A Scalable Register File Architecture for Dynamically Scheduled Processors

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697
swallace@ece.uci.edu, nader@ece.uci.edu

A Scalable Register File Architecture for Dynamically Scheduled Processors

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697
swallace@ece.uci.edu, nader@ece.uci.edu

Abstract

A major obstacle in designing dynamically scheduled processors is the size and port requirement of the register file. By using a multiple banked register file and performing dynamic result renaming, a scalable register file architecture can be implemented without performance degradation. In addition, a new hybrid register renaming technique to efficiently map the logical to physical registers and reduce the branch misprediction penalty is introduced. The performance was simulated using the SPEC95 benchmark suite.

1 Introduction

The instruction level parallelism in programs allows a superscalar microprocessor to execute multiple instructions per cycle. It can be exploited by hardware providing resources to perform *dynamic instruction scheduling*. Independent instructions can be discovered at run-time and scheduled to functional units out-of-order. To avoid anti and output dependencies, registers can be renamed by using a *reorder buffer* [5] or a *mapping table* [10]. The Power PC [7], Pentium Pro [6], and MIPS R10000 [10] are examples of superscalar microprocessors that perform out-of-order issue and register renaming.

The register file is a design obstacle for superscalar microprocessors. If N instructions can be issued in a cycle, then a superscalar microprocessor's register file needs $2N$ read ports and N write ports to handle the worst-case scenario. The area complexity for the register file grows proportional to N^2 [2]. Therefore, a new architecture is needed to keep the ports of a register file cell constant as N increases.

In addition, the register requirements for high performance and exception handling can be quite high. Farkas et. al. conclude that for best performance, 160 registers are needed for a four-way issue machine, and 256 registers are needed for an eight-way issue machine [4]. Therefore, it is desirable to reduce the register requirements and still maintain performance.

A major difficulty in the simultaneous multithreading (SMT) architecture, introduced by Tullsen et. al., was the size of the register file [8]. They supported eight threads on an eight-way issue machine, using 356 total registers with 16 read ports and 8 write ports. Compared to a standard 32-register, 2 read port, 1 write port register file of a scalar processor, the area of the SMT register file is estimated to be over seven hundred times larger. To account for the size of the register file, they took two cycles to read registers instead of one. This underscores the need for a mechanism to scale the register file, yet still have the benefits of area and access time of a register file for a scalar processor.

To attack this problem, we approach the problem of scalability of a register file in two directions. First, we show how it is possible to use multiple banked register file to reduce the port requirement to that of a scalar processor: 2 read ports and 1 write port. Second, we show how to reduce the total register requirement by improving the utilization of the registers.

Register renaming is performed by mapping the logical registers into physical registers, as in the MIPS R10000 [10]. When an instruction is decoded, a new physical register from a free list is allocated for its destination register and entered into a mapping table. The old physical register for that register is entered into a recovery list. The recovery list (also called the active list) maintains the in-order state of instructions and can be used to undo the mappings in the event of a mispredicted branch or exception. After an instruction completes and all previous instructions have completed, its register is committed and the old value is discarded by freeing the old physical register contained in the recovery list. The mapping table is used to lookup the current logical to physical registers for source operands.

A major drawback with this mapping technique is the large penalty required to recover from a branch misprediction or exception, except if a checkpoint mechanism is used. The R10000 uses checkpointing for up to four branches, but not for exceptions [10]. In order for checkpointing to be realistic in hardware, checkpoint storage must be integrated into the basic cell of the mapping table to have direct access for a single cycle recovery. Thus, a mapping table's cell size is greatly increased and is not scalable with the number of branch checkpoints. With increased speculation from wide-issue superscalars and larger mapping tables from SMTs, using standard RAM cells becomes imperative for speed and scalability. Therefore, we introduce a new hybrid register renaming technique which significantly reduces this penalty yet still retains some of the benefits of a mapping table, yet still remains scalable.

To begin with, we explain our simulation environment and machine model. Then we discuss register renaming and our hybrid technique. After discussing register file utilization, we finally present the dynamic result renaming mechanism for a scalable register file architecture.

