Turboscalar: A High Frequency High IPC Microarchitecture

Bryan Black and John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{black,bohuslav,shen}@ece.cmu.edu

Abstract

There is significant performance motivation to build larger and wider superscalar machines, however the implementation complexity can be overwhelming. When superscalar machines grow they necessarily become deeper in order to maintain frequency. As the pipeline depth increases the performance gained by a wide instruction fetch and dispatch is lost to branch misprediction penalty cycles. This work proposes the new ***Turboscalar*** *microarchitecture, which is strongly based on the superscalar paradigm. Turboscalar utilizes run time information to optimize instruction execution. This new microarchitecture increases performance by reducing implementation complexity, allowing the construction of very shallow wide pipelines, which yield high performance.*

Results: *A realistic Turboscalar implementation is proposed, that improves performance 66% over a wide deep superscalar that utilizes a block-based trace cache.*

1 Introduction

Most technologists anticipate the continuation of Moore's law of increasing chip density and complexity for another 10 years. However, existing superscalar techniques for harvesting instruction-level parallelism (ILP) are encountering strong diminishing returns. In order to justify building wider superscalar processors, new microarchitecture techniques capable of achieving significantly higher IPC (average instructions executed per cycle) for ordinary programs are essential.

Many of the traditional methods of improving IPC performance (branch prediction, larger caches, wider pipelines) add complexity and increase the cycle time and/or pipeline depth of a design. Current microprocessors depend primarily on technology enhancements and pipelining to improve frequency and overall performance. Figure 1 illustrates the effects of front-end pipeline depth and instruction fetch/dispatch width on the IPC performance of a superscalar microarchitecture. As the number of instructions issued per cycle increases, more parallelism can be extracted resulting in increased IPC; however, greater front-end pipeline depth increases branch misprediction penalty resulting in reduced IPC. The benefit of shallow wide pipeline configurations in achieving high IPC is clear from Figure 1. The challenge is how to do this without increasing front-end pipeline depth and without increasing clock cycle time.

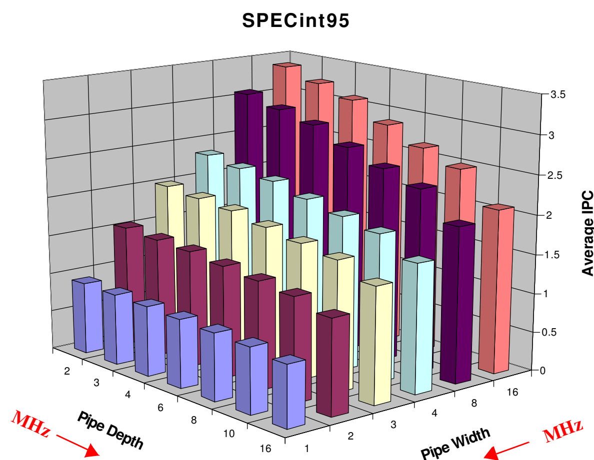

Figure 1 - Average IPC for a superscalar machine with varying front-end pipeline depth and overall pipeline width.

This work proposes and demonstrates a new microarchitecture paradigm (Turboscalar) that makes shallow wide superscalar microprocessor designs possible. It utilizes run time information to simplify design complexity, which leads to a high frequency implementation with very high IPC.

2 Previous Work

The complexity of superscalar microprocessors is increasing rapidly with each new generation. As newer and larger performance enhancing devices are added, design time increases and high frequency design becomes more challenging. When developing new techniques to improve IPC the impact on frequency and complexity must be examined. It is important not to propose large cache structures and complicated control logic. Another complexity concern is that the RC delay of wire interconnect is not scaling as process technology shrinks. Wiring delay manifests itself in every aspect of implementation, including instruction result forwarding paths, and dispatch crossbars. The impact of these complexity issues on microprocessor design has been explored in [11].

An early attempt to reduce the complexity of hardware implementations is the VLIW architecture [14]. Specifically the VLIW moves the complexity of hardware implementation to the compiler. The compiler is responsible for resolving all data and resource hazards, thus eliminating register renaming, result forwarding, and the more traditional dispatch crossbar. The VLIW architecture effectively trades hardware-extracted IPC performance for simple design and higher frequency implementation. More recently the EPIC architecture [9] introduced by Intel and HP utilizes the compiler to simplify the design, similar to VLIW. A key problem with the VLIW/EPIC machines is that the code optimized for a current generation machine may not run well on future generation designs.

Academic researchers have investigated the complexity of microarchitecture and attempted to reduce the complexity of result forwarding paths and wide instruction dispatch widths; examples include Multiscalar [18] and the Trace Processor [16]. Multiscalar and Trace Processors relieve the data forwarding problem by breaking up the execution resources into sub-units. However, they do not address the front-end complexity.

This work proposes the new ***Turboscalar*** microarchitecture, which is strongly based on the superscalar paradigm. Turboscalar utilizes run time information to optimize instruction execution. This new microarchitecture increases performance in three primary ways. First, it reduces complexity by eliminating the repetitive and redundant instruction execution overhead. This overhead is the invariant portion of instruction processing that does not contribute to the intrinsic function of the instruction. For example, instruction decode, register read, and register renaming are all necessary to instruction execution, but do not contribute to the final data result. Second, Turboscalar facilitates the dynamic transformation of the object code via an Instruction-Path Coprocessor [3].

Finally, through dynamic code transformation, the Turboscalar microarchitecture can implement optimized execution cores that still efficiently execute legacy code. Ideally the execution core of a new microarchitecture is optimized for the new code base of a target machine, however this typically penalizes the execution of legacy code. The need to support legacy code can adversely affect the efficiency of the final design. With dynamic code transformation, the legacy code can be translated into the internal format for highly efficient execution by the new execution core.

The previous work that most resembles the Turboscalar idea is the Dynamic Instruction Formatting (DIF) work of Nair and Hopkins [10]. In that work the focus is on the mapping of PowerPC instructions into an internal VLIW format and the potential performance gains. This work does not strictly employ VLIW as the internal format, it still employs the superscalar paradigm.

The focus of Turboscalar is on efficient implementation through knowledge of execution behavior. It is found that many aspects of a superscalar machine can be simplified simply by remembering much of the instruction processing work. Section 3 discusses the concept of the Turboscalar microarchitecture, while Section 4 details a possible implementation.

