Rapid Development of a Flexible Validated Processor Model

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

Manish Vachharajani
Department of Electrical Engineering
University of Colorado
Boulder, CO 80309
manishv@colorado.edu

Liberty Research Group Technical Report 04-03, November 2004

For a variety of reasons, most architectural evaluations use simulation models. An accurate baseline model validated against existing hardware provides confidence in the results of these evaluations. Meanwhile, a meaningful exploration of the design space requires a wide range of quickly-obtainable variations of the baseline. Unfortunately, these two goals are generally considered to be at odds; the set of validated models is considered exclusive of the set of easily malleable models. Vachharajani et al. challenge this belief and propose a modeling methodology they claim allows rapid construction of flexible validated models. Unfortunately, they only present anecdotal and secondary evidence to support their claims.

In this paper, we present our experience using this methodology to construct a validated flexible model of Intel's Itanium 2 processor. Our practical experience lends support to the above 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. Our experience with this model showed us 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. Using this as our motivation, we explore the flexibility of the model by modifying it to target specific error constituents, such as front-end stall errors. In $2\frac{1}{2}$ person-weeks, average CPI error was reduced to 5.4%. The targeted error constituents were reduced more dramatically; front-end stall errors were reduced from 5.6% to 1.6%. The swift implementation of significant new architectural features on this model further demonstrated its flexibility.

1 Introduction

Despite great advances in the field of analytical modeling [1, 2, 3, 4] and due to barriers in prototype development, simulation is the preferred method of measurement in 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 [5], Gibson et al. [6], and Desikan, et al. [7] have shown that this effect can be significant and that non-validated models would lead to incorrect conclusions in certain

cases. Non-validated simulators may also contain performance bugs which may go undetected for lack of a known-correct reference. These bugs may affect the results in undesirable ways [7, 8].

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 [5]. A second, related reason is that even if simulators were easy to construct, not enough information about a design is publicly available to decide what to construct [9]. Finally, validated models are often so detailed that they may be too time-consuming to modify for initial studies of many different design points [10]. This may impede innovation if it becomes difficult to study design points not immediately adjacent to the reference machine.

Desikan et al. suggest that initial studies be performed via abstract non-validated models whose trends have been checked against a validated model [10]. Once a final design point is chosen, it should be simulated in both the abstract model and a modified version of the validated model. The results of these models are then compared to ensure the study's accuracy. Unfortunately, no suggestion is made for ensuring the fidelity of the abstract model with respect to the validated model. Furthermore, no suggestion is made for quickly fixing the abstract model and re-doing the early design-space exploration in the event of a mismatch between the final abstract model and the modified validated model.

Vachharajani et al. claim that validated models need not be hard to construct and need not be difficult to modify to explore wide areas of the design space, and they present a modeling methodology (which we will call the Liberty modeling methodology) to support this claim [8, 11, 12]. 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, while they support their claims with anecdotal and secondary evidence, their primary claims have gone unsubstantiated with respect to the difficult realities of validated models.

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*, unbounded internal error constituents may simply offset each other. We show that such errors are present in our initial model and that these "offsetting errors" can lead to incorrect 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 internal canceling errors.

To assess validated model malleability claims, 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 [13].

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 all sources of significant error in a system. Black and Shen identify three such sources of error in performance models [5]. 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. Typically, one believes or hopes that such behavior is irrelevant to the experiments being performed.

To this list we add a fourth source of error: sampling error introduced by techniques such as SMARTS or SimPoint. [14, 15]. The use of these powerful techniques for sampling, and consequently the sampling error, is unavoidable given the speed of cycle-accurate simulators.

Vachharajani et al. describe a modeling methodology, which we will call the Liberty modeling methodology, that they claim allows rapid construction of validated models [8, 11, 12]. To do so, the methodology must provide a means to control the above sources of error while also enabling rapid model construction. Consequently, the Liberty modeling methodology is centered around three modeling principles. They claim that each of these modeling principles 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 [8]. 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 [8].
- **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** Vachharajani et al. suggest that 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. Notice that this iterative refinement procedure requires that a working model be available at each stage, even for incomplete specifications.