2 Experimental Methodology

The performance of the concepts we describe was simulated by running the SPEC95 benchmark suite on the SPARC architecture using the Shade instruction-set simulator [3]. Each program was compiled using the SunPro compiler with standard optimizations (-O) and simulated for the first 100 million instructions. Also, we simulated the SPECint95 benchmark suite on the SDSP (Superscalar Digital Signal Processor) architecture [9], using the GNU CC compiler and second-level optimizations (-O2). The instruction set of the SDSP is very similar to MIPS.

To verify that our new register file architecture performs, we chose a reasonable machine model that resembles commercial processors including Power PC 604 [7] and MIPS R10000 [10]. Table 1 lists the quantity, type, and latency of the different function units modeled. The quantity of functional units for the 8-way superscalar architecture is twice that of the 4-way superscalar architecture.

The machine model parameters used in simulation are: **instruction cache**: 64 Kbyte, two-way set associative LRU, 16 byte line size, 2 banks, self-aligned fetching, 10 cycle miss penalty; **data cache**: 64 Kbyte, two-way set associative LRU, 16 byte line size, 4 banks, 2 simultaneous accesses per cycle, lockup-free, write-around, write-through, 4 outstanding cache request capability, 10 cycle miss penalty; **branch prediction**: 2K x 2-bit pattern history table indexed by XOR of PC and global history register [11]; **speculative execution**: enabled; **interrupts**: precise; **instruction window**: centralized, 32 entries for 4-way, 64 entries for 8-way; **register file**: separate general purpose and floating point register files; logical registers are mapped to physical registers; **recovery list**: 32 entries for 4-way, 64 entries for 8-way; **store buffer**: 16 entries.

The instruction scheduling logic uses a single instruction window for all functional units [5]. We chose a reasonable size, 32 entries for 4-way issue and 64 entries for 8-way issue, that would give good performance and produce a strong demand for registers during issue and writeback. A variable number of instructions, up to the decode width of 4 or 8 (equal to issue capability), may be inserted into the instruction window, if entries are available. Instructions are issued out-of-order using an oldest first algorithm. Store instructions are issued in-order and load instructions may be issued out-of-order in between store instructions.

Table 1. Functional Unit Parameters

Quantity		Type	Latency
4-way	8-way		
4	8	ALU	1
2	4	Load unit	1
2	4	Store unit	-
1	2	Int. multiply	2
1	2	Int. divide	10
4	8	FP add	3
1	2	FP multiply	3
1	2	FP divide	16
1	2	FP other	3

Each cycle, a variable number of registers, up to the decode width, may be retired from the recovery list. If entries are available, then they may be used by new instructions, up to the decode width. The old physical register of corresponding to the same destination register is inserted into the recovery list.

The pipeline stages of the processor modeled are instruction fetch, decode and rename, issue, register read, execute, register write, and commit. Consequently, two levels of bypassing are required for back-to-back execution of dependent instructions. Also, we optimistically issue instructions dependent on a load, in expectation of a cache hit [8]. If a cache miss occurs, then the dependent instructions must be re-issued.

3 Register Renaming

A major performance penalty with using a mapping table and a recovery list is the time it takes to recover from a mispredicted branch. Using the recovery list, the mapping table can be recovered by undoing each entry in the list one at a time. The mapping table can be updated in groups at a time, but the rate is limited by the number of read and write ports. If P ports are available and M register mappings need to be recovered, then it will take M/P cycles to recover.

Hence, the longer it takes for the branch to be resolved and found incorrectly predicted, the larger the penalty. In fact, this essentially doubles the branch misprediction penalty, compared to a mechanism that can recover from a mispredicted branch in one cycle.

In addition, the ports of the mapping table grow proportional to N, so it is not ideally scalable. On the other hand, with a fixed number of entries and a relatively small entry size, N would have to be large in order to cause real problems from a practical standpoint. Nevertheless, it would be beneficial to reduce the number of ports.

3.1 CAM/Table Hybrid

The beauty of a register renaming mechanism that uses CAM logic, such as a reorder buffer [5], is that it can recover from a mispredicted branch in one cycle by simply invalidating appropriate entries relative to the branch. We could use a CAM lookup to search for speculative registers (extra registers reserved for temporary or speculative use until committed), and a table to hold the logical to physical mapping of committed registers. In this case, the recovery time of a mispredicted branch would be one cycle, as desired. Unfortunately, the CAM logic scales worse than the mapping table. To begin with, a CAM cell is more expensive and slower than a normal RAM cell. In addition, the lookup array grows as the number of speculative registers increases. Therefore, it is desirable to have the significant performance benefits of the CAM lookup and the reasonably scalable table lookup.

