Rapid Development of a Flexible Validated Processor Model

David A. Penry David I. August
Department of Computer Science
Princeton University
Princeton, NJ 08544
{dpenry,august}@princeton.edu

Manish Vachharajani
Dept. of Electrical and Computer Engineering
University of Colorado at Boulder
Boulder, CO 80309
manishv@colorado.edu

Abstract

Given the central role of simulation in procesor design and research, an accurate, validated, and easily modified simulation model is extremely desirable. Prior work proposed a modeling methodology with the claim that it allows rapid construction of flexible validated models. In this paper, we present our experience using this methodology to construct a flexible validated model of Intel's Itanium 2 processor, lending support to their claims. Our initial model was constructed by a single researcher in only 11 weeks and predicts processor cycles-per-instruction (CPI) to within 7.9% on average for the entire SPEC CINT2000 benchmark suite. We find that aggregate accuracy for a metric like CPI is not sufficient; aggregate measures like CPI may conceal remaining internal "offsetting errors" which can adversely affect conclusions drawn from the model. We then modified the model to reduce error in specific performance constituents. In $2\frac{1}{2}$ person-weeks, overall constituent error was reduced from 3.1% to 2.1%, while simultaneously reducing average aggregate CPI error to 5.4%, demonstrating that model flexibility allows rapid improvements to accuracy. Flexibility is further shown by making significant changes to the model in under eight person-weeks to explore two novel microarchitectural techniques.

1. Introduction

Simulation is the preferred method of measurement for much of the computer architecture community. For simulation, computer architects would prefer to use *validated* models of realized systems over *non-validated* models for several reasons. First, a real system provides a golden standard which the community can use as a point of reference. Second, the existence of the real system provides proof that the model is implementable and suggests that reasonable variants are implementable as well. Finally, and perhaps most importantly, a validated model provides more confidence in conclusions drawn using the model. Prior to validation, models often fail to represent important interactions between different parts of the system and fail to capture surprising corner case behavior. Previous works such as those by Black and Shen [2], Gibson et al. [6], and Desikan et al. [3] have shown that this effect can be significant and that non-validated models can lead to incorrect conclusions.

Unfortunately, computer architects tend not to use validated models for several practical reasons. First, validated models can be extremely time-consuming to construct because constructing highly detailed models is difficult [2]. Second, even if simulators were easy to construct, not enough information about a design may be publicly available to decide what to construct [7]. Finally, validated models are often so detailed that they may be too time-consuming to modify for initial studies of many different design points [4]. This may impede innovation if it becomes difficult to study design points not immediately adjacent to the reference machine.

Prior work claims that validated models need not be hard to construct and need not be difficult to modify to explore wide areas of the design space. To support this claim, prior work presented a modeling methodology (which we will call the Liberty Modeling Methodology) [17, 16]. This modeling methodology is centered around three modeling principles:

1. structural modeling of the system
2. aggressive reuse of model components
3. iterative refinement of the model

They have also released a modeling framework consisting of a structural modeling language and a simulator-constructing compiler to support these principles called the Liberty Simulation Environment (LSE). However, their claims have not yet been substantiated with a complete validated model.

In this paper, we present our experience building a validated Itanium 2 model using the Liberty methodology and tools. This experience serves as an instance proof that it

is indeed possible to rapidly construct an easily-modifiable *validated* processor model. Using LSE, a lone computer architect was able to construct an initial validated model of Intel's Itanium 2 processor in only 11 weeks including the time to reverse engineer the physical hardware. This model predicted hardware cycles-per-instruction (CPI) to 7.9% with a maximum error of 20% across all SPEC CINT2000 benchmarks.

During this investigation, we discovered that traditional metrics of validated model *quality* are inadequate. This paper shows that models validated against a single aggregate metric, such as CPI, are insufficient for proper design-space exploration since the model may still contain large internal error constituents. With *aggregate validation*, internal error constituents may simply offset each other. We show that such errors exist in our initial model and that these "offsetting errors" can lead to poor design decisions. To correct these errors, we refined our model until it was validated against the hardware for multiple constituent metrics. This refinement took an additional 2.5 person-weeks and resulted in the current *constituent-validated* model that predicted CPI to within 5.4% across all SPEC CINT2000 benchmarks and contained far fewer offsetting errors.

To further assess the model's flexibility, we constructed two novel derivatives — an Itanium 2 with a variable latency tolerance technique and an Itanium 2 CMP processor with an unconventional interconnection mechanism [14].

The remainder of this paper is organized as follows. Section 2 describes the Liberty Modeling Methodology. Section 3 describes the Itanium 2. Section 4 then describes our first experience using the Liberty Modeling Methodology to build a validated model of the Itanium 2 and presents data regarding the aggregate quality of the model. Section 5 describes why, despite low CPI error, an aggregate-validated model may be unsuitable for microarchitecture research. Section 6 describes how we refined our initial model using *constituent validation* to correct this shortcoming and gives results. Section 7 describes experience modifying the model to explore novel ideas. Section 8 concludes.

2. The Modeling Methodology

To build a validated model, one must be able to control the sources of significant error in a system. Black and Shen identify three such sources of error in performance models [2]. These sources of error are:

Specification errors One does not fully understand the system being modeled and so models the wrong system.

