In Proceedings of the 4th International Symposium on High Performance Computing (ISHPC), May 2002, (c) Springer-Verlag.

An EPIC Processor with Pending Functional Units

Lori Carter, Weihaw Chuang and Brad Calder
Department of Computer Science and Engineering
University of California, San Diego
{lcarter,wchuang,calder}@cs.ucsd.edu

Abstract

The Itanium processor, an implementation of an Explicitly Parallel Instruction Computing (EPIC) architecture, is an in-order processor that fetches, executes, and forwards results to functional units in-order. The architecture relies heavily on the compiler to expose Instruction Level Parallelism (ILP) to avoid stalls created by in-order processing.

The goal of this paper is to examine, in small steps, changing the in-order Itanium processor model to allow execution to be performed out-of-order. The purpose is to overcome memory and functional unit latencies. To accomplish this, we consider an architecture with *Pending Functional Units* (PFU). The PFU architecture assigns/schedules instructions to functional units in-order. Instructions sit at the pending functional units until their operands become ready and then execute out-of-order. While an instruction is pending at a functional unit, no other instruction can be scheduled to that functional unit. We examine several PFU architecture designs. The minimal design does not perform renaming, and only supports bypassing of non-speculative result values. We then examine making PFU more aggressive by supporting speculative register state, and then finally by adding in register renaming. We show that the minimal PFU architecture provides on average an 18% speedup over an in-order EPIC processor and produces up to half of the speedup that would be gained using a full out-of-order architecture.

1 Introduction

The Itanium processor, the first implementation of the IA64 Instruction Set Architecture, relies on the compiler to control the scheduling of instructions through the processor, which reduces the complexity and increases the scalability of the processor. In-order processing has severe IPC performance limits due to the inability to allow execution to continue past an instruction with an outstanding register use, where the register is being produced by a long latency instruction currently executing. In this situation the whole front-end of the processor stalls and it cannot issue any more instructions until the oldest instruction in the issue window has both of its operands ready. Out-of-order processors have the capability to allow execution to continue, but at the cost of increased hardware complexity.

This paper describes our work in examining how to adapt the Itanium processor model to overcome the limitations caused by unpredictable load latencies as well as long latency instructions that the compiler could not hide. Our goal is to examine moving towards an out-of-order processor without adding a large amount of additional complexity. Rau [8] suggested the idea of small-scale reordering on VLIW processors to support object code compatibility across a family of processors. We describe a method that makes use of features that currently exist in the Itanium model.

In the Itanium, each cycle can have up to 6 instructions scheduled to proceed down through the pipeline and begin executing together. If there is an outstanding dependency for a member of the scheduled group,

then all of the instructions in that group stall at the functional units and wait there until all instructions in that scheduled group can start to execute at the same time. To facilitate this, the Itanium already has functional units that have bypassing logic to allow the values being produced to be directly consumed in the next cycle by another functional unit.

In this paper, we examine *Pending Functional Units* (PFU) that expose a small window of instructions (those that have been allocated function units) to be executed out-of-order. The PFU model implements a simple in-order instruction scheduler. Instructions in this new architecture are issued exactly the same as in a traditional in-order VLIW architecture (instructions cannot issue if there are WAW dependencies). In this paper we examine three PFU models. Each model increases the complexity of the processor, but also increases the amount of ILP exposed. All of these form their scheduled group of instructions in-order, and the only instruction window that is exposed for out-of-order execution are the instructions pending at the functional units.

2 Pending Functional Units

In this section we describe the baseline in-order processor modeled, along with the different variations of the PFU architecture.

2.1 Baseline In-Order EPIC Processor

We model the Itanium processor as described in [9]. The IA64 ISA instructions are *bundled* into groups of three by the compiler. A template included with the bundle is used to describe to the hardware the combination of functional units required to execute the operations in the bundle. The template will also provide information on the location of the stop bits in the bundle. Like Itanium, our *baseline in-order EPIC processor* has up to two bundles directed or *dispersed* [2] to the functional units in the EXP stage, subject to independence and resource constraints. All instructions dispersed are guaranteed to be independent of each other. We use the bundles and the stop bits to determine this independence. We call this in-order group of instructions sent to the functional units together a *scheduled group* of instructions. If any of the instructions in the scheduled group must stall because a data dependence on an instruction outside of this group is not yet satisfied, the rest of the instructions in the scheduled group will stall in the functional units as well.