As a compromise, we propose a CAM/table mapping hybrid technique. Figure 1 is a block diagram of the renaming hardware of such a hybrid technique. There are a limited number of CAM lookup entries, while the rest of the speculative registers are controlled by the recovery list and the mapping table. Source operands are first searched for matching entries in the CAM lookup list for the most recent destination register. If a match is not found, then the register is looked up in the mapping table. New destination registers enter the top of the lookup list and shift in a FIFO manner. When a destination register leaves the CAM lookup list, only then it is entered into the mapping table. Hence, when a mispredicted branch is encountered, if it is still in the CAM lookup entries, then there is no additional penalty. If it has been entered into the mapping table, then the recovery list is used to undo the mappings. In this situation there is a significant savings in the misprediction penalty, which is determined by the size of the CAM lookup.

To compare the performance benefit from a full mapping table, full CAM lookup, and hybrid, Table 2 lists the misprediction penalty and instructions per cycle (IPC) for CAM depths (number of entries divided by decode width) of 0, 2, 4, and 8. Of course, the simulator continued down the mispredicted path until it was resolved to determine the misprediction penalty.

We observe a significant performance improvement when the CAM lookup is used because the misprediction penalty is reduced. After about half the total depth ($CAM = 4$), there is a marginal improvement in performance compared to a full CAM lookup ($CAM = 8$). Therefore, we conclude that the hybrid CAM/table is a good compromise between cost and performance. Also, there might not be enough time to do a CAM lookup and table lookup serially, so it can be done in parallel at the expense of additional ports in the table.

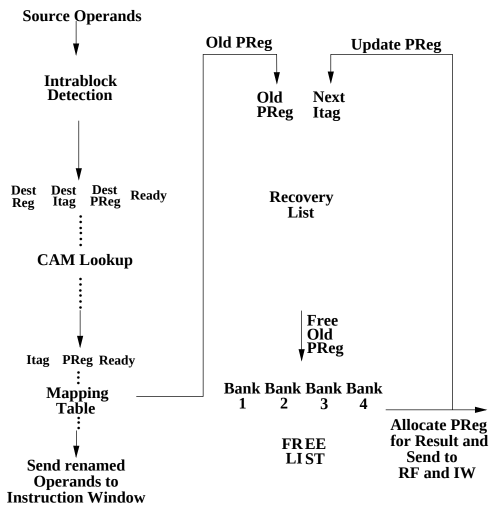

Figure 1. Block Diagram of Hybrid Renaming

3.2 Intrablock Decoding

Many operands are dependent on a result in the same block or in the recent past. For example, in a block of four instructions, each with one operand (since about half are usually constant or not used), about 1.2 operands are expected to be dependent within that block. If intrablock dependencies are exploited, the number of CAM ports required to search the lookup entries may be reduced. If there is not enough time in the decode stage to determine the intrablock dependencies, then pre-decode bits in the instruction block can be used. Each source operand in a block requires $lg(N)$ bits to encode which of the previous $N - 1$ instructions it is dependent on, or none at all. In addition, if an operand is dependent on an instruction which comes *before* the starting position of a block, the dependency information is ignored. When each line is brought into the instruction cache, or after the first access, the line containing a block of instructions is annotated with $2Nlg(N)$ bits (for two source operands) indicating if an instruction is dependent on another one in

Table 2. Bad Branch Penalty and Performance

Arch/Suite/	CAM=0		CAM=2		CAM=4		CAM=8	
Issue	Pen	IPC	Pen	IPC	Pen	IPC	Pen	IPC
SDSP/Int/4	5.7	2.63	4.4	2.70	3.8	2.74	3.5	2.75
SPARC/Int/4	6.0	2.20	4.6	2.29	4.0	2.32	3.7	2.35
SPARC/FP/4	6.1	1.58	4.8	1.61	4.1	1.63	3.8	1.64
SDSP/Int/8	6.8	3.70	5.4	3.85	4.7	3.93	4.2	3.99
SPARC/Int/8	8.0	2.70	6.4	2.84	5.5	2.91	5.0	2.96
SPARC/FP/8	8.6	1.91	7.1	1.96	6.1	2.00	5.4	2.03

the same block.