3 Turboscalar Microarchitecture

The Turboscalar microarchitecture, illustrated in Figure 2, has three key components that differentiate it from a conventional superscalar microarchitecture: the *hot-cold pipeline*, the *dynamic instruction cache*, and the *optimizing back-end*. The cold pipeline trains the hot pipeline, for fast efficient execution, using the optimizing back-end and the dynamic instruction cache.

3.1 Hot-Cold Pipeline Organization

The Turboscalar microarchitecture is implemented with two superscalar pipelines. The cold pipeline is tall and thin, while the hot pipeline is short and fat. The cold pipeline is analogous to the traditional superscalar microarchitecture. It must fetch, decode, predict branches, read sources, rename results, and dispatch instructions to the executions units, before instruction execution may commence. It suffers from the scaling problems of instruction dispatch width and front-end pipeline depth as discussed in Section 1. On the other hand, the hot pipeline avoids these problems by learning about instruction execution characteristics from the cold pipeline and leveraging this knowledge. Thus the hot pipeline can be very wide without incurring front-end pipeline depth.

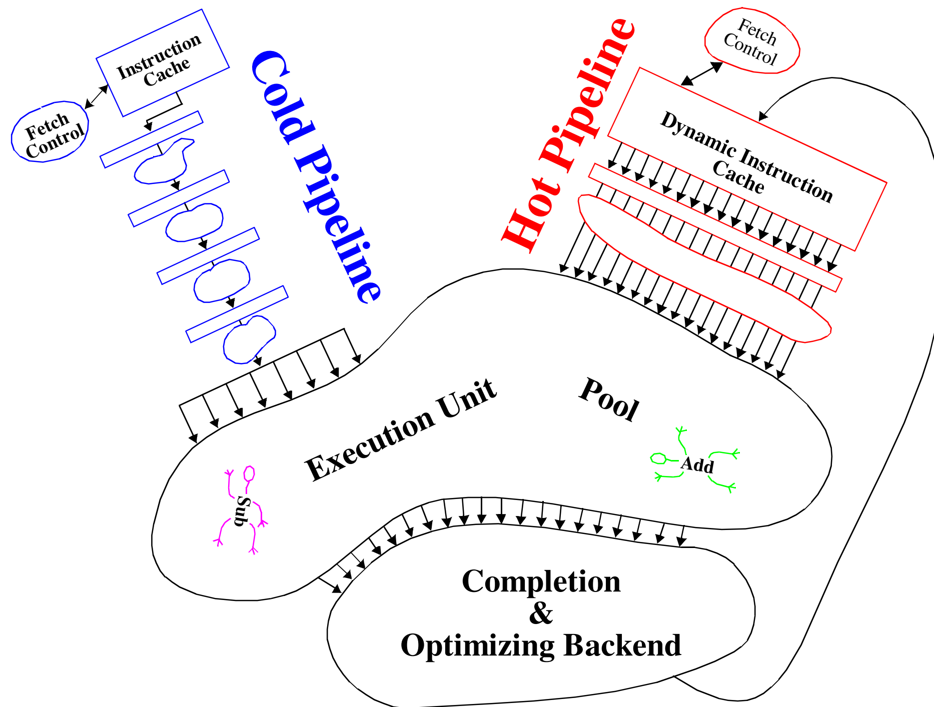

Figure 2 - The Turboscalar microarchitecture.

The cold pipeline executes instructions as a traditional superscalar. As instructions complete in the cold pipeline, the optimizing back-end records execution characteristics and saves them in a dynamic instruction cache. Once the hot pipeline is trained, instruction execution shifts from the cold pipeline to the hot pipeline. Obviously, there is a small penalty in this hand off process. If the program executes many more instructions in the hot pipeline than in the cold pipeline, then there can be a significant performance gain.

Figure 3 illustrates the foot print of traditional superscalar machines as the machine width and the pipeline depth increase. The traditional machine necessarily becomes deeper and wider as performance is improved. Continuing this trend towards ultra-wide superscalar will lead to very large and complex implementations. The advantage of the Turboscalar microarchitecture is the interaction between the hot and cold pipelines. Using the cold pipeline to train the hot pipeline allows the Turboscalar microarchitecture to implement the ultra-wide superscalar with a much shallower and higher frequency cold and hot pipeline combination, shown in Figure 4. The cold pipeline is tall and thin providing a high frequency implementation, while the hot pipeline is wide and shallow allowing very high IPC performance.

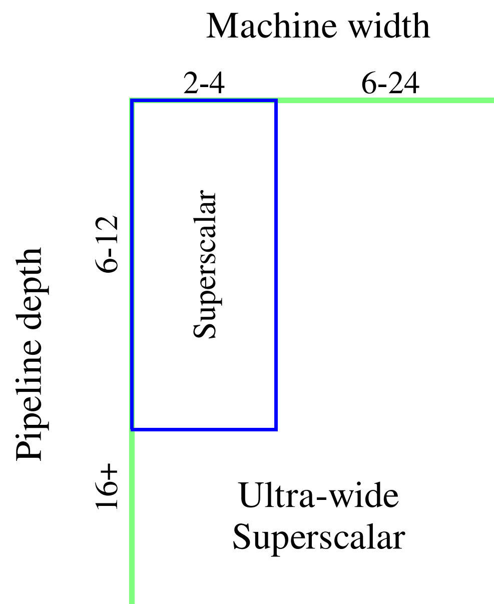

Figure 3 - Traditional superscalar compared to a ultra-wide superscalar foot print.

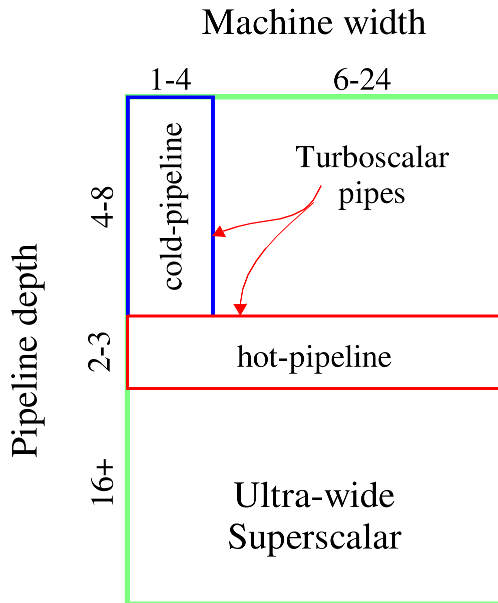

Figure 4 - Turboscalar compared to a ultra-wide superscalar foot print.