2.2 Limited Out-Of-Order Execution Provided by Pending Functional Units

The goal of this research is to examine, in increasing complexity, the changes needed to allow the Itanium processor to execute out-of-order, for the purpose of overcoming memory and long latency functional unit delays. We provide this capability by turning the functional units of the Itanium processor into *Pending Functional Units*. These are similar to reservation stations [12], but are simpler in that no scheduling needs to be performed when the operands are ready, because the instruction already owns the functional unit it will use to execute. We limit our architecture to 9 functional units in this paper, since we model the PFU architecture after the Itanium processor.

In providing the PFU limited out-of-order model, we need to address how we are going to deal with the following three areas:

1. **Register Mapping.** The Itanium processor provides register renaming for rotating registers and the stack engine, but not for the remaining registers. We examine two models for dealing with registers for the PFU architecture. The first approach does not rename any registers, and instead the scheduler makes sure there are no WAW dependencies before an instruction can be issued in-order to a functional

unit. Note, that WAR dependencies do not stall the issuing of instructions, since operands are read in-order from the register file and there can be only one outstanding write of a given register at a time. The second approach we examine allows multiple outstanding writes to begin executing at the same time by using traditional out-of-order hardware renaming.

2. **Scheduling.** When forming a schedule for the baseline Itanium processor, the dispersal stage does not need to take into consideration what has been scheduled before it in terms of resource constraints. In this model, the whole scheduled group goes to the functional units and they either all stall together, or all start executing together. For the PFU architecture, the formation of the scheduled group of instructions needs to now take into consideration functional unit resource constraints, since the functional unit may have a pending instruction at it. The scheduling of instructions (assigning them to functional units) is still performed in-order. To model the additional complexity, we add an additional scheduling pipeline stage (described in more detail below) to the PFU architecture. It is used to form the scheduled groups taking into consideration these constraints. Flush replay is used when the schedule is incorrect.

3. **Speculative State.** The Itanium processor can have instructions complete execution out-of-order. To deal with speculative state, the Itanium processor does not forward/bypass its result values until they are non-speculative. For our PFU architecture we examine two techniques for dealing with speculative state. The first approach is similar to the Itanium processor, where we do not forward the values until they are known to be non-speculative. The second approach examines using speculative register state (as in a traditional out-of-order processor), and updating the non-speculative state at commit. This allows bypassing speculative state at the earliest possible moment – as soon as an instruction completes execution.

2.3 Instruction Scheduling

As stated above, for the Itanium processor, the formation of the scheduled group of instructions occurred without caring about the state of the functional units. Therefore, all of the functional units are either stalled or available for execution in the next cycle. This simplifies the formation of execution groups in the dispersal pipeline stage of Itanium.

In our PFU architecture we add an additional pipeline stage to model the scheduling complexity of dealing with pending functional units. The scheduler we use includes instructions *in-order* into a scheduled group and *stops* when any of the three conditions occur:

- An instruction with an unresolved WAW dependency is encountered.
- There will be no free functional unit N cycles into the future (N is 2 for the architecture we model) for the next instruction to be included into the group.
- The scheduled group size already includes 6 instructions, which is the maximum size of a scheduled group.

The second rule above is accomplished by keeping track of when each functional unit will be free for execution. This is the same as keeping track, for each instruction pending at a functional unit, when the operands for that instruction will be ready to execute. All functional units are pipelined, so the only time a functional unit will not be free is if the instruction currently pending at the functional unit has an unresolved dependency. The only dependencies that are non-deterministic are load's, which result in cache misses. The schedule is formed assuming that each load will hit in the cache.

When the schedule is wrong a replay needs to occur to form a legal scheduled group of instructions. The Alpha 21264 processor uses *Flush Replay* [5] (also referred to as squash replay) for schedule recovery. If one of the instructions in the scheduled group of instructions cannot be assigned because a functional unit is occupied, up to *X* groups of instructions scheduled directly after this scheduled group are flushed, and a new schedule is issued. The *X* cycle penalty is determined by the number of cycles between formation of the schedule and the execution stage. In this paper we assume that the penalty is two cycles.