Vachharajani et al. show that the above principles require explicit tool support in practice. By definition, structural modeling requires a structural modeling language and a simulator-generating optimizing compiler to generate an *efficient* simulator [8, 16]. Furthermore, for reuse to be practical, the system must simplify the use of flexible components by inferring parameters and avoiding overly redundant user specifications [11, 17]. Finally, iterative refinement requires system support to emulate behaviors not yet modeled, so that a working simulator may be produced at each stage of modeling.

Beyond support for the above principles, they also advocate the use of independent specification of the model and experimental infrastructure. The performance measurement, sampling, and check-pointing code necessary to improve the usability of the model and facilitate model refinement should not interfere with the model itself. In the data collection example, this allows a clearer description of both the model's behavior and the data being collected, reducing the likelihood of error in either portion. It also allows a variety of data collection codes to be used interchangeably with each model. Specifying experimental infrastructure in this way can be considered a form of aspect-oriented programming and it is important for the tools to support this feature [11].

Reconsidering Black and Shen's sources of error, notice that the above methodology is focused on reducing modeling errors and model construction times. Elimination of specification and abstraction errors are, for the most part, left to the architect. Some suggest quantitatively validating approximations but do not suggest a strategy to do so rapidly [12]. Throughout the remainder of the paper, we describe our experience trying to manage and control abstraction and specification errors. We also describe our experience using the Liberty methodology to control modeling errors.

3 Our Target: Itanium 2

To fully understand the case study presented in this paper, we first define our target model, the Intel Itanium 2. The description in this section is far from complete but provides sufficient background to understand the development of our Itanium 2 model.

The Intel Itanium 2 processor is a member of the Itanium Processor Family (IPF) and implements the IA-64 instruction set architecture (ISA) [18], an EPIC [19] instruction set. Each IA-64 instruction has a complexity similar to that of a single RISC instruction. Up to 3 instructions are grouped into a bundle with stop bits (described later) in 128 bits. 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 architecture [20], as well as rotating registers for use in conjunction with software pipelining.

Since IA-64 is an EPIC ISA, data dependences are explicitly marked. In IA-64 this is done via *stop bits* that appear in a bundle. The encoding supports placing a stop bit between any pair of instructions (with certain limitations that are beyond the scope of this paper). The compiler must ensure that any operations appearing between a pair of stop bits are data independent

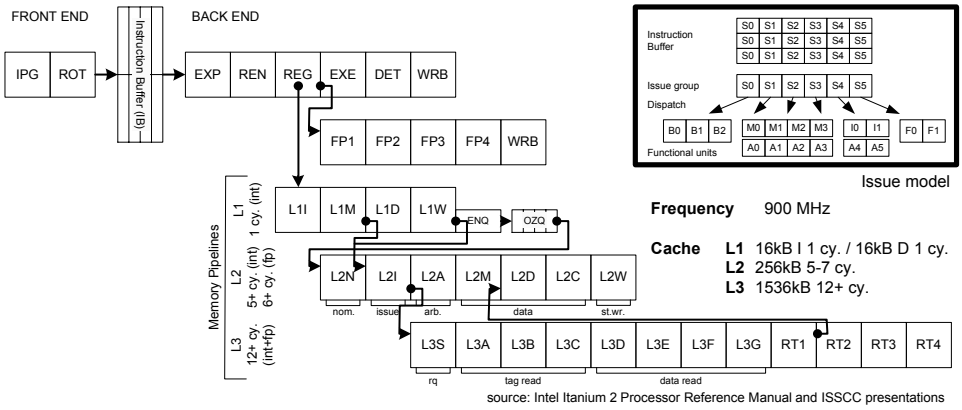


Figure 1: Itanium 2 Pipeline

Unit	Characteristics
Functional Units	6-issue, 6 ALU, 4 Memory, 4 FP, 3 Branch
Data Model	64-bit
L1 Instruction Cache	1 cycle, 16 KB, 4-way, 64B lines
L1 Data Cache	1 cycle, 16 KB, 4-way, 64B lines, writethrough
L2 Cache	5,7,9+ cycles, 256KB, 8-way, 128B lines, writeback
L3 Cache	12+ cycles, 1.5 MB, 6-way, 128B lines, writeback
Main memory	4GB, 141 cycles (HP workstation zx6000)