Modeling errors Mistakes are made while incorporating understood system behavior into the model. The model does not do what one thinks it does.

Abstraction errors One deliberately decides not to model some behavior accurately, either by leaving out the behavior entirely or by not modeling all of its details.

Prior work describes a modeling methodology, which we will call the Liberty Modeling Methodology, with the claim that it allows rapid construction of validated models [17, 16]. The Liberty Modeling Methodology is centered around three modeling principles, each of which they claim has a role in ensuring a reduced error rate and an improved time-to-model. Here is brief summary of these principles and the role they play:

Structural system modeling A hardware system design is conceived as hierarchically composed concurrently executing blocks. Attempts to map this hierarchical and concurrent system to another composition strategy, such as composition via procedure invocation in C or C++, naturally introduce modeling errors [17]. As a result, the modeling environment should be concurrent and structural. This feature is also key to reducing model specification time because it simplifies the mapping of hardware design to model *and* because it is the key enabler for component reuse [17].

Aggressive component reuse By aggressively reusing model components, the cost of building a component is amortized across many different designs, reducing total exploration time. Reuse also has the side benefit of reducing error rates because basic behaviors are specified once and validated in many different models. This limits the source of modeling errors to incorrect component usage and eliminates the component specification from consideration in most cases.

Iterative model refinement In order to build an accurate model rapidly one should iteratively refine the model. This occurs by constructing a model for each hardware component and insuring that it functions correctly when added to the model. As the modeling effort proceeds, hardware component models are refined and their accuracy validated. Once all hardware components are modeled, the overall accuracy of the model is checked. The model portions responsible for any error are identified and the model is refined to the desired level of accuracy.

Prior work showed that the above principles require explicit tool support in practice, requiring both a structural modeling language and a simulator-generating optimizing compiler to generate an *efficient* simulator [17]. Furthermore, for reuse to be practical, the system must simplify the use of flexible components by inferring parameters and avoiding overly redundant user specifications [16].

Note that the above methodology is focused on reducing modeling errors and model construction times. Elimination of specification and abstraction errors are left to the

architect. In the remainder of this paper, we will describe our experience of managing and controlling abstraction and specification errors as well as our experience of using the Liberty methodology to control modeling errors.

3. Our Target: Itanium 2

The Intel Itanium 2 processor is a member of the Itanium Processor Family (IPF) and implements the IA-64 instruction set architecture (ISA) [10]. Each IA-64 instruction has a complexity similar to that of a single RISC instruction. Three instructions are grouped into a 128-bit bundle, which also contains *stop bits* which indicate which instructions may not be issued together due to data dependencies. Instructions have access to 128 architected general purpose registers and 64 predicate registers and may be independently predicated. The ISA also supports limited compiler controlled renaming for procedure arguments, locals, and return values reminiscent of the rotating register windows in the SPARC architectureas well as rotating registers for use in conjunction with software pipelining.

The Itanium 2 has an eight-stage in-order pipeline which can issue up to two bundles per cycle (i.e., up to 6 RISC-like operations per cycle). A diagram of the pipeline appears in Figure 1. Bundles are fetched from the instruction cache in the IPG stage and placed into an instruction buffer in the ROT stage. Fetched instruction bundles are broken into issue groups based upon the stop bits and functional unit structural hazards in the EXP stage. Once an issue group is formed, the group proceeds in lock-step down the pipeline until reaching the DET stage. The REN stage implements a Register Stack Engine which manages register stack frames and inserts implicit register spill and fill instructions. The REG stage detects and stalls on data hazards. The EXE and DET stage and additional stages for floating-point and memory operations execute instructions; all exceptions are known in the DET stage. Branches are also resolved in the DET stage. The WRB stage updates registers.

The data cache unit is quite complex. The L1 data cache is tightly integrated into the main pipeline. The L2 unified cache operates independently from the main pipeline; it is non-blocking and reorders transactions to avoid bank conflicts. A unified L3 cache and system bus controller handles misses from the L2 cache. Data cache sizes are indicated in Figure 1. For comparison, we use a HP workstation zx6000 with 2 900 MHz Intel Itanium 2's running Redhat Advanced Workstation 2.1. This system has 4GB of memory with a minimum latency of 141 processor cycles.

4. Constructing the Initial Model

In this section, we describe how we applied the Liberty Modeling Methodology to construct a validated model of Intel's Itanium 2 processor. Since the Liberty Modeling Methodology is focused on modeling error, we also present extensions to its iterative refinement principle that address specification and abstraction errors.

4.1. The Modeling Process

Using iterative refinement, each pipeline stage or major processor component of the model was developed in three steps: investigating the system behavior, determining the level of abstraction to use, and building a model for the hardware. This process was repeated for each stage of the pipeline moving from the front of the pipeline to the back.

The development activities for each week were (as recorded in the modeler's journal):

Week 1: Read documentation and decided on basic overall model structure. Modeled basic IPG and ROT stages without branch prediction.

Week 2: Investigated branch behavior on short loops. Discovered that branch predictor updates insert pipeline bubbles in a complex fashion. Determined structure of pipeline logic to use branch prediction results.