3.2 Dynamic Instruction Cache

The dynamic instruction cache is used to support instruction fetch in the hot pipeline. This mechanism allows the cold pipeline to train the hot pipeline for faster and more efficient execution (hence the analogy to "turbocharging" in a combustion engine). The hot pipeline, via the dynamic instruction cache and the optimizations performed by the associated optimizing back-end, effectively *remembers* and leverages previous work, allowing it to fetch and dispatch more instructions in a faster clock cycle and process instructions in a much shallower pipeline because the majority of the work has been done ahead of time and recorded in the dynamic instruction cache.

3.3 Optimizing Back-end

The optimizing back-end of Turboscalar is responsible for all code transformation, performance optimization, and the training of the hot pipeline. To effectively remove repetitive instruction processing overhead the optimizing back-end constructs instruction fetch groups, performs code motion to eliminate the dispatch crossbar, predecodes instructions to eliminate or simplify the decode stage and pre-renames (including dependency check) to simplify register read and renaming after instruction fetch. The optimizing back-end can also perform compiler type optimizations as in [7]. Instruction translation can also be performed if the Turboscalar core employs a different internal format. Similar to the Pentium Pro uop [17], the Turboscalar can translate macro-instructions into internal micro-instructions, however it only needs to perform this translation once to train the hot pipeline, for repeated subsequent executions. Once a group of instructions is gathered and all optimizations are complete it is inserted into the dynamic instruction cache for execution in the hot pipeline. The recently introduced Instruction-path Coprocessor (I-COP) [3] can be an effective and efficient way to implement the optimizing back-end of the Turboscalar microarchitecture.

4 A Turboscalar Implementation

The fundamental contribution of Turboscalar is the interaction of the hot-cold pipelines, the dynamic instruction cache and the optimizing back-end. This unique organization allows the Turboscalar microarchitecture to perform many operations early in the optimizing back-end and remember them in the dynamic instruction cache. As a result the hot pipeline implementation is simple, shallow and wide, leading to high performance and high frequency.

There are many possible implementations of a Turboscalar microarchitecture. This section explores one specific implementation that is used for study in Section 6. The implementation presented is not intended to be the best or optimal solution, but merely a single point solution in a large design space for illustration purpose. For discussion the microarchitecture is divided into three major components the front-end, execution core, and the back-end.

4.1 Front-end

The front-end of the Turboscalar microarchitecture includes both the cold pipeline and the hot pipeline. These two pipelines are responsible for all instruction fetch, decode, dependency checking, register renaming, register read, and dispatch to the reservation stations.

4.1.1 Cold Pipeline

The cold pipeline is simply the front-end of a traditional superscalar machine. Instructions are fetched from a traditional instruction cache and branch prediction is performed early in the pipeline. Instructions are decoded and after dependency checking is performed source operands are read and result operands are renamed. Finally instructions are dispatched to the reservation stations in the execution core. The cold pipeline only fetches instructions after a branch misprediction and when the hot pipeline is empty.

4.1.2 Hot Pipeline

The hot pipeline fetches instructions from the dynamic instruction cache. The dynamic instruction cache stores both instructions and execution information. For this study

the dynamic instruction cache is implemented with the block-based trace cache [1], which is enhanced to include decode bits, dependency check results, and virtual renaming tags for result operands. Figure 5 illustrates the flow of instructions through the hot pipeline. ***b*** Instructions are fetched from each of the four copies of the block cache. Each instruction block contains enough decode and dependency information such that little or no decode time is necessary before instructions are ready for dispatch. Only source operand read is required after instruction fetch. (**Note:** no branch prediction is performed in the hot pipeline.) All branch prediction for the hot pipeline is performed in the optimizing back-end, during trace construction, using the new TMP multiple branch predictor [13], as discussed in Section 4.3.

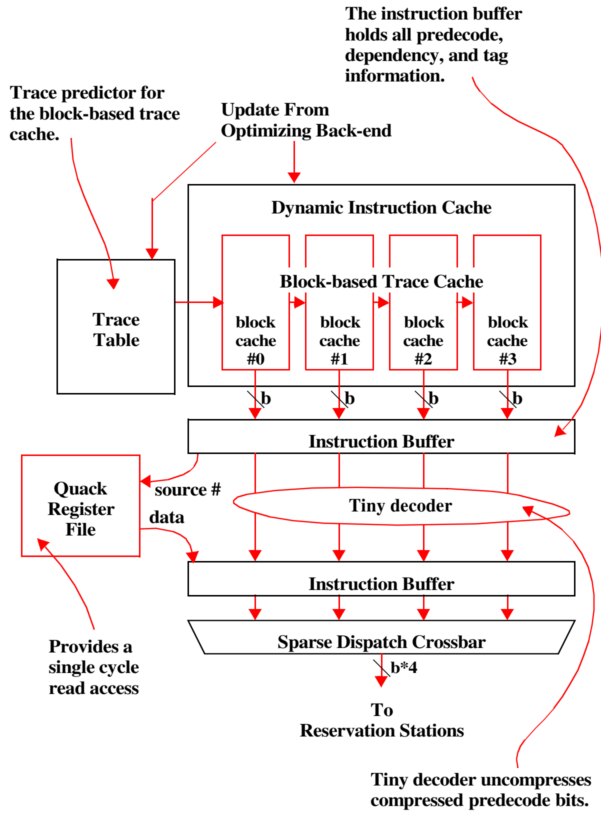

Figure 5 - Hot pipeline intruction flow.

Given the shallow hot pipeline of the Turboscalar, a new fast single cycle register file is required. The Quack register file, proposed in [2], provides three key ingredients to the Turboscalar implementation. First it utilizes a virtual rename tag that can be generated in the back-end, requiring no renaming in the front-end of the machine. Second, it has a high frequency read that is independent of the physical rename register count and instruction dispatch width. Third, fewer read and write ports are required by the Quack register file than more traditional implementations, further facilitating source operand read for a large number of instructions in a single cycle. [2] analyzes the latency of the Quack register file for a 0.18um process, and demonstrates that source operand read for 24 instructions can be performed in a single cycle. The Quack register file organization is illustrated in Figure 6. (Extensive details of the Quack register file are documented in [2].)