Table 1: Itanium 2 Characteristics

(with exceptions for certain predicate defining and branch operations). Note that there is no implicit stop bit at the end of a bundle; this allows for groups containing an arbitrary number of independent operations.

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 units are 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. These transactions arbitrate for access to the L2 data array or the L3 unified cache and system bus (which are controlled as a single unit). One unusual aspect of the L2 behavior is that the tags are read by the L1 data cache pipeline and thus hit or miss indications are available early. This early tag read requires that in some cases, particularly secondary misses, the L1 data cache and main pipelines occasionally be stalled to allow L2 tags to be re-read.

For this paper, an HP workstation zx6000 with 2 900 MHz Intel Itanium 2's running Redhat Advanced Workstation 2.1 was used for comparison. A summary of the particular processor implementation in this system is given in Table 1.

4 Constructing the Initial Model

In this section, we describe how we applied the Liberty modeling methodology in constructing a validated model of Intel's Itanium 2 processor. Since the Liberty modeling methodology is focused on modeling error, we also present our extensions to their iterative refinement process 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 (e.g., IPG and ROT) to the back end (e.g., EXP and DET).

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

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

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.

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

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.

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

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

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

8: Continued to investigate DCU structure.

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

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).

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

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 [21, 22], slides from symposium presentations [23, 24], white papers [25], magazines [26], and academic publications [27]. 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 behavioral 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, the size of the second-level cache request queue is 32 requests, as stated in the L2 cache section of the processor reference manual [21]. However, because it takes some time between the time that the first-level data cache issues the request and the L2 data cache actually enqueues it, 12 entries are reserved for in-flight requests. 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, many documents stated the cache sizes and most gave the write allocation policy of the L2 cache, but only one explicitly stated the write allocation policy of the L3 cache. 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 [21] is incorrect with respect to usage of load and store ports.

Documents were also contradictory. For example, the processor reference manual [21] and the microarchitectural optimization manual [22] make contradictory statements about how bank conflicts in the L2 data cache are resolved. The former states that when loads in the same issue group both miss in the L1 data cache and access the same bank of the L2 cache, users of the data produced by the second load see a 7 cycle latency for the data. The latter states an 11 cycle latency.

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 [5] and Desikan et al. [7], with Perfmon [28] 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 a 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 latency was then computed by adding 1 to the number of EXE-stage bubbles reported by Perfmon. 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-

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

(a) Bank conflict micro-benchmark

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

(b) Three bank conflicts micro-benchmark

Figure 2: Micro-benchmarks

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 re-issue 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 of behavior proved to be very beneficial for final performance, 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 is 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 Register Stack Engine (RSE) was modeled by simply inserting stalls proportional to the number of registers to spill or fill, without actually performing memory accesses.
- The memory hierarchy beyond the L1 caches was extremely abstract; a latency would be calculated by probing the caches for hit/miss status; the transmission of the instruction result to the core would then be delayed the appropriate amount.
- 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 some poorly chosen abstractions made it *more* difficult to modify the model later. The decision to abstract the Register Stack Engine (RSE) behavior caused difficulties since it required the scoreboard to have special cases to handle the register spills and fills in the RSE. Dealing with these special cases when changing pipeline organization was extremely difficult. To remedy these problems we modified this initial model to remove the RSE abstraction during our research.