After running simulations using intrablock detection, we conclude decoding four instructions requires one less CAM port to search the lookup array for equivalent performance. Instead of needing five or six CAM ports, now the CAM lookup can use four or five. When the block size is doubled to eight instructions, we expect about four out of eight register operands to be intrablock dependent. Therefore, instead of doubling the number of CAM ports when the decode size doubles, an increase of only one port is needed – to about five or six.

If we refer back to Figure 1, the intrablock decoding now can be done before any CAM lookups or table mappings. As N increases, the number of operands needed for CAM and table lookup only increases slightly because it becomes more likely operands will be dependent within the same block. Hence, The hybrid CAM/table renaming scheme is scalable.

4 Register File Utilization

A disadvantage with allocating a physical register at decode time is that physical registers go unused until they receive their result value. As a result, a good portion of the register file is wasted most of the time. The total register file *utilization* is defined to be the ratio of the number of physical registers with a useful value and the total number of physical registers. In addition, the *speculative* register file utilization is the ratio of the number of physical registers with a useful speculative value and the total number of physical registers reserved for speculative results (does not include logical registers). A value is considered to be useful if it is needed to ensure proper execution of the machine. With speculative execution and precise interrupts, this occurs from the time a register receives its result until it is committed.

Table 3 shows the average speculative and total register file utilization per cycle for 4-way and 8-way superscalar processors. The mean, median, and 90^{th} percentile of the number of useful speculative registers are shown. Physical registers used to store the state of the logical register file will always be active, so the total register file utilization is not as meaningful as the speculative register file utilization.

Table 3. Average RF Utilization per Cycle

Arch/Suite/ Issue	% spec RF util	% total RF util	mean	median	90^{th} %
SDSP/Int/4	26.3	63.2	8.43	8	17
SPARC/Int/4	16.0	84.0	5.11	4	12
SPARC/FP/4	6.3	53.2	2.03	1	6
SDSP/Int/8	24.6	49.7	15.73	15	32
SPARC/Int/8	12.4	72.0	7.91	5	21
SPARC/FP/8	4.8	54.4	2.80	1	9

From the results presented, it is observed that less than one quarter of the registers reserved for speculative execution are used on the average. Less than half of the available speculative registers are used 90% of the time. The floating point RF used in the SPARC SPECfp95 has an extremely low utilization: less than 6%. Hence, the majority of speculative registers are going to waste most of the time.

5 Dynamic Result Renaming

As has been shown, many physical registers in the register file have no value or contain a useless value. Therefore, one way to reduce the size of the RF is to improve its utilization. Physical registers allocated with no value can be virtually eliminated by allocating at result write time instead of decode time. This is accomplished by splitting the register file into multiple banks, each bank with two read ports and one write port, as shown in Figure 2. Each bank maintains its own free list (see Figure 1), and old physical registers are freed when an instruction commits. In addition, a bank is directly connected to one result bus. When functional units arbitrate for result buses, each bank allocates an entry for a result. This cannot be done at decode time, since it is not known exactly which functional unit and bus a result will eventually arrive. On the other hand, by allocating the entry when results are written, multiple banks can be used with one write port and have no conflicts with writing results into the same bank. As a result of allocating physical registers at result write time, the size of each bank can remain constant as the number of banks increase proportionally to the issue width.

Although allocating physical registers at write time creates no conflicts for the single write port in a bank, the two read ports on one bank can cause contention with the dispatch queues/instruction window. For example, three ALUs could require three operands from a single bank. With only two read ports, one ALU would not be able to issue its instruction. Even though this event can happen, it is not a likely event for two reasons. First, not every instruction issued requires two register operands. Some have one operand, while others require an immediate value. Second, most instructions issued bypass one of their results from the result of an instruction completed the previous cycle. Consequently, such a limited number of read ports per bank has a very limited impact on performance.