The three advantages of the Quack register file are achieved by organizing the physical rename registers into stacks of registers or silos. There is one register silo for each architected register. The top of each silo is the most recent rename of that architected register. Maintaining the most recent rename in the same physical location at all times, significantly reduces source operand read latency, and removes the need for a rename map table.

After instructions are fetched and source operands are read, they are dispatched through a sparse crossbar to the reservation stations in the execution core. In order to reduce the complexity of the dispatch crossbar, each dispatch slot is reserved for particular instruction types, similar to the Al-

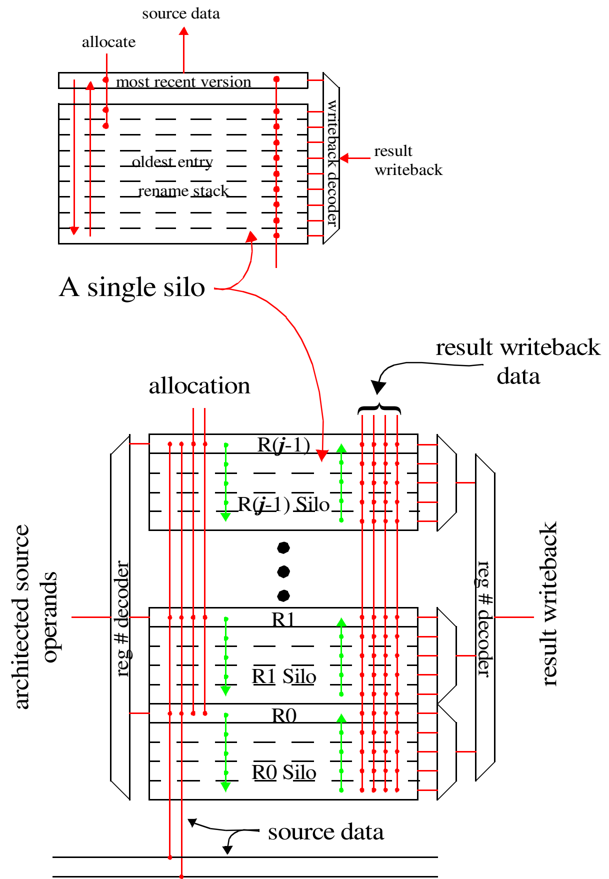

Figure 6 - The Quack register file structure.

pha 21064 [6]. All instruction alignment to dispatch slots is performed by the optimizing back-end.

4.2 Execution Core

The execution core of both the baseline machine model and the Turboscalar microachitecture are idealized, yielding higher achievable performance. Execution units are clustered by function type with 128 entry reservation stations per cluster. Each cluster contains an unbounded number of functional units and instructions are issued out-of-order from the reservation stations. Result forwarding between functional units is a single cycle. This highly idealized execution core stresses the performance of the front-end and back-end of the Turboscalar microarchitecture. Current ongoing research is investigating more realistic implementations of the execution core within the Turboscalar framework.

4.3 Back-end

The optimizing back-end of the Turboscalar microarchitecture performs several tasks that simplify the front-end of the hot pipeline. First, to support the trace construction of the block-based trace cache, instructions are arranged into blocks and traces are constructed from the completing blocks. As part of trace construction a completion time multiple branch predictor is used to determine which blocks belong in a trace and which traces are candidates for future trace fetching. The TMP predictor [13] utilizes a tree structure and three levels of branch predictors that record branch behavior for the control flow graph. The TMP predictor is very effective and as a result fetch time branch prediction is not necessary.

The second function of the back-end is to perform early dependency checking as instructions are bundled into blocks for insertion into the block-based trace cache. Early register renaming is also performed at the block level and virtual tags are assigned to destination operands and propagated to all dependent sources in the instruction block. Finally, to reduce the dispatch crossbar complexity, instructions within a block are reordered to align with dedicated dispatch slots. A preliminary performance study of the alignment is presented in Section 6. Instruction alignment facilitates a sparse dispatch crossbar, reducing the latency of instruction dispatch.

4.4 Hot or Cold?

Instructions are fetched from the hot pipeline whenever possible. The mechanism controlling which pipeline to access is very similar to a trace cache in parallel to an instruction cache [2][12][15]. After trace prediction it is known if the block-based trace cache will hit or miss in the next cycle. If the trace cache will hit then the hot pipeline gains control. If it will miss then the cold pipeline is accessed. When a branch misprediction is detected the cold pipeline is automatically accessed, until the hot pipeline warms up again. Neither pipeline becomes active until the other pipeline has dispatched all instructions, potentially incurring a significant fetch stall. The fetch interlock is necessary to ensure that instructions from the two pipelines rename in the correct order. Having high percentage of instructions fetched from the hot pipeline versus the cold pipeline is critical to the performance of Turboscalar. Section 6.4 examines the performance of hot pipeline instruction fetch.

5 Experimental Methodology

All the experimental data reported are generated by an integrated functional/performance simulator based on the PowerPC ISA. The performance model is developed from published reports [5][8][19] and accurately models all key features of the microarchitecture.

5.1 Baseline Machine Model

The reference machine uses the execution core outlined in Section 4.2. The instruction fetch, dispatch, and completion bandwidth is 16 instructions per cycle. An unlimited number of rename registers and reorder buffer entries is assumed.

The memory hierarchy is fully modeled with a perfect main memory, a 32KB Level-1 I-cache, a 32KB Level-1 D-cache, and a 256KB unified Level-2 cache. Access latencies are 1, 3, and 100 cycles for the L1, L2 caches and the main memory, respectively. On-chip implementation of the L2 is assumed. An unlimited load miss queue and an unlimited store queue handle all load and store execution. The store queue performs data forwarding, and load/store instructions execute out-of-order if no address aliasing is detected. All register and memory data dependencies are enforced. Instruction execution latencies can be found in [8], and accurately reflect the PowerPC 604.

This baseline machine model is currently very aggressive, and is specifically designed to gather the preliminary data presented in Section 6. Future versions of this work will contain a realistic implementation for all aspects of both the baseline superscalar machine model and the Turboscalar microarchitecture.

5.2 Benchmarks

The benchmark set used is the SPECint95 suite, compiled by gcc 2.7.2. To reduce simulation time, we use small input files and limit run length to 200 million instructions for each benchmark, totaling 1.6 billion instructions. All user library calls are modeled, though system calls are not.