2.4 Managing Speculative State

The out-of-order completion of instructions creates result values that are speculative and they may or may not be used depending upon the outcome of a prior unresolved branch.

The baseline EPIC processor we model does not forward its result values until they are non-speculative. Therefore, its pipeline up through the start of execution does not have to deal with speculative result values.

In the PFU architecture we examine two methods of dealing with speculative result values:

- Non-Speculative - The first approach is similar to the Itanium processor, where we do not forward the values until they are known to be non-speculative.
- Speculative - The second approach forwards the values in writeback right after the instruction finishes execution. This means that the register state dealt with in the earlier part of the pipeline is considered speculative register state (as in a traditional out-of-order processor), and the non-speculative register state is updated at commit. Note, multiple definitions are not an issue with speculative bypassing, since WAW dependencies must be resolved prior to an instruction entering a functional unit.

2.5 Summary of PFU Configurations Examined

The above variations are examined in 3 PFU configurations, which we summarize below:

- PFU - The base PFU configuration adds one stage to the pipeline for in-order scheduling as described earlier, and flush replay occurs when a functional unit predicted to be idle in the schedule has a pending instruction at it. It uses PFUs with no renaming, following the example of Thorton's simplification [11] of Tomasulo's early out-of-order execution design [12]. This configuration only forwards results and updates register state when they are known to be non-speculative.
- PFU Spec - Builds on PFU allowing bypassing of values as soon as they finish execution. The processor therefore needs to support speculative and non-speculative register state.
- PFU Spec Rename - Builds on PFU Spec adding traditional hardware renaming for predicates and computation registers. This eliminates having to stop forming scheduled group of instructions because of a WAW dependency.

3 Methodology

We created an EPIC simulator by modifying SimpleScalar [4]. It performs its simulation using IA64 traces. We extended the baseline SimpleScalar to model the Itanium stages and to simulate the IA64 ISA. Our extension focused on 5 areas. First, we added the ability to execute in-order and include the detection and enforcement of false WAW dependencies. Second, the simulator had to be extended to support predicated execution. One of the most significant changes in this area was the need to include the possibility of multiple definitions for the use of a register due to the fact that the same register could be defined along multiple paths

benchmark	suite	end init	begin trace	description
go	95 C	6600	9900	Game playing; artificial intelligence
ijpeg	95 C	4900	7200	Imaging
li	95 C	1200	1800	Language interpreter
perl	2k C	4900	7200	Shell interpreter
bzip2	2k C	400	600	Compression
mcf	2k C	1300	1900	Combinatorial Optimization

Table 1: Presents description of benchmarks simulated including instruction count in millions where the initialization phase of the execution ended and where the trace began recording.

that are joined into one through if-conversion. The IA64 supports software pipelining, so we implemented the functionality of rotating registers along with specialized instructions that implicitly re-define predicate registers. Another important feature of the IA64 ISA is its support for control and data speculation. To support this feature, we modeled the implementation of the ALAT, with its related operations and penalties. Finally, we added the ability to detect bundles and stop bits and appropriately issue instructions using this information.

Our traces are generated on IA64 machines running Linux through the ptrace system interface [6]. This allows a parent program to single-step a spawned child. Each trace is built using the *ref* data set, and includes 300 million instructions collected well after the initialization phase of the execution has finished. Table 1 shows the end of the initialization phase and the beginning of the trace in millions of instructions. The end of the initialization phase was determined using the Basic Block Distribution Analysis tool [10].

For each instruction in the trace range, we record the information necessary to simulate the machine state. For all instructions, we record in the trace file the instruction pointer, the current frame marker [1] and the predicate registers. In addition, we record the effective-address for memory operations and the previous function state for return. IA64SimpleScalar then reads in the trace file to simulate the program's execution.

IA64SimpleScalar decodes the traces using an IA64 opcode library containing a record for each IA64 instruction, and a library that interprets each instruction including templates and stop bits. This was built from the GNU opcode library that contains opcode masks to match the instruction, operand descriptions, mnemonic, and type classification. We enhanced this by adding a unique instruction identifier, quantity of register writers, and target Itanium functional units.