4.4 Modeling

The model was constructed using the Liberty Simulation Environment (LSE) [8], which provides explicit support for structural modeling, aggressive reuse, and iterative refinement. Recall that the investigation took 5 of 11 weeks, meaning that the

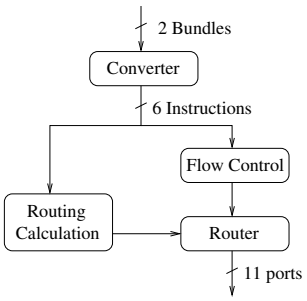


Figure 3: EXP stage model

modeling activity required only 6 weeks in total. The features of the LSE designed to support the three Liberty modeling principles were essential to this rapid model development.

For iterative refinement, LSE's functional emulator abstraction was helpful in enabling partial model evaluation. As each major hardware component model was completed, the model was able to ran full programs. LSE's emulator abstraction was able to automatically handle the architectural functionality of the missing microarchitectural model components. Thus, at any point in the refinement it was possible to run the model on actual programs for evaluation of the existing structures.

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[1] (i.e., groups of instructions that have no internal data dependencies). The EXP stage routes each instruction in the issue group 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. Notice how the entire stage is modeled by instantiating, connecting, and parameterizing modules (component templates in LSE) from the standard module library. The parameterization only sets a few simple parameters on the components and uses special userpoint parameters [8] 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 concurrent, structural nature of the specification made the actual interconnection easy to see once the hardware was understood.

Notice further that the model of this stage had 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 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 routing computation, described earlier, need be changed. Additionally, LSE automatically parameterizes the entire stage for width; if more input

[1] Recall from Section 3, the compiler provides stop bits which indicate when a group of independent instructions has ended. A stage for forming issue groups from bundles is needed because these stop bits may be in the middle of a bundle.

bundles or output ports are connected to the stage, it adapts to route or use them. This made modifying the model even easier, and thus faster.

LSE's data collectors proved useful. The initial model used these data collectors to incorporate debugging and experimental infrastructure including monitors, sampling, checkpointing, and performance counters. The collection mechanism allowed a clean separation of the infrastructure from the model. As a result, it was easy to remove monitors when they were not necessary without requiring changes to the model. For example, two versions of the simulator could be built – one with, and one without the debug monitors – without a change to the model itself, but just a line in the testbench specification including or excluding the monitors. Particular effort was made to provide, for any implemented behavior, all the performance counters available in the hardware. These were essential for deciphering differences in performance between the hardware and the model.

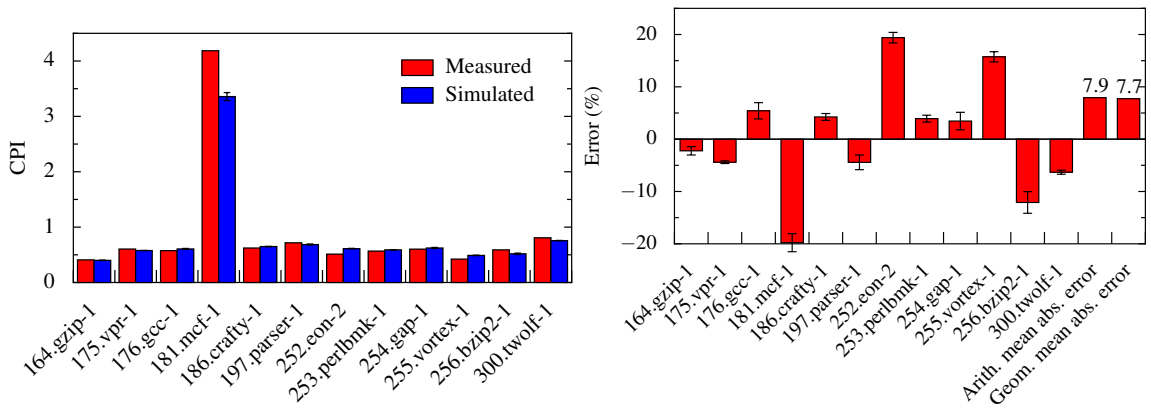
In the EXP stage example above, all of the components came from the module library. In fact, 82% of the model's 210 component instances were instantiated from 19 modules in the standard LSE module library. The remaining instances were instantiated from hierarchical modules created through composition 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. In all, this indicates a high degree of reuse and is consistent with data previously presented by Vachharajani et al. for non-validated models [11].