6 Experimental Results

The baseline machine model outlined in Section 5, is modified for the Turboscalar implementation as described in Section 4. All features of the Turboscalar implementation are fully modeled. This section begins with an analysis of the optimal configuration of the cold pipeline, determining the pipeline depth and instruction width. The sparse dispatch crossbar is explored and the Turboscalar performance is compared to superscalar machines with trace cache instruction fetch mechanisms.

6.1 Cold Pipeline Configuration

The most significant design parameter of the Turboscalar microarchitecture is the cold pipeline depth and width. The wider and shallower the cold pipeline the faster the hot pipeline is trained. On the other hand, the clock frequency is adversely affected as the cold pipeline becomes wider and shallower. Figure 7 shows the performance of the Turboscalar microarchitecture for different configurations of the cold pipeline. The hot pipeline is 24 instructions wide and 3 pipe stages deep in its front-end, as illustrated in Figure 5. Turboscalar performance is shown as a function of both the

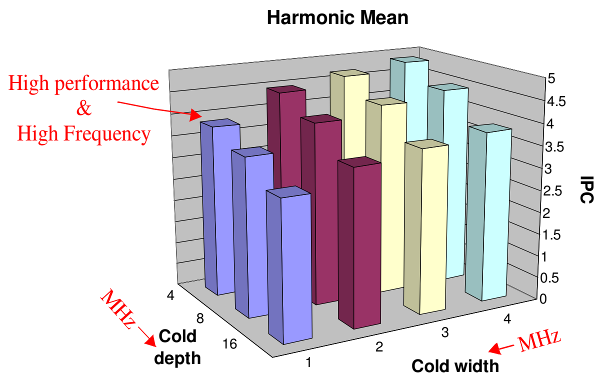

Figure 7 - Turboscalar performance vs. cold pipeline configurations.

cold pipeline depth and width. (**Note:** As the cold pipeline depth increases the frequency increases and as the cold pipeline width increases the frequency decreases.) Turboscalar performance varies from 3.1 IPC to 4.9 IPC. At 3.1 IPC the cold pipeline depth is 16 and the width is only 1 instruction, with potentially the highest frequency. At 4.9 IPC both the width and depth are 4. This figure demonstrates the importance of the cold pipeline and how it trains the hot pipeline. The wider and shallower the cold pipeline the better overall performance becomes. Optimizing for both frequency and performance is difficult. The remainder of this paper assumes a cold pipeline depth of 4 and a width of 1 instruction, yielding an average IPC of 3.9 for the Turboscalar microarchitecture.

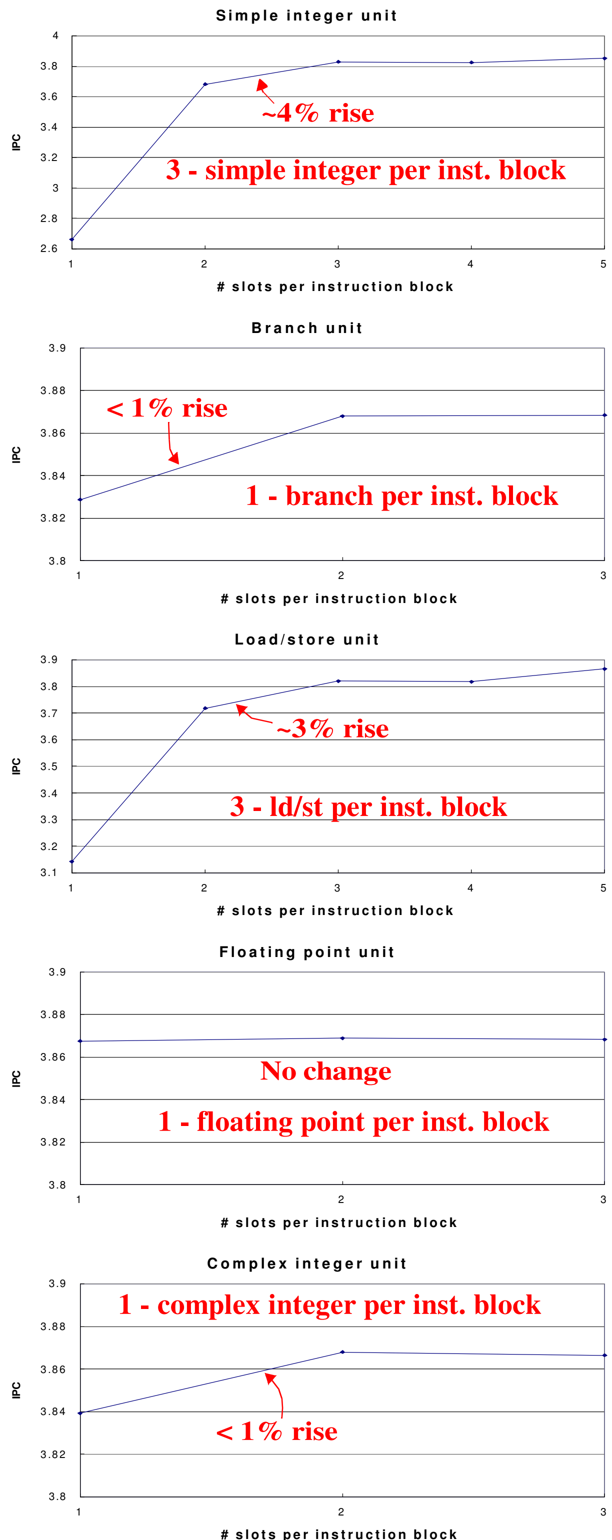

Figure 8 - Number of dispatch slots of each instruction type required per instruction block.

6.2 Sparse Dispatch Crossbar

In order to reduce the complexity of the dispatch crossbar two limits are imposed on the dispatch mechanism. First, each dispatch slot is reserved for a particular instruction type. As instructions are bundled into blocks, for the block-based trace cache, they are aligned to the dedicated dispatch slots. The number of dispatch slots required, within each instruction block, for each instruction type is shown in Figure 8. The graphs in Figure 8 show the diminishing return for additional dispatch slots for each of the five PowerPC instruction types. Only 1 slot is required for branch unit instructions (branches and condition code logicals) and complex integer instructions (compare, integer multiply and divide). Both load/store and simple integer (add, sub, etc.) require 3 slots each to maintain optimal performance. Finally, since only integer benchmarks are examined 1 floating point slot is sufficient. A total of 9 instruction type slots are required per block of 6 instructions, resulting in 2 or fewer instruction types allowed in each dispatch slot, which significantly reduces the complexity of the traditional dispatch crossbar.