Week 3: Continued investigating branch behavior and front-end bubble insertion.

Week 4: Finished investigation and modeling of branches and front-end bubbles. Investigated and modeled the EXP stage. Began investigating the REN stage, and discovered that speculation of the bottom of the register stack frame was required.

Week 5: Finished modeling the REN stage without speculation. Implemented a simple scoreboard (REG stage) w/o bypasses, EXE, DET, and WRB stages.

Week 6: Implemented REN-stage speculation. Added logic for corner cases of predicate scoreboarding. Added sampling support. Began debugging major benchmarks.

Week 7: Continued debugging. Added bypass logic. Started investigating the data-cache unit (DCU) structure.

Week 8: Continued to investigate DCU structure.

Week 9: Implemented the DCU L1 data cache, advance load address table (ALAT), and translation look-aside buffers (TLBs).

Week 10: Added very abstract level two data cache (L2), level 3 data cache (L3), memory models. Implemented level 1 instruction cache (L1I), and instruction TLBs (ITLBs).

Week 11: Cleaned up the model and added monitors to match hardware counters. Continued debugging. Added dynamic branch prediction and return address stack.

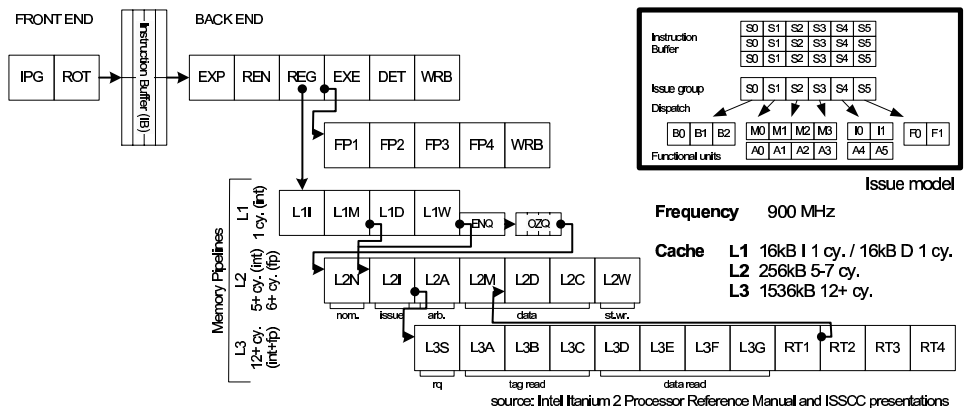

Figure 1. Itanium 2 pipeline

Notice that in each phase of the iterative refinement we strictly repeated three steps: each component was first systematically investigated, modeling decisions were made, and *then* the model portion was constructed. This discipline extends iterative refinement to act as our primary means of controlling specification error in addition to modeling error. A fourth step, evaluation of the overall model, was used throughout the refinement process when appropriate. We now describe each of these steps in more detail.

4.2. Investigation

In total, investigation took 5 of the 11 weeks. The purpose of the investigation step at each phase of refinement was to understand the behavior and structure of the processor to avoid specification errors. Two sources of information proved to be useful: documents and experiments run upon the actual hardware.

The documents used included processor manuals [9, 8], slides from symposium presentations [15, 11], white papers [1], magazines [12], and academic publications [5]. The different kinds of documents served different purposes. Slides, magazines, and white papers provided the basic pipeline structure and the parameters of structures such as caches. Processor manuals described instruction latencies, structural hazards, interesting corner cases, and performance counters. Academic publications clarified structural details of the cache designs and integer bypass paths.

While all documents were helpful, their use was not without difficulty. Documents sometimes lacked clarity or organization, obscuring vital details. For example, some elements of the second-level cache request queue are reserved. This information is provided, not in the L2 section, but rather in the L1 data cache section. Much useful information was also omitted. For example, only one document explicitly stated the write allocation policy of the L3 cache. Even worse, sometimes when a single document provided information, it was wrong. For example, the description of the EXP stage rules for memory instructions in the processor reference manual [9] is incorrect with respect to usage of load and store ports. Finally, documents were sometimes contradictory. For example, the processor reference manual [9] and the microarchitectural optimization manual [8] make contradictory statements about how bank conflicts in the L2 data cache delay accesses.

These difficulties led to the formulation of a general principle: *Quantitatively verify all documents.* Documents are good for making hypotheses about structure or behavior, but they cannot be relied upon.

Experiments were used for two purposes: to test hypotheses and to explore the behavior of the processor. Experiments were generally performed using *micro-benchmarks*, as advocated by Black and Shen [2] and Desikan et al. [3], with Perfmon [13] used to provide measurements of the Itanium 2 hardware performance counters. The typical micro-benchmark consisted of a loop with the code to be tested inside of it. The loop had a trip count high enough to overcome fluctuations in the tools used to measure hardware performance and other transients.

As an example of hypothesis testing, consider the contradiction in the documentation which was described earlier. To test which document was correct, it was necessary to set up a bank conflict between two loads which miss the first-level data cache with a use of the second load following immediately, as in Figure 2(a). The results from this microbenchmark indicated an 11 cycle latency, validating the claims made in the microarchitectural optimization manual.