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 (after the RSE abstraction was removed) is shown in Figure 4. The left panel shows cycles per instruction (CPI) while the right panel shows 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[29] 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 [5, 7]. 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 seems to support the Liberty modeling methodology claim that relates to the rapid construction of validated models; a further examination of accuracy is performed in the next section. Also at this point in our experience, model flexibility and utility claims remain unaddressed.



(a) Overall CPI

(b) CPI Error

Figure 4: Initial model results

5 Aggregate vs. Constituent Error

Computer architects develop models to explore the effects of proposed changes on real machines. Therefore, to truly measure the utility of any model, we need to first explore how accurately it reports the impact of a change to the hardware. In this section, we illustrate that constituent error measurements serve as a more accurate measure of model validity than the pervasively reported aggregate error. Then, with this insight, we find a weakness in our initial, aggregate-error validated Itanium 2 model. This motivates further refinement described in Section 6.

5.1 An Illustrative Experiment

To evaluate the effectiveness of the initial model, we explore its accuracy in reporting the impact of a change to the hardware. Specifically, we evaluate the effect of instruction prefetching on Itanium 2. Instruction prefetching was selected since Itanium 2 allows for instruction prefetching to be activated or deactivated by an application itself. This allows for an exact hardware measurement of the effect of enabling and disabling prefetching. Specifically, 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. For the purposes of this illustration, we examine 186.crafty in more detail.

The first column of Table 5 indicates the baseline CPI (no instruction prefetching) for the Itanium 2 hardware and two models validated for aggregate CPI, our initial model and an alternate CPI-validated model. The alternate model was created in attempt to further refine the overall CPI of the initial model without regard for constiuent error. As can be seen in the table, the baseline CPI in the alternate model is actually closer to the Itanium 2 hardware performance than the initial model (1.7% error versus 2.0% error).

Despite the accuracy of the baseline, Figure 6 shows that this is a consequence of larger offsetting constituent errors. 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 indicates the sign. 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, and thus the distance between the bottom of the rightmost bar and the zero axis indicates the

Model	Overall CPI		Speedup
	Baseline	Prefetching	
Alternate Model	0.647	0.571	13.3 %
Initial Model	0.649	0.597	8.7 %
Actual Itanium 2	0.636	0.623	2.1 %

Figure 5: CPI and Speedup of instruction prefetching on a CPI validated model, the initial constituent validated model, and an actual Itanium 2 machine running 186.crafty.

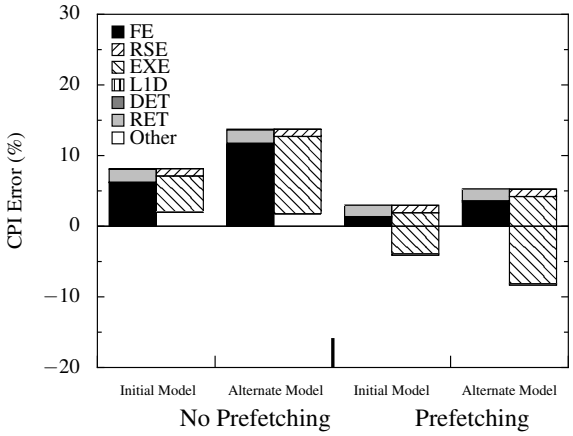


Figure 6: Constituent errors relative to actual Itanium 2 machine running 186.crafty.

total performance difference between the model and hardware. For the baseline (i.e., no-prefetching) on the initial model, 186.crafty is approximately 8% too fast due to differences in the FE category and 6% too slow due to differences in the EXE category. The FE category captures all stalls in the front end of the pipeline. The EXE stall category is composed of data hazard stalls, primarily load-use dependences. Since FE stalls are related to instruction fetch, we would expect an interaction with instruction prefetching. This is not the case for EXE stalls[2].