The second dispatch limitation is on the number of instructions of each type that are allowed to dispatch in a single cycle. Figure 9 illustrates the diminishing return as the number of dispatch slots for each instruction type is increased. Again, since integer benchmarks are analyzed only 1 floating point dispatch slot is necessary. 4 branch slots are required which coincides with the 4 instruction block fetch. 2 complex integer dispatch slots, 10 load/store slots, and 12 simple integer slots are required to support the 24 instruction wide hot pipeline.

The dispatch crossbar complexity is reduced by limiting the number of instruction types that can reside in any slot to 2. It is further reduced by limiting the number of each instruction type that can be dispatched in a single cycle. Consequently, the branch unit crossbar is 6:4, the complex integer unit crossbar is 6:2, the floating point unit crossbar is 6:1, the simple integer unit crossbar is 18:12, and the load/store unit crossbar is 18:10. Increasing the number of dispatch slots per cycle available for each instruction type would completely remove the dispatch crossbar. The Turboscalar performance study in the next section utilizes this sparse dispatch crossbar.

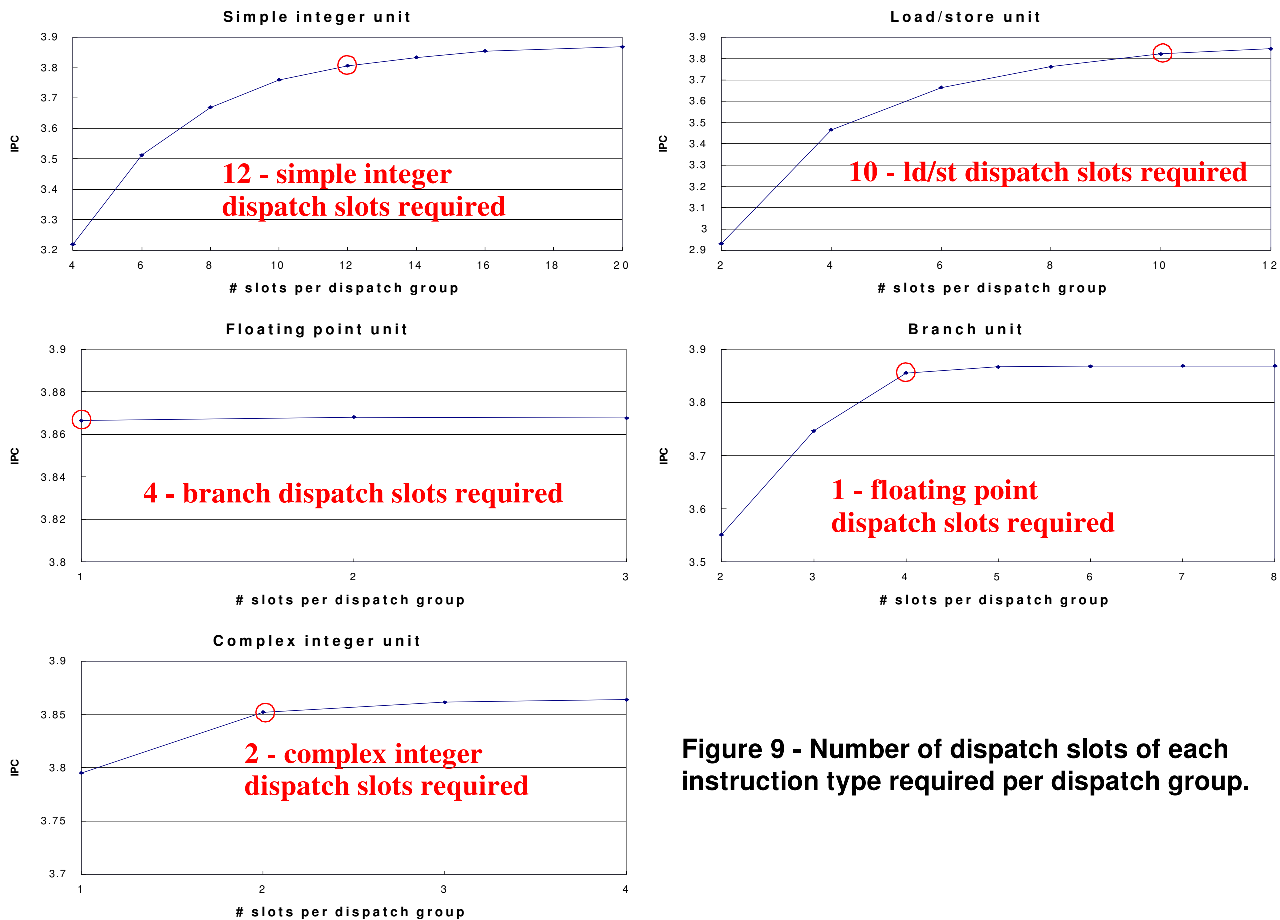

Figure 9 - Number of dispatch slots of each instruction type required per dispatch group.

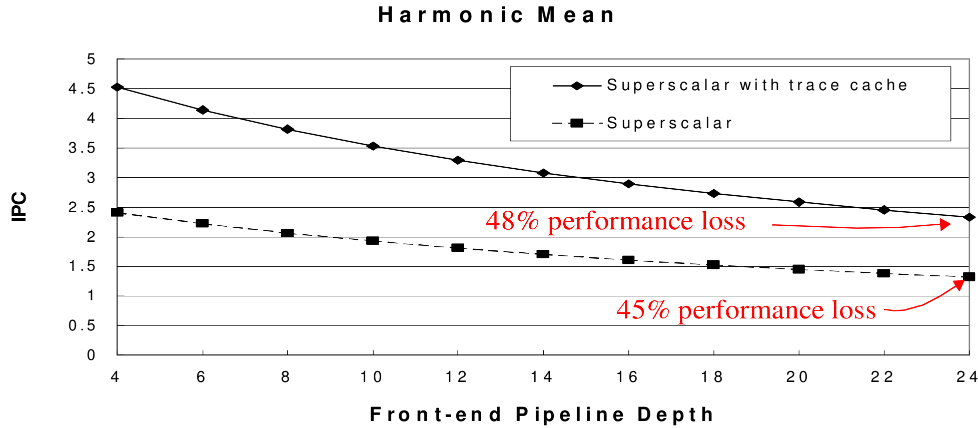

Figure 10 - Effects of front-end pipeline depth on performance for wide instruction fetch.