As an example of behavioral exploration, consider Figure 2(b). By varying the number of issue groups of nops (no-operation instructions) inserted between the first and second issue group, we discovered that the third load could be caused to have a bank conflict with the second load's reissue after its own bank conflict. Surprisingly, the latency of the third load becomes 7, not 11 as would be expected based on the previous experiment.

Detailed investigation and modeling of behavior proved

```
{ // 1st issue group
   ld4.nt1   r20 = [r5]    // forces L1 miss
   ld4.nt1   r21 = [r5] ;; // will conflict
}
{ // 2nd issue group
   add       r2 = r21, r0  // to see latency
}
```

(a) Bank conflict micro-benchmark

```
{ // 1st issue group
   ld4.nt1 r20 = [r5] ld4.nt1 r21 = [r5] ;; }
{ //2nd issue group
   ld4.nt1 r22 = [r5] ;; // extra conflict }
//insert nop groups here
{ // 3rd issue group add r2 = r22, r0 }
```

(b) Three bank conflicts micro-benchmark

Figure 2. Micro-benchmarks

to be very beneficial for final accuracy, even when the rationale behind the behavior was initially unclear. For example, we found that the Register Stack Engine sometimes issues one spill or fill per cycle but at other times issues two, depending upon the address at which the spill or fill begins. This behavior was very easy to model, even though we did not understand why it was happening. Later, as the data cache unit was being modeled in more detail, we observed that this behavior is precisely that required to avoid bank conflicts in the second-level data cache.

4.3. Abstraction

Along the way, many decisions about the abstraction level of the model needed to be made. Some of the abstractions and approximations were:

- No instruction prefetch engine was implemented.
- The memory hierarchy beyond the L1 caches was extremely abstract: the model probed the caches, calculated a hit/miss latency, and then delayed the instruction by that many cycles.
- A constant value was charged for hardware page table walks.

Note that these abstractions were *not* validated using quantitative measurements. As will be discussed in Section 6, this mistake led to large abstraction errors. Based on our experiences with this non-quantitative strategy, we strongly discourage its use.

Of particular interest is that poorly chosen abstractions can *reduce* flexibility. The Register Stack Engine was originally modeled by causing pipeline stalls proportional to the number of registers to spill or fill, without actually performing memory accesses. This proved to require special case logic in the scoreboard, which in turn made it difficult to change pipeline organization. The time necessary to correct this poor abstraction choice (two weeks) is included in the time needed to change the model in Section 7, though the performance effects of the correction are included in all reported results.

4.4. Modeling

The model was constructed using the Liberty Simulation Environment (LSE) [17], which provides explicit support for structural modeling, aggressive reuse, and iterative refinement. Recall that the investigation took 5 of 11 weeks, meaning that the modeling activity required only 6 weeks in total. The features of the LSE designed to support the three Liberty Modeling Methodology principles were essential to this rapid model development.

The modeling of the EXP stage illustrates how LSE language features were helpful in modeling. The EXP stage takes two bundles of instructions from the Instruction Buffer (IB) as inputs and creates issue groups (i.e., groups of instructions that have no internal data dependencies). The EXP stage routes each instruction in the issue group up to the first stop bit to one of 11 *ports*. Each kind of instruction can be routed to only a subset of the ports; over-subscription is possible, in which case the EXP stage must "split" the issue group.

Figure 3 shows the structure of the EXP stage model. The entire stage is modeled by instantiating, connecting, and parameterizing modules (component templates in LSE) from the standard module library. The parameterization sets a few simple parameters on the components and uses special userpoint parameters [17] to fill in the routing computation and the bundle-to-instruction conversion algorithms. All of the work of actually manipulating signals and routing instruction information is handled by the modules, saving much time and effort.

The model of this stage has no global controller. This is a result of LSE's default flow-control semantics, which provide back-pressure from later stages automatically. LSE's default control semantics, together with a small piece of stop-bit flow-control logic and the structure of the datapath of the EXP stage model, *automatically* prevent instructions after a stop bit from leaving the IB. Because so much behavior was implicit in the modules and LSE, this stage could be modeled in only a few hours. Note however that, had the implicit behavior been incorrect, LSE would have allowed us to override the default control.

The EXP stage also illustrates another desirable outcome of structural modeling in LSE: a separation of mechanism from policy. The modules and their interconnections provide the mechanism, while the customizations (i.e., the parameter values) provide the policy. This makes it easy to modify the policy during design-space exploration; only the

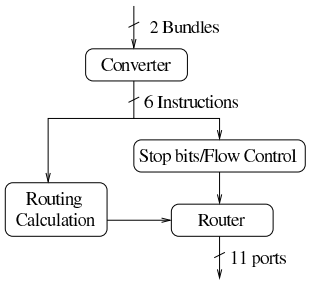

Figure 3. EXP stage model

routing computation, described earlier, need be changed.

In the EXP stage example above, all of the components came from the module library. In the complete model, 82% of the 210 component instances were instantiated from 19 modules in the standard LSE module library. The remaining instances were instantiated from hierarchical modules created through composition and parameterization of component instances. In all, 23,000 lines of composition and parameterization code were written by the user and 293,000 lines of code were generated by LSE from the modules. This indicates a high degree of reuse and is consistent with data previously presented by Vachharajani et al. for non-validated models [16].