When prefetching is added to the above models and activated on the hardware, we see performance improvement in all cases. The real hardware indicates a 2.1% speedup while the initial model demonstrates a larger 8.7% speedup. The alternate model shows a much larger 13.3% speedup. Figure 6 reveals the source of this overestimation in both models to be the original overestimation of FE stalls. The alternate model fares worse because the unvalidated FE constituent error is larger. Notice how the model with the lower aggregate CPI error reports the speedup *less* accurately. By induction, we can see that models with no aggregate CPI error could potentially report any result as a consequence of offsetting constituent errors.

The resulting order of magnitude difference between the speedup reported by the alternate 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. The error analysis is presented in below. Section 6 describes the refinement process.

5.2 Initial Model Constituent Error Analysis

Compared to other reported model errors, the initial model is extremely accurate, but, as we have seen, its effectiveness as a tool is related to the constituent error it exhibits. The constituent error is presented in Figure 7 for all SPEC CINT2000 benchmarks. The categories used in this figure are the same ones used before and correspond to the different kinds of pipeline

[2]The specifics of the remaining categories will be described later.

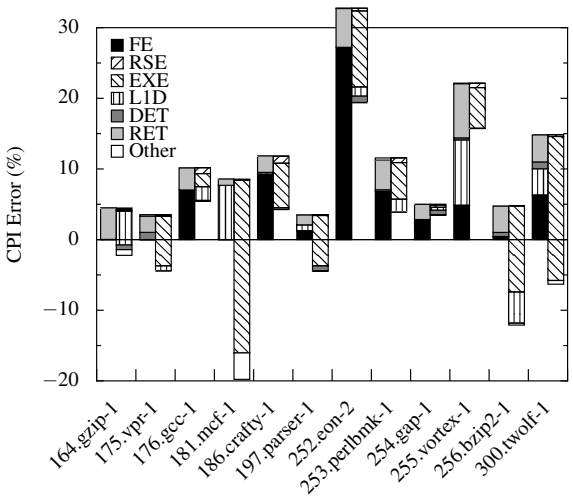


Figure 7: Initial model error constituents

bubbles reported by the Itanium 2 performance counters[3]:

- FE - front-end bubbles. These bubbles are due mostly to instruction cache and instruction TLB misses. They can also be caused by interlocks when branch prediction state is updated or repaired.
- RSE - Register Stack Engine bubbles. These represent cycles in which register spills and fills are being performed.
- EXE - Data hazard bubbles. These are nearly always due to load-use dependencies in these benchmarks.
- L1D - L1 data cache pipeline bubbles. These come from a variety of sources. The most straight-forward are hardware page table walks and load-store conflicts. More subtle causes arise because the L2 cache shares the integer return datapath with the L1 data cache and because the L2 cache tags are part of the L1 pipeline. Thus returning data and situations where the L2 needs to re-read or modify its tags can cause pipeline bubbles.
- DET - Pipeline flush bubbles.
- RET - Retirement cycles. These are cycles in which instructions retire, which are affected by the issue group formation logic and situations where only a single bundle is available in the instruction buffer.
- Other - Unaccounted-for differences between the models. These stem from sampling error and variations in hardware performance between runs.

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 more than one error in model behavior affects that constituent.

The second observation to make, given this breakdown, 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

[3]Note that a finer breakdown of some of the bubbles is possible with the Itanium 2 performance counters, though complicated somewhat by the fact that the sub-events of many of these counters can happen in the same cycle and both increment, though the total bubble count only increments once. This breakdown is not provided in the interest of clarity.

metric, in this case CPI, can seem accurate while having substantial error in certain pipeline details. The prevalence of offsetting errors in all cases suggests that this is a likely scenario in most models. Since the users of our model rely on it for accurate results during architectural exploration, we needed a better model. In the next section, we describe experience refining the model further given our understanding of constituent errors.