6.3 Turboscalar Performance Comparison

Recently there have been many proposals that very effectively increase instruction fetch bandwidth. These mechanisms include the new trace cache [2][12][15] and multiple branch predictors accessing multiple instruction caches [4]. All of these proposals increase instruction fetch bandwidth to 16 or more instructions, yet maintain a very shallow front-end pipeline. They assume instruction decode, source read, result renaming, and instruction dispatch will scale to accommodate the new wider instruction fetch mechanisms. Figure 10 illustrates the effect of front-end pipeline depth on a 16 instruction wide superscalar machine both with and without a trace cache. As the pipeline depth increases the performance gained by the wide instruction fetch is lost to branch misprediction penalty cycles. A wide superscalar machine loses 45-48% of its performance as the front-end depth increases from 4 to 24 cycles. Not only are these shallow pipelines not feasible, the performance is lost once the pipeline depth increases.

The Turboscalar microarchitecture proposes a feasible and shallow front-end pipeline implementation. Figure 12 compares the performance of the Turboscalar microarchitecture to the superscalar with and without a trace cache for each benchmark. The Turboscalar significantly out performs a wide superscalar as the front-end pipeline depth increases. At a 24 cycle front-end pipeline, Turboscalar performance is 37% to 92% better than a wide superscalar utilizing a trace cache.

Figure 11 is the harmonic mean of the benchmark results in Figure 12. The Turboscalar microarchitecture begins to out perform superscalar with trace cache at a 10 cycle front-end. When the superscalar front-end reaches 24 cycles Turboscalar is 66% better.

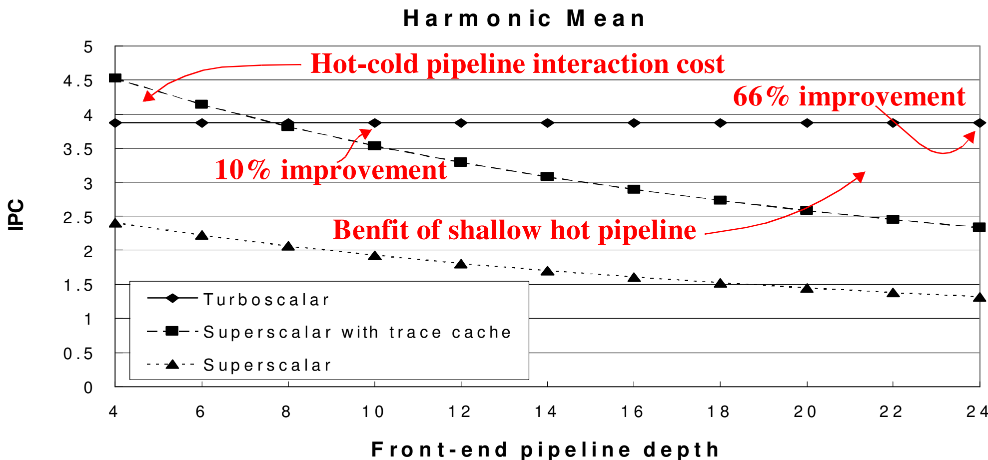

Figure 11 - Turboscalar performance compared to superscalar performance (harmonic mean).