In this paper we used a number of SpecInt2000 and SpecInt95 benchmarks, four of which were compiled using the Intel IA64 electron compiler (go, ijpeg, mcf, bzip2), and two were compiled using the SGI IA64 compiler (li,perl). We used the "-O2" compilation option with static profile information. This level of optimization enables inline expansion of library functions, but confines optimizations to the procedural level. It produces a set of instructions in which an average of 24% found in the trace were predicated, with an average of 4.3% predicated for reasons other than branch implementation and software pipelining. This can arise due to if-conversion, or predicate definitions for use in floating point approximation. Table 2 shows the parameters we used for the simulated microarchitecture, which were modeled after the Itanium.

4 Results

This section presents results for adding PFUs to an EPIC in-order processor, and then compares the performance of this architecture to a traditional out-of-order processor.

L1 I Cache	16k 4-way set-associative, 32 byte blocks, 2 pipeline cycles
L1 D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	96k 6-way set-associative, 64 byte blocks, 6 cycle latency
Unified L3 Cache	2Meg direct mapped, 64 byte blocks, 21 cycle latency
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Functional Units	2-integer ALU, 2-load/store units, 2-FP units, 3-branch
DTLB, ITLB	4K byte pages, fully associative, 64 entries, 15 cycle latency
Branch Predictor	meta-chooser predictor that chooses between bimodal and 2-level gshare, each table has 4096 entries
BTB	4096 entries, 4-way set-associative

Table 2: Baseline Simulation Model.

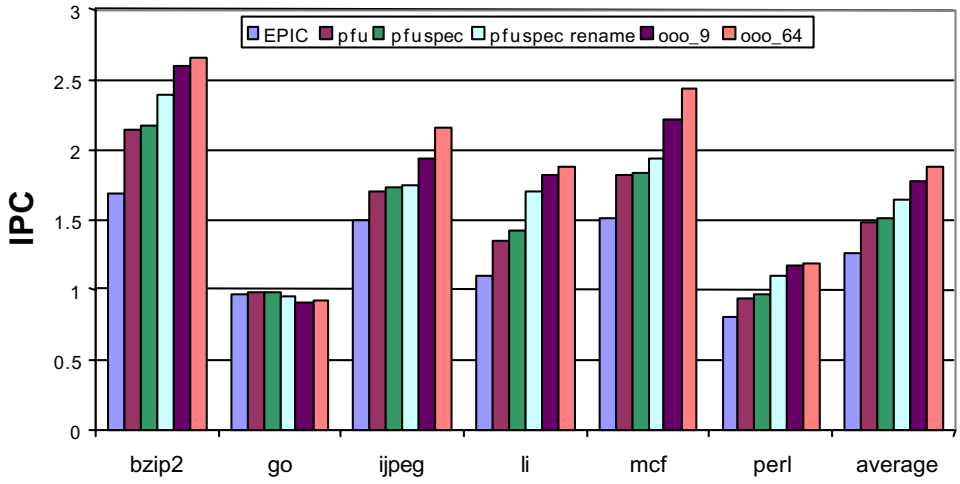

Figure 1: Comparing the IPCs of the various PFU models and the out-of-order models with issue buffer sizes of 9 and 64 to the EPIC model.

4.1 Adding PFUs to an EPIC In-Order Processor

Figure 1 shows the performance benefit of allowing the in-order EPIC processor to execute out-of-order using Pending Functional Units. It shows IPC results for the baseline EPIC processor, the various PFU configurations, and a full out-of-order processor with issue buffer sizes of 9 and 64.