Table 4 demonstrates this fact by showing the distribution of read operand types, and the percentage of individual operand requests failed due to insufficient ports. If the operand is a register, then it can originate from the first or second level of bypassing, the first or second read port of a bank, or be an identical register read from the first or second read port. On the other hand, if it is not a register operand, then it can be an immediate value, zero value, or no operand

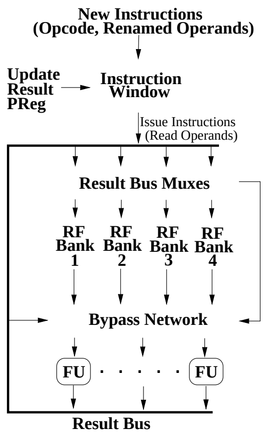

Figure 2. Diagram of Multiple Banked RF

at all. Interestingly, although a significant percentage of register operands came from the first read port, few required the second read port. Furthermore, a large percentage of registers are bypassed, especially at the first level. Since many operands are bypassed, a traditional RF for map on decode could reduce the number of read ports by about 50%. The number of write ports, however, must remain the same. Consequently, although the size of its RF may be reduced, this does not lead to a *scalable* solution.

The allocation of registers for results is pipelined. Two cycles before the result will be ready for writing, write arbitration takes place and entries for the corresponding result buses/banks are allocated. If allocation fails, then the pipeline for the functional unit which is writing to that bank is stalled. Also, another instruction may be issued to the functional unit before the instruction window is notified that the functional unit is stalled. Consequently, two instructions may be waiting in the functional unit's pipeline. This does not create a problem since already two levels of bypassing exist. If subsequent instructions require a stalled result, then the result continues to use the bypass network until it is written to the register file.

In order to be able to allocate registers at result write time and be able to do register renaming for out-of-order execution, two types of renaming must take place. Before an instruction is placed into the instruction window, its destination operand is renamed to a unique *instruction* tag (*itag*) and inserted into the mapping table. This contrasts to renaming the register to a physical register since allocation has not taken place yet. After the physical register has been allocated for the result, then the instruction window, mapping table, and recovery list need to be notified of the physical register (*preg*). Using the result's destination register identifier, the mapping table is updated, if necessary. Matching *itag* entries in the instruction window receive the *preg*. Entries in the instruction window are already matched to mark its operand ready, so there is little additional cost involved besides storage cost. The recovery list may contain an entry with an invalid old *preg* because its value is not ready. As a result, there needs to be some way of finding that entry so it can be updated. One way would be to do a CAM matching based on the *itag*. A more efficient mechanism would be to store the next *itag* in the recovery list. When the entry in the recovery list is marked complete, the next *itag* is read, and the *preg* is written into the entry index by the next *itag*.

The greatest cost in hardware using dynamic result renaming is the full multiplexer network used for reading and writing registers. This cost, however, does not begin to compare to the enormous time and space savings by using a two read port and one write port register file. The bypass network is still required for map on decode case. In addition, the bypass network might be reduced by implemented suggestions by Ahuja et. al. [1].

Table 4. Read Operand Category Distrib. (%)

	4 Issue			8 Issue		
Architecture	SDSP	SPARC		SDSP	SPARC	
SPEC	Int	Int	FP	Int	Int	FP
Bypass L1	25.1	19.7	19.4	19.7	13.8	21.5
Bypass L2	4.8	4.3	2.6	4.3	2.9	2.3
Read P1	13.2	18.1	20.4	18.1	9.6	18.8
Read P2	0.8	1.5	5.2	1.5	1.1	3.4
Identical	0.3	0.6	1.6	0.6	1.0	2.5
Zero Val	9.3	9.3	8.7	9.3	13.8	8.7
Imm. Val	32.7	32.7	38.8	32.7	54.5	38.9
No Operand	13.8	13.8	3.3	13.8	3.5	4.0
Failed Read	0.2	0.2	7.3	0.2	7.4	5.1

5.1 Deadlocks

The most critical aspect of using dynamic result renaming is avoiding deadlock situations. If there are fewer speculative registers than entries in the recovery list, then it is possible all registers can be allocated with results still pending and create a deadlock situation. To guarantee a deadlock will not occur, two conditions must exist: the oldest instruction must be able to issue its instruction, and the oldest instruction must be able to write its result.

It may occur that the oldest instruction is not able to issue its instruction if the functional unit it requires is stalled and there are no free registers available. Therefore, if this situation arises, the oldest instruction and only the oldest instruction is permitted to issue its instruction to a functional unit whose results are stalled and latched into its two stages. When it completes, the result is delivered to the register file and written using the old physical register stored in the recovery list (thereby guaranteeing the second condition).