4.5. Evaluation

Evaluation was carried out as new components of the processor were added to the model. During the initial phases of refinement, we used simple micro-benchmarks to determine whether modeling errors had been introduced. Benchmark programs were introduced during later phases; their principal purpose was to ensure that the model executed programs properly. After the full model was developed, performance accuracy was evaluated.

The initial model quality is shown in Figure 4, which reports the percentage difference in CPI between the model and the hardware. A positive difference indicates that the model was slower than the hardware, while a negative difference indicates that the model was faster. The input sets are the longest (in instruction count) "train" input (indicated in the name of the benchmark) from each of the SPEC CINT2000 benchmarks. Sampling using the TurboSMARTS framework[18] was used during simulation, with 10,000 to 20,000 samples per benchmark. The error bars indicate 99.7% confidence intervals. Only user-mode instructions are measured and modeled.

Overall error in this initial model was 7.9% with a maximum error of 20%, and it was constructed in 11 weeks. The initial model's accuracy compares favorably with with that reported in the literature [2, 3]. For example, Desikan, et al.'s validated model of the Alpha 21264 achieved an average error of 18.19% on a selection of SPEC CPU2000 benchmarks with a maximum error of 43.0%. This experience supports the Liberty Modeling Methodology claim that validated models can be constructed rapidly.

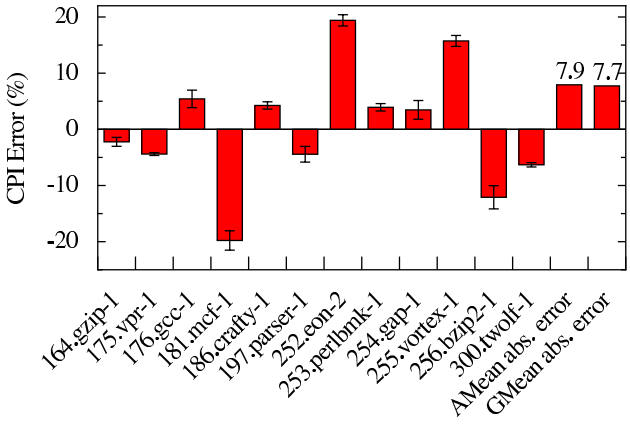

Figure 4. Initial CPI error

5. Aggregate vs. Constituent Error

It is important that models not only be accurate with respect to a baseline, but also that they accurately report the impact of changes to the hardware. In this section, we illustrate that constituent error measurements serve as a better measure of model validity than the pervasively reported aggregate error. We then analyze constituent error in our initial, aggregate-error validated Itanium 2 model.

5.1. An Illustrative Experiment

To evaluate the validity of the initial model, we explore its accuracy in reporting the impact of the addition of instruction prefetching to the hardware. Instruction prefetching was selected because Itanium 2 controls instruction prefetching through software, allowing an exact hardware measurement of the effect of enabling and disabling prefetching. Prefetching is controlled by the `.many` and `.few` completers on branch instructions. We wrote a simple tool that turned off prefetching by rewriting Electron-generated binaries with `.many` branch completers into a binary with only `.few` branch completers.

We compare the speedup achieved by instruction prefetching in the benchmark 186.crafty for the actual hardware and for two models: the initial model described in the previous system, and a model modified to highlight the effects of large offsetting errors. The modified model has higher instruction cache miss and lower data cache miss penalties chosen specifically to offset each other in this benchmark.

Table 1. CPI, speedup, and constituent errors of instruction prefetching for 186.crafty

Model	Overall CPI		Speedup	CPI error for baseline / prefetching due to:		
	Baseline	Prefetching		Front End Stalls	Load-Use Stalls	Other
Actual Itanium 2	0.636	0.623	2.1 %	—/—	—/—	—/—
Initial Model	0.649	0.597	8.7 %	6.2% / 1.2%	-5.1% / -5.8%	0.9% / -0.5%
Modified Model	0.647	0.571	13.3 %	11.7% / 3.4%	-11.0% / -12.4%	1.0% / 0.6%

Table 1 shows the overall CPI, speedup, and errors in performance constituents (relative to hardware CPI) for hardware and both simulation models. The measurement of the performance constituent errors is described more fully in the next section. The baseline overall CPI of both models (first column) is extremely accurate; the modified model is slightly more accurate.

Despite the CPI accuracy of the models for the baseline, both predict too high a speedup (second and third columns) for prefetching. To understand this, we examine the final three columns of Table 1. Both models have too high a penalty for instruction cache misses (here seen as front end stalls) in the baseline case. This excessive cache miss penalty is not seen fully in the overall CPI because it has been "masked out" by offsetting errors in load-use stalls. When the instruction cache misses are reduced through instruction prefetching, the error in front end stalls is also reduced, leaving the error in load-use stalls. This causes the CPI predictions to be too low and the predicted speedup due to instruction prefetching to be too high. Comparison of the initial and the modified model shows that the larger the offsetting errors, the larger the error in the predicted speedup, even though in the baseline case, the model with larger offsetting errors actually has a lower error in overall CPI.