The improvement for using the minimal PFU architecture over the baseline EPIC processor ranged from 1% (`go`) to 27% (`bzip2`). The average improvement of the PFU model over the EPIC model was 18%. The low IPC and limited improvement seen for `go` can be attributed to poor cache and branch prediction performance as shown in Table 3. This table describes, for each benchmark, information about cache and branch predictor activity for the baseline EPIC configuration. In particular, the second column describes the number of load misses that occurred in the L1 data cache per 1000 executed instructions. The third column is the number of instructions that missed in the L1 instruction cache per 1000 executed instructions. The next two columns refer to the L2 and L3 caches, which are unified. The last column provides the branch prediction hit rate produced by the number of hits divided by the number of updates to the predictor. The predictor is updated for every non-speculative branch executed. As can be seen from this table, the branch prediction accuracy for `go` was significantly lower than that of the other benchmarks at 69%. In addition, `go` also had a high L1 instruction cache miss rate, where there were 48 misses for every 1000 executed

benchmark	dl1 miss/1K	il1 miss/1K	ul2 miss/1K	ul3 miss/1K	Br Pred Hit Rate
bzip2	.279	.005	.138	.131	91%
go	11.242	47.930	13.592	1.771	69%
ijpeg	2.629	21.193	1.058	.493	87%
li	6.336	3.451	3.066	.071	83%
perl	11.780	41.791	13.327	2.357	83%
mcf	30.949	.002	32.017	15.893	84%

Table 3: Presents misses per 1K instructions in the various caches. Levels 2 and 3 are unified caches. The last column shows the branch prediction accuracy.

instructions.

The second configuration considered is PFU Spec. This approach adds an additional 2% improvement, bringing the average improvement in IPC over the EPIC model up to 20%. The largest additional improvement achieved using speculative state is seen in the `li` benchmark, at 6%.

For the final PFU configuration, we added full register renaming to the model just described. On average, renaming improved the performance of the previous model by 10% as compared to the baseline EPIC, bringing the total average improvement to 30%. `Li` saw the biggest improvement over EPIC with a gain in IPC of 56%. In this case, the IPC produced by `go` was actually lower than the baseline EPIC processor. This is due to the additional pipeline stages, which we modeled in the PFU architecture, resulting in increased penalties related to branch mis-predictions. Hence, the poor prediction accuracy recorded by `go` translated into reduced IPC.

4.2 Comparing to a Complete Out-of-Order Processor

We now compare the PFU architecture to an out-of-order IA64 processor. The out-of-order processor we model has three major architectural changes over the baseline in-order EPIC architecture:

1. **Out of Order instruction scheduler**, selecting instructions from a window of 9 or 64 instructions to feed the functional units. Instead of using Flush Replay, modern out-of-order processors use *Selective Replay* to avoid large penalties for forming incorrect schedules often caused by load misses. The scheduler in the Pentium 4 [7] uses a replay mechanism to selectively re-issue only those instructions that are dependent on the mis-scheduled instruction. This adds more complexity in comparison to using flush replay as in the PFU architecture. We modeled flush replay for out-of-order execution, but the IPC results were the same or worse on average than the PFU architecture. Therefore, we only present out-of-order results in this paper for selective reply.

2. **Register renaming** to eliminate WAW dependencies

3. **Out-of-order completion** of execution, so speculative register state needs to be maintained (a reorder buffer or register pool).

An out-of-order pipeline architecture tries to make use of the available run-time information and schedules instructions dynamically. In an out-of-order environment, instructions effectively wait to execute in either an issue buffer, active list of instructions, or reservation stations. We use the notion of an issue buffer that is dynamically scheduled to supply instructions to all functional units. In this section, we examine

configurations of an out-of-order architecture issue buffer sizes of 9 and 64. We chose 9 in order to make a comparison to the PFU model with 9 pending functional units.

The main difference between PFU spec rename and the OOO architecture is that PFU spec rename forms its schedule in-order constrained by predicting if there is going to be a free functional unit for the next instruction to be included into the schedule. In comparison, the OOO architecture forms its schedule picking any instruction from the 9 or 64 oldest un-issued instructions that it predicts will have their operands ready by the time they reach the functional units.

Our out-of-order version of the EPIC pipeline varies from the traditional out-of-order architecture only as was required to support EPIC features such as predication and speculation. For example, in the renaming process, each definition of a register is given a unique name, and then that name is propagated to each use. However, predication allows for multiple possible reaching definitions. Consequently, this process must be extended to include knowledge of guarding predicates in conjunction with definitions. We implement out-of-order renaming as described in [13]. As in that study, we add 4 stages to the baseline EPIC pipeline, 2 for renaming, and 2 for scheduling, to model the additional complexity for out-of-order execution.