Moreover, if the oldest instruction is unable to issue or complete its instruction, then the processor temporarily executes in a scalar manner. In addition, each functional unit must have access to enough banks that equal to at least the number of entries in the recovery list plus the number of logical registers (in most situations, the entire register file).

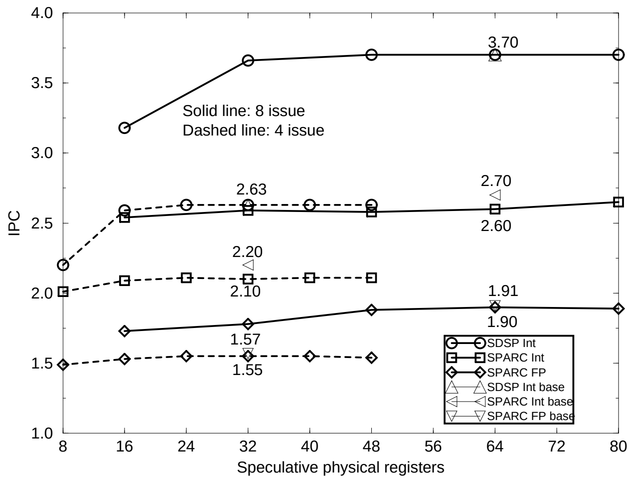

Figure 3. RF Performance Comparison

5.2 Performance

The performance of dynamic result mapping is compared to the performance of mapping during decoding. The number of speculative physical registers is varied. Figure 3 shows the IPC for a 4-way and 8-way issue processors. The map on decode (referred as the base case) does not need to vary the speculative physical registers because the recovery list is constant at 32 (or 64) entries and requires a fixed amount. On the other hand, map on write can be affected by more or less registers than this amount.

The performance difference from the base case is negligible, except for SPECint95 on the SPARC architecture, where a 5% decrease is observed. Increasing the number of physical registers reduced this gap for the 8-way issue processor but did not help much for the 4-way issue processor.

The reason why a slight performance decrease is observed in the SPARC architecture and not in the SDSP architecture is the difference in the number of logical integer registers. The ratio of logical to speculative registers in the SDSP is 1:1 while the ratio is 4.25 for the SPARC (136 integer registers) for a 4-way issue processor. This ratio is cut in half when the decode width and recovery list are doubled. When there is a large ratio, the speculative registers have a difficult time competing against logical registers in a bank. Logical registers can pool into one particular bank, thereby restricting its usage. Registers will then tend to be allocated from a bank with most of the free registers, and functional units will stall since writing becomes limited. This performance problem can be avoided by reducing the ratio and/or increasing the number of banks.

The total and speculative utilization of the register file increases, and the number of speculative registers is reduced by 25% to 50%, with none or a small performance decrease. Sometimes the performance can actually increase by decreasing the number of registers. This is not due to the renaming, but from a slight reduction in the misprediction penalty.

6 Conclusion

A new scalable register file architecture was introduced that is suitable for wide-issue, dynamically scheduled processors. This was accomplished by using a multiple banked register file and binding the result to a physical register at the write stage. Our simulation results show the proposed technique did not have any significant performance drawbacks compared to mapping at instruction decode. In addition, we proposed a hybrid technique of a CAM lookup list and a mapping table to significantly reduce the branch misprediction penalty and still tolerate a large number of speculative registers.

References

[1] P. Ahuja, D. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *28th Annual International Symposium on Microarchitecture*, November 1995.

[2] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1992.

[3] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS*, 1994.

[4] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *2nd International Symposium on High-Performance Computer Architecture*, pages 40–51, February 1996.

[5] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.

[6] D. B. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, pages 8–15, April 1996.

[7] S. P. Song, M. Denman, and J. Chang. The Power PC 604 RISC microprocessor. *IEEE Micro*, pages 8–17, October 1994.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choise: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[9] S. Wallace and N. Bagherzadeh. Performance issues of a superscalar microprocessor. *Microprocessors and Microsystems*, 19(4):187–199, May 1995.

[10] K. C. Yeager. MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1996.

[11] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Cost, Australia, May 1992.