. 294–304, 1999.

[20] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence Decoupling: Making Use of Incoherence," in *Proceedings of ASPLOS-XI*, pp. 97–106, 2004.

[21] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM," in *Proceedings of ISCA-26*, 1999.

[22] K. M. Lepak and M. H. Lipasti, "Temporally Silent Stores," in *Proceedings of ASPLOS-X*, pp. 30–41, 2002.

[23] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *Proceedings of ISCA-27*, pp. 248–259, June 2000.

[24] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter, "Interconnect-Aware Coherence Protocols for Chip Multiprocessors," in *Proceedings of ISCA-33*, June 2006.

[25] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. Carter, "Wire Management for Coherence Traffic in Chip Multiprocessors," in *Proceedings of the 6th Workshop on Complexity-Effective Design, held in conjunction with ISCA-32*, June 2005.

[26] 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*, vol. Q1, 2001.

[27] B. Black, M. Annavaram, E. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *Proceedings of MICRO-39*, December 2006.

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

[29] 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.

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

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

[32] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, March/April 1999.

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

[34] 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.

[35] D. Burger and T. Austin, "The Simplescalar Toolset, Version 2.0," Tech. Rep. TR-97-1342, University of Wisconsin-Madison, June 1997.

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

[37] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proceedings of ISCA-23*, May 1996.

[38] 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.

[39] 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.

[40] T.-Y. Chiang, S. Souri, C. Chui, and K. Saraswat, "Thermal Analysis of Heterogeneous 3-D ICs with Various Integration Scenarios," in *Proceedings of IEEE IEDM*, 2001.

[41] S. Das, A. Chandrakasan, and R. Reif, "Timing, Energy, and Thermal Performance of Three-Dimensional Integrated Circuits," in *Proceedings of GLSVLSI*, 2004.

[42] W. Hung, G. Link, Y. Xie, N. Vijaykrishnan, and M. Irwin, "Interconnect and Thermal-Aware Floorplanning for 3D Microprocessors," in *Proceedings of ISQED*, March 2006.

[43] D.F.Wong and C.L.Liu, "A new algorithm for floorplan design," in *Proceedings of the 23rd ACM/IEEE conference on Design automation*, pp. 101–107, 1986.

[44] R. Kumar, V. Zyuban, and D. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads, and Scaling," in *Proceedings of the 32nd ISCA*, June 2005.

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

[46] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, and K. Sankaranarayanan, "Temperature-Aware Microarchitecture," in *Proceedings of ISCA-30*, pp. 2–13, 2003.

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

[48] K. Banerjee and A. Mehrotra, "A Power-optimal Repeater Insertion Methodology for Global Interconnects in Nanometer Designs," *IEEE Transactions on Electron Devices*, vol. 49, pp. 2001–2007, November 2002.

[49] W.-L. Hung, G. Link, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, "Interconnect and thermal-aware floorplanning for 3d microprocessors," *isqed*, vol. 0, pp. 98–104, 2006.

[50] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," *Journal of Instruction-Level Parallelism*, vol. 1, October 1999.

[51] J. Casmira and D. Grunwald, "Dynamic Instruction Scheduling Slack," in *Proceedings of KoolChips Workshop at MICRO-00*, 2000.

[52] E. Tune, D. Liang, D. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," in *Proceedings of HPCA-7*, pp. 185–196, January 2001.

[53] W. J. Dally and B. Towles, "Route packets, not wires: Onchip interconnection networks," in *Proceedings of Design Automation Conference*, pp. 684–689, 2001.

[54] L. Benini and G. D. Micheli, "Networks on Chip: A New Paradigm for Systems on Chip Design," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 418–419, 2002.

[55] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express Virtual Channels: Towards the Ideal Interconnection Fabric," in *Proceedings of the 34th International Conference on Computer Architecture.*, June 2007.

[56] W. J. Dally, "Express cubes: Improving the performance of k-ary n-cube interconnection networks," pp. 1016–1023, 1991.

[57] S. Kaxiras and C. Young, "Coherence Communication Prediction in Shared-Memory Multiproce," in *Proceedings of HPCA-6*, pp. 156–167, 2000.

[58] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the 25th International Conference on Computer Architecture*, pp. 179–190, July 1998.

[59] J. Chang, J. Huh, R. Desikan, D. Burger, and G. S. Sohi, "Using Coherent Value Speculation to Improve Multiprocessor Performance," in *Proceedings of 1st Value-Prediction workshop*, 2003.

[60] A. R. Lebeck and D. A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," in *Proceedings of ISCA-22*, pp. 48–59, 1995.

[61] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35(2), pp. 50–58, February 2002.

[62] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, 2005.

[63] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of ISCA-22*, pp. 24–36, June 1995.

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

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

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

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

[68] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, N. Vijaykrishnan, and M. Kandemir, "Design and Management of 3D Chip Multiprocessors Using Network-in-Memory," in *Proceedings of ISCA-33*, June 2006.

[69] G. Loi, B. Agrawal, N. Srivastava, S. Lin, T. Sherwood, and K. Banerjee, "A Thermally-Aware Performance Analysis of Vertically Integrated (3-D) Processor-Memory Hierarchy," in *Proceedings of DAC-43*, June 2006.

[70] N. Muralimanohar and R. Balasubramonian, "Interconnect Design Considerations for Large NUCA Caches," in *Proceedings of ISCA-34. To Appear.*, June 2007.

[71] T. Li, A. R.Lebeck, and D. J.Sorin, "Quantifying Instruction Criticality for Shared Memory Multiprocessors," June 2003.