Figure 1, provides IPC results for the 2 different configurations that were evaluated for the out-of-order model. As mentioned, these varied by the size of the schedulable window of instructions. In the out-of-order architecture, up to six instructions can be scheduled/issued to potentially 9 functional units. The results show that an issue window of 64 instructions produces an additional speedup of 31% over the base PFU configuration. For `bzip2`, the baseline PFU architecture experienced half of the improvement seen by the full out-of-order model. `Go` is again an interesting case. The out-of-order models perform worse than all other configurations. This is attributed to the high penalties due to a deeper pipeline and poor branch prediction.

For several of the programs, the out-of-order model with an issue window of 9 performs only slightly worse than the out-of-order model with 64. For the benchmarks `bzip2`, `go`, `li`, and `perl`, the difference in IPC was under 3%. The average difference in IPC between the two out-of-order configurations was only 5%. For `bzip2`, `ijpeg` and `perl`, the baseline PFU architecture achieves half the performance improvement that is seen by the aggressive OOO model with a window of 9 instructions.

5 Conclusions

In-order processors are advantageous because of their simple and scalable processor pipeline design. Performance of the program is heavily dependent upon the compiler for scheduling, load speculation, and predication to expose ILP. Even with an aggressive compiler, the amount of ILP will still be constrained for many applications.

To address this problem we explored adding the ability to start executing instructions out-of-order to an in-order scheduled processor. We examined the Pending Functional Unit (PFU) architecture, where instructions are allocated to the functional units in order (as in the Itanium), but they can execute out-of-order as their operands become ready. Augmenting the EPIC in-order processor with PFUs provides an average 18% speedup over the EPIC in-order processor. The additional cost in complexity to achieve this speedup is the dynamic in-order scheduling of the functional units and the implementation of flush replay for when the schedule is incorrect.

When comparing the base PFU architecture to a full out-of-order processor, our results show that the PFU architecture yields 44% of the speedup, on average, of an out-of-order processor with an issue window of 9 instructions and renaming. The out-of-order processor with a window of 9 instructions achieved results within 5% of having a window of 64 instructions.

Executing instructions in a small out-of-order window is ideal for hiding L1 cache misses, and long latency functional unit delays. Another way to attack these stalls would be to increase the amount of prefetch-

ing performed by the processor, and reduce the latency of the L1 cache and functional units. The plans for the next generation IA64 processor, McKinley, includes reducing latencies, increasing bandwidth, and larger caches [3]. While latency reduction via prefetching and memory optimizations will provide benefits, a small out-of-order will most likely still be advantageous, since not all L1 misses will be able to be eliminated. Performing this comparison is part of future work.

Acknowledgements

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF grant No. 0073551, a grant from Intel Corporation, and an equipment grant from Hewlett Packard and Intel Corporation. We would like to thank Carole Dulong and Harpreet Chadha at Intel for their assistance bringing up the Intel IA64 Electron compiler.

References

[1] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.

[2] Itanium Processor Microarchitecture Reference:for Software Optimization., 2000. http: //www.developer.intel.com/design/ia64/itanium.htm.

[3] 2001 - a processor odyssey: the first ever McKinley processor is demonstrated by hp and Intel at Intel's developer forum, February 2001. http: //www.hp.com/products1/itanium/news_events/archives/NIL0014KJ.html.

[4] D.C. Burger and T.M. Austin. The Simplescalar Tool Set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, Jun 1997.

[5] COMPAQ Computer Corp. Alpha 21264 microprocessor hardware reference manual, July 1999.

[6] S. Eranian and D. Mosberger. The Linux/IA64 Project: Kernel Design and Status Update. Technical Report HPL-2000-85, HP Labs, June 2000.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[8] B. R. Rau. Dynamically scheduled vliw processors. In *Proceedings of the 26th Annual Intl. Symp. on Microarchitecture*, pages 80–92, December 1993.

[9] H. Sharangpani and K. Arora. Itanium processor microarchitecture. In *IEEE MICRO*, pages 24–43, 2000.

[10] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[11] J. E. Thornton. Design of a Computer, the Control Data 6600, 1970. Scott, Foresman, Glenview, Ill.

[12] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[13] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, February 2001.