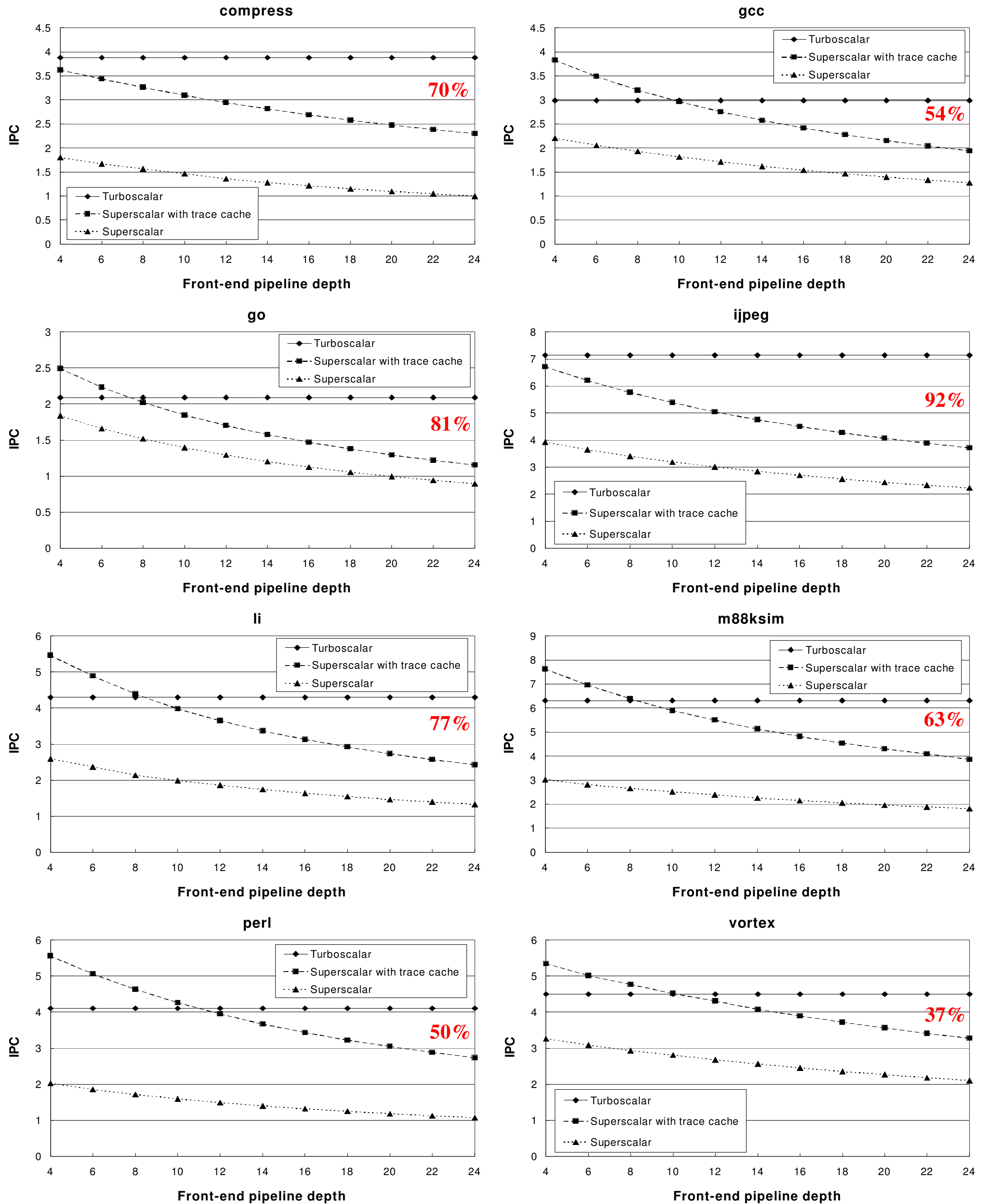

Figure 12 - Turboscalar performance compared to superscalar performance (all benchmarks).

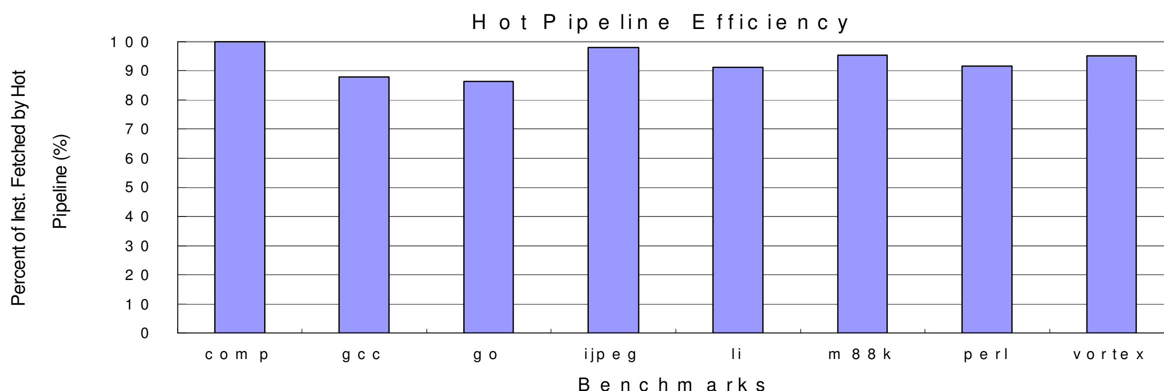

Figure 13 - The percentage of dynamic instructions fetched by the hot pipeline.

6.4 Instruction Fetch

The performance of the Turboscalar is dependent on the percentage of time instructions are fetched from the hot pipeline instead of the cold pipeline, shown in Figure 13. From this figure the hot pipeline is fetching between 86% to 99% of the dynamic instructions for these benchmarks. As better multiple branch prediction mechanisms are developed the Turboscalar microarchitecture will become even more advantageous.

7 Conclusion

The data presented in Figure 11, Figure 12, and Figure 13 demonstrate the potential of the Turboscalar microarchitecture. Given a strong hot path bias, as shown in Figure 13, the Turboscalar can improve performance significantly over a trace-cache based superscalar implementation. The Turboscalar can take advantage of the cold pipeline to train a hot pipeline for fast efficient implementation and high throughput code execution.

The Turboscalar with a cold pipeline width of 1 instruction and a depth of 4 cycles out performs trace-cache based superscalar machines with front-end pipelines greater than 8 cycles. Once the superscalar front-end pipeline depth reaches 24 cycles, Turboscalar performance is 66% better. The Turboscalar microarchitecture is a clear paradigm shift in microprocessor design over current superscalar designs. The smart hot pipeline of the Turboscalar facilitates high performance at a high frequency, while utilizing a very thin, deep, high frequency cold pipeline to train itself.

This paper presents an early exploration of a new microarchitecture paradigm that can potentially harvest unprecedented levels of ILP at very high frequencies. The Turboscalar microarchitecture utilizes run time information to construct a shallow wide high frequency high ILP hot pipeline implementation.

8 References

[1] B. Black, B. Rychlik, and J. Shen, “The Block-based Trace Cache” In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 196-207, May 1999.

[2] B. Black and J. Shen, “Scalable Register Renaming via the Quack Register File” Technical Report CMuArt 00-1, Carnegie Mellon University, April 2000.

[3] Y. Chou and J. Shen, “Instruction Path Coprocessors” To appear in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[4] T. Conte, K. Menezes, P. Mills, and B. Patel, “Optimization of Instruction Fetch Mechanisms for High Issue Rates” In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-343, June 1995.

[5] K. Diefendorf, and E. Silha, “The PowerPC User Instruction Set Architecture” In *IEEE Micro*, Vol. 14, No. 5, pp. 30-41, October 1994.

[6] Digital Equipment Corporation, DECchip 21064-AA Microprocessor Hardware Reference Manual, 1992.

[7] D. Friendly, S. Patel, and Y. Patt, “Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors” In *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 173-181, December 1998.

[8] IBM Microelectronics Division, PowerPC 604 RISC Microprocessor User’s Manual, 1994.

[9] Intel Corporation, “Intel Itanium Processor Microarchitecture Overview” Found on Intel web site: http://developer.intel.com/design/IA-64/microarch_ovw/index.htm.

[10] R. Nair and M. Hopkins, "Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Group" In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[11] S. Palacharia, N. Jouppi, and J. Smith, “Complexity-Effective Superscalar Processors” In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206-218, June 1997.

[12] S. Patel, M. Evers, and Y. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing" In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 262-271, June 1998.

[13] R. Rakvic, B. Black, and J. Shen, "Completion Time Multiple Branch Prediction for Enhancing Trace Cache Performance" To appear in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[14] B. Rau and J. Fisher, "Instruction-Level Parallelism" A special issue of the *Journal of Supercomputing*, Vol. 7 No. 1/2, Kluwer Academic Publishers, 1993.

[15] E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching" In *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, December 1996.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors" In *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 138-148, December 1997.

[17] J. P. Shen, Fundamentals of Superscalar Processor Design. Chapter 4, to be published by McGraw-Hill, 1997.

[18] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors" In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414-425, May 1995.

[19] S. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC Microprocessor" *IEEE Micro*, Vol. 14, No. 5, pp. 8-17, October 1994.