The difference between the speedup reported by the modified model and the speedup of the actual hardware may adversely affect cost/benefit decisions made on proposed hardware or erroneously overstate the impact of published research results. This realization encouraged us to investigate the constituent errors of our initial model in detail and then refine our model to reduce these errors.

5.2. Initial Model Constituent Error Analysis

The Itanium 2 hardware includes performance counters which are able to classify clock cycles as either cycles with completing instructions or bubbles. Bubbles can be further classified by where in the pipeline they originated. These classifications can be used as performance constituents: the total number of cycles is the total number of bubbles of all kinds plus the number of cycles with a completing instruction. By implementing analogous performance counters in the simulation model it is possible to compare the hardware with the simulation model at the level of performance constituents.

The performance constituents are:

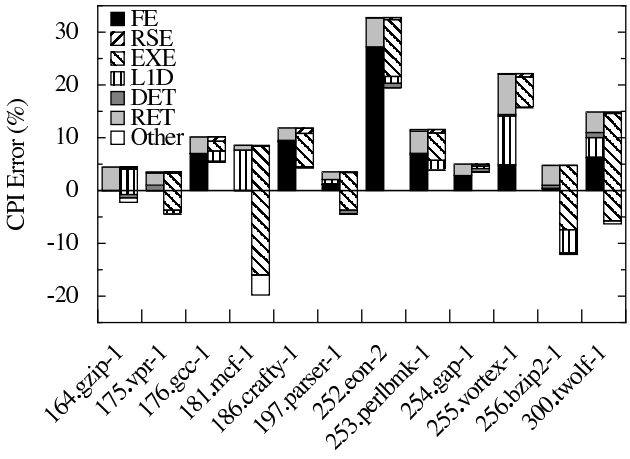

Figure 5. Initial model error constituents

FE - Front-end bubbles – mostly due to instruction cache and instruction TLB misses.

RSE - Register Stack Engine bubbles – cycles in which register spills and fills are being performed.

EXE - Data hazard bubbles – nearly always due to load-use dependencies in these benchmarks.

L1D - L1 data cache pipeline bubbles – from a variety of sources, including hardware page table walks, load-store conflicts, and L2 back-pressure.

DET - Pipeline flush bubbles.

RET - Instruction retirement cycles.

Other - Unaccounted-for differences between the models – due to sampling error and variations in hardware performance between runs.

Figure 5 presents the errors in performance constituents relative to overall CPI for all SPEC CINT2000 benchmarks. In this figure, the size of a bar segment indicates the magnitude of the difference of a constituent of CPI between the two models, while its position (left or right) indicates the sign of the difference. For each benchmark, the left-hand bar contains constituents of CPI where the model is slower than the hardware. The right-hand bar contains constituents of CPI where the model is faster than the hardware. The two bars are aligned at the top so that the distance between the bottom of the rightmost bar and the zero axis indicates the total error in the model. For example, for 181.mcf, the

error in the L1D constituent is approximately +7%, the error in the EXE constituent is about -24%, and the total error is -20%.

The first observation to make from these results is that the performance difference of individual constituents varies widely by benchmark. This indicates that despite similar CPIs among many of the benchmarks, they have quite different behavior (though the converse is not true, similar accuracy does not imply similar behavior.) Some constituents (e.g. L1D) are even positive in some benchmarks and negative in others. This probably indicates that there are offsetting errors within the L1D constituent.

The second observation to make is that there are both positive and negative constituents of error for each benchmark; errors are offsetting, giving better overall results than the individual constituents, indicating that the model quality is not as high as initially believed, despite validation. The data shows that a model validated to a single aggregate metric, in this case CPI, can seem accurate while having substantial error in certain pipeline details.

6. Refining the Itanium 2 Model

Based on the error analysis in Section 5 we iteratively refined the initial model to reduce FE, L1D, and EXE constituent errors. As in the initial model development, the steps consisted of investigating behavior, deciding upon a (new) level of abstraction, updating the model, and evaluating the results.

6.1. Refinements

The largest FE error occurred in 252.eon. Inspection of sub-event counters showed that the L1 instruction cache miss rate was much higher in the model. Inspection of the 252.eon binary showed that it made heavy use of the streaming prefetching hints provided in the ISA. One of the abstractions in the initial model was to ignore the prefetch engine. Thus this error was an abstraction error. The solution to this abstraction error was to implement a simplified (i.e., still somewhat abstracted) prefetch engine. This required less than a day to create a simple state machine to generate prefetch requests, connect the requests to the cache hierarchy, change the L1 instruction cache LRU algorithm to bias against prefetched data, and add some performance and debug monitors.

L1D errors were significant in 255.vortex and 181.mcf. Looking at the sub-events of L1D, we found that the TLB miss rate was too high and the average cost of a miss was too high. The miss rate was a specification error; we had assumed a 4K page size instead of the proper 16K page size. The average cost difference was due to an abstraction error; we modeled TLB misses with a fixed cost. We corrected the page size (a simple parameter change) and replaced the fixed cost TLB-miss model with a less abstract one that performs accesses to the page table. The TLB analysis and fix took less than a day and required changes only to a small portion of the memory hierarchy.