6 Refining the Itanium 2 Model

Based on the error analysis in Section 4 we refined the initial model by concentrating on FE, L1D, and EXE constituent errors. After describing the refinements, we will discuss the current state of the model, including its prediction accuracy.

6.1 Refinements

Iterative refinement was used to target three particular constituents of error: FE bubbles, L1D bubbles, and EXE bubbles. 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.

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 only several hours of work: the creation of a simple state machine which begins generating prefetch requests when a taken branch fetch redirection with a `.many` completer is seen, connection of the request to the cache hierarchy, a change to the L1 instruction cache LRU algorithm to bias against prefetched data, and addition of 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 model spent too many cycles for hardware page table walks in most benchmarks, 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 DCU recirculation (this will be explained later). 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.

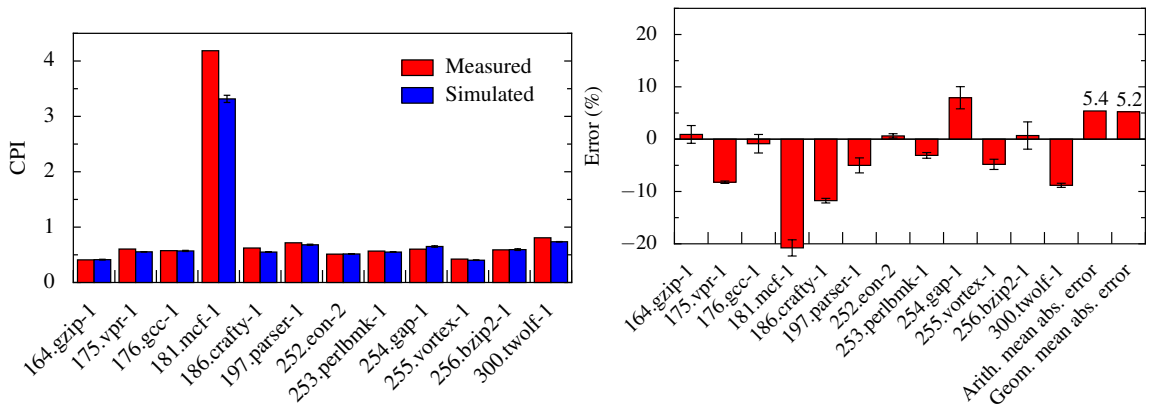
The L2 cache subsystem is non-blocking and is able to process requests out of order to improve performance. Three aspects of its behavior are unusual. First, the L2 cache will sometimes start processing a request and then determine it should not have done so. This is called a "cancellation"; bank conflicts are the major source of cancellations. Second, when line replacements and secondary misses occur, L2 cache requests need to be "recirculated" back into the L1 data cache pipeline because the L2 tag array is in that pipeline. Finally, minimum latency accesses require use of a "bypass" path; earlier recirculations and cancellations can prevent this. All three of these behaviors increase the access latency and the number of EXE bubbles. Recirculations and integer data returning to the pipeline cause L1D bubbles when L1 data cache accesses happen to occur at the same time. Also, when there are many L2 cache requests (such as is the case with 181.mcf), increases in effective L2 latency caused by recirculations, cancels, and loss of the bypasses result in increased L2 request queue occupancy and more back-pressure to the L1 cache, resulting in more L1D bubbles.

L2 cache behavior was modeled with a high degree of detail, using a state machine and timers to track the movement of accesses through the bypasses, data arrays, and cancellation states. Recirculation is present, but modeled in less detail because investigations have not yet uncovered much about this behavior. 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.

Two other errors were discovered while modeling the more detailed L2 cache. The first one was a specification error in the EXP routing policy caused by an error in the manuals. Fixing it involved a single character change in the routing calculation. The second was a modeling error wherein a status field was set incorrectly, causing a single instruction type to be sent to the wrong portion of the data cache unit. A small modification to the data cache unit fixed this error.