EXE errors were large in several benchmarks, and particularly large in 181.mcf and 300.twolf. Furthermore, after the TLB fix, these same benchmarks had large negative L1D errors, which had been masked previously by offsetting positive TLB errors. Examination of the EXE sub-events showed that it was nearly all due to load-use stalls, while examination of the L1D sub-events showed errors in L2 back-pressure and L2 tag re-reads. L1 data cache miss rates were generally correct. Taken together, this evidence indicated that the L2, L3, and memory models were causing the errors. These errors are abstraction errors.

Reducing these error constituents required an understanding of the memory hierarchy beyond the L1 caches. As before, documents and experiments were used to develop this understanding. Eight days were spent in the investigation. This investigation took so long because the L2 cache subsystem is non-blocking and is able to process requests out of order to improve performance. Bank conflicts, data bypasses, non-LRU cache replacement policies, re-reads of the L2 tag array, and secondary miss processing all combine to make it very difficult to determine what the latency of a particular cache access should be. Furthermore, while the cache controller is richly endowed with performance counters, the documentation of these counters is very limited. Nevertheless, it was possible to understand much of the L2 cache behavior and some of the L3 behavior.

After the eight days of investigation, L2 cache behavior was modeled with a high degree of detail. The total amount of time used in creating the new L2 cache controller was four days. The only changes needed in components other than the L2 cache controller were some minor changes in the L1 data cache to L2 cache interface. The L3 cache and bus interface were modeled in less detail than the L2 cache, but in more detail than in the initial model. Creating the new L3 cache and bus interface model took one day and required changes to no other components of the model.

Note that nearly all the constituent errors in the model were due to specification and abstraction errors, lending support to the claim that structural modeling helps prevent modeling errors. Furthermore, all of the refinements described in this section were achieved in an iterative fashion over the course of 16 days. About half of that time was spent investigating the behavior of the hardware. The reason that modification time was small is due to the structural modeling approach. We found that the natural hardware-based partitioning of the model provided internal interfaces at the locations where changes needed to be made. Thus changes often consisted of simply creating a new portion of

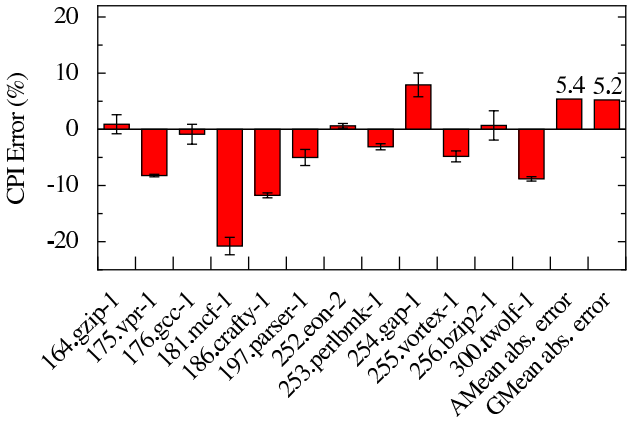

Figure 6. Current model CPI error

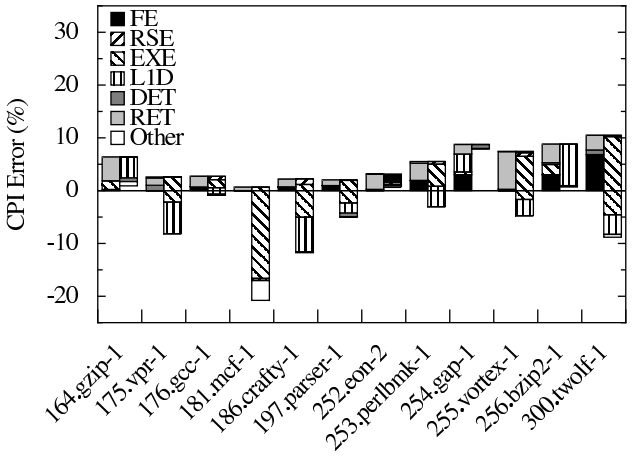

Figure 7. Current model error constituents

the model and hooking it up to the rest of the design in place of the logic it replaced, with no need to even look at other portions of the model. In a few cases where information from other parts of the model was needed, it was accessible and merely needed to be routed to the new portion.

6.2. Current model

The results of the refinement are given in Figure 6. The breakdown of performance constituents is given in Figure 7. These figures show an overall reduction in error, down to 5.4% on average, but more importantly, the targeted constituent errors have decreased significantly. Figure 8 shows the average absolute constituent error across all benchmarks for both the initial and the final model. The FE and EXE error constituents have been significantly reduced. L1D errors have increased slightly because, as discussed before, there were offsetting errors within this constituent. In the 16 days of refinement, overall constituent error decreased by 34%.

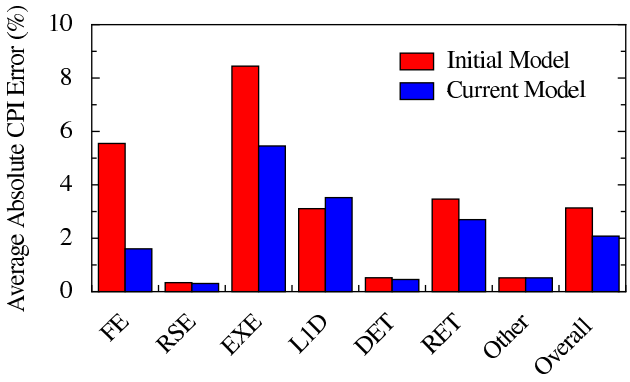

Figure 8. Constituent error change

7. Evaluating flexibility

While the refinement process provides evidence that the Itanium 2 model is flexible, we further show that the model can be used to explore a wider design space. We recount here two independent user experiences.

The first user of the model, the model developer, used the initial model as a starting point for studying modifications to the Itanium 2 processor pipeline that involved adding a limited degree of out-of-order behavior to the pipeline. Instructions were allowed to issue while ignoring load-use dependencies. These instructions would recirculate through the pipeline until the dependencies were resolved. Creating the model for this modified pipeline required modifications only to the portions of the model most closely involved with the changes in the pipeline: the scoreboard logic, the register files, the data cache unit, and the branch resolution logic. This overhaul of the pipeline organization with debugging of the new technique was made in only 6 weeks (including two weeks used to reduce the amount of abstraction used to model the Register Stack Engine), indicating that the initial model could indeed be modified rapidly.

The second user, unfamiliar with the model but familiar with the LSE tools, used the constituent-validated Itanium 2 processor model to explore a novel chip-multiprocessor interconnection mechanism called the synchronization array [14]. This required that he instantiate two Itanium 2 cores and a synchronization array, connect the cores to a shared L2/L3 memory hierarchy, and augment the Itanium 2 datapath to handle the sychronization array instructions. This was accomplished in two weeks; this time includes both the development/debugging time for the model configuration and time spent debugging errors in modified benchmark binaries.

8. Conclusions

Given the central role of simulation in modern processor design and research, a validated baseline model upon which

credible studies can be based is extremely desirable. Unfortunately, validated models are not used because it is widely believed that these validated models are either too time-consuming to develop [4, 2], too difficult to develop due to lack of published information [7], or too time-consuming to modify for wide ranging design-space exploration [4]. Some have claimed that the Liberty Modeling Methodology [17, 16] addresses these issues.

In this work, we present our experience building a validated model of Intel's Itanium 2 processor using the Liberty methodology and the Liberty Simulation Environment (LSE). Though not an ironclad proof or exhaustive study, this experience shows that the Liberty Modeling Methodology and supporting tools can be extremely effective. An initial model was constructed by a single modeler in only 11 weeks. This model predicted hardware cycles-per-instruction (CPI) to within 7.9%. This supports the first set of prior work claims: validated models can be constructed rapidly.

We learned three lessons the hard way. First, we learned that documentation is sometimes in error and often contradictory and vague; any information gathered from it should be validated with quantitative experiments. Second, all approximations should be supported with quantitative experiments that support their validity. Third, and most importantly, we show in this paper that building a model that is validated according to a single aggregate metric does *not* necessarily provide an adequate model for exploring design alternatives. We show that, despite having a high degree of accuracy in an aggregate metric such as CPI, the model can contain significant constituent errors that happen to offset each other.

We were able to apply the Liberty Modeling Methodology to refine our initial Itanium 2 model to reduce the constituent errors with only $2\frac{1}{2}$ additional weeks of effort. This new model predicts overall CPI to within 5.4% with substantially reduced error for the targeted constituents. Furthermore, these Itanium 2 models were modified to explore a novel multiprocessor communication mechanism and novel pipeline organizations for EPIC machines. These significant additional modifications were made in under 8 person-weeks in total. The speed with which these modifications were made supports the second set of prior work claims: validated models built using the Liberty Modeling Methodology can be rapidly modified for design-space exploration.

References

[1] Inside the Intel Itanium 2 processor. Hewlett Packard Technical White Paper, July 2002.

[2] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[3] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 266–277, July 2001.

[4] R. Desikan, D. Burger, S. W. Keckler, L. Cruz, F. Latorre, A. Gonzalez, and M. Valero. Errata on "Measuring Experimental Error in Microprocessor Simulation". *ACM SIGARCH Computer Architecture News*, 30(1):2–4, March 2002.

[5] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad. A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor. In *IEEE Journal of Solid-State Circuits*, volume 37, pages 1433–1440, November 2002.

[6] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 49–58, November 2000.

[7] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, pages 40–49, February 2002.

[8] Intel Corporation. *Introduction to Microarchitectural Optimzation for Itanium 2 Processors: Reference Manual*. Santa Clara, CA, 2002.

[9] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, April 2003.

[10] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference, Revision 2.1*. Santa Clara, CA, 2004.

[11] T. Lyon. Itanium 2 processor microarchitecture overview. Vail Computer Elements Workshop, June 2002.

[12] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE MICRO*, 23(2):44–55, March-April 2003.

[13] Perfmon: An IA-64 performance analysis tool. http://www.hpl.hp.com/research/linux/perfmon.

[14] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[15] D. Soltis and M. Gibson. Itanium 2 processor microarchitecture overview. Hot Chips 14, August 2002.

[16] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A high-level modeling language for component reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 195–206, June 2004.

[17] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[18] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report 2004-003, Computer Architecture Lab at Carnegie Mellon, November 2004.