The L3 cache and bus interface were modeled in less detail than the L2 cache, but in more detail than in the initial model. The model consists of a state machine and queues to model the L3 tag arrays, the L3 data arrays, and the memory. These queues model the latency required to access the structures. Because the data arrays and memory are not perfectly pipelined, the queues also model inter-operation latency (e.g., one data array access can begin every four cycles). Creating the new L3 cache and bus interface model took one day and required changes to no other components of the model.

All of the refinements described in this section required more time to investigate the behavior of the hardware than they did to actually model that behavior. The reason that modification time was small related to the structural modeling approach. These modifications were not *foreseen* modifications in the sense that we had not planned specifically to make them possible; very little thought was taken for making modifications possible at all. Yet they were easily made. One could imagine a non-structural processor model that had been cleverly designed to allow a certain set of modifications through careful function interface design. Yet any modifications outside of those pre-conceived ones could be difficult because state or functionality needed is not accessible or partitioned properly [8]. Such cleverness in model design is not necessary when using structural modeling as no mapping from hardware components to function or objects is necessary. We found that the natural hardware-based partitioning of the model provides internal interfaces at the locations where changes need to be made. Thus changes often consisted of simply creating a new portion of 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 cases where additional information was needed, it



(a) Overall CPI

(b) CPI Error

Figure 8: Current model results

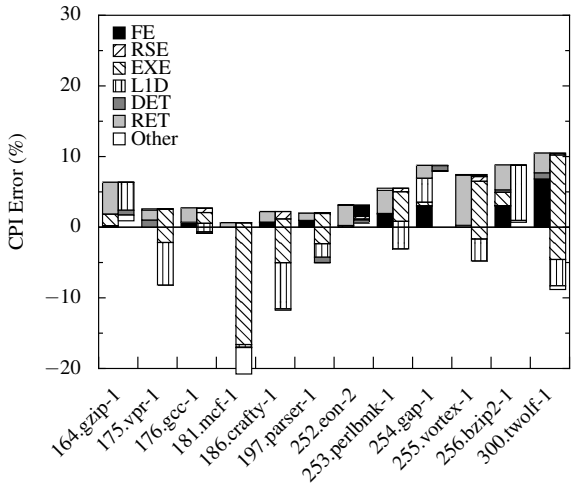


Figure 9: Current model error constituents

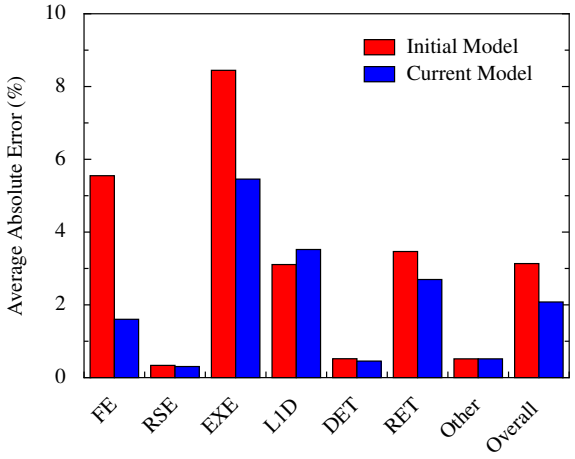


Figure 10: Constituent error change

was also accessible and merely needed to be routed to the new section.

All of the refinements in this section were achieved in an iterative fashion over the course of 16 days. The TurboSMARTS sampling methodology [29] played a key role in accelerating this refinement process. While this methodology allows for complete, entire benchmark suite results to be generated within a few hours, the random sampling nature of TurboSMARTS revealed trends after only 15 to 30 minutes of simulation time. These trends were sufficient in many cases to indicate whether a change has achieved its goal. TurboSMARTS helped bring the modify-evaluate cycle to under an hour for many of the cases.

6.2 Current model

The results of the refinement are given in Figure 8. The breakdown of performance constituents is given in Figure 9. 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 10 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 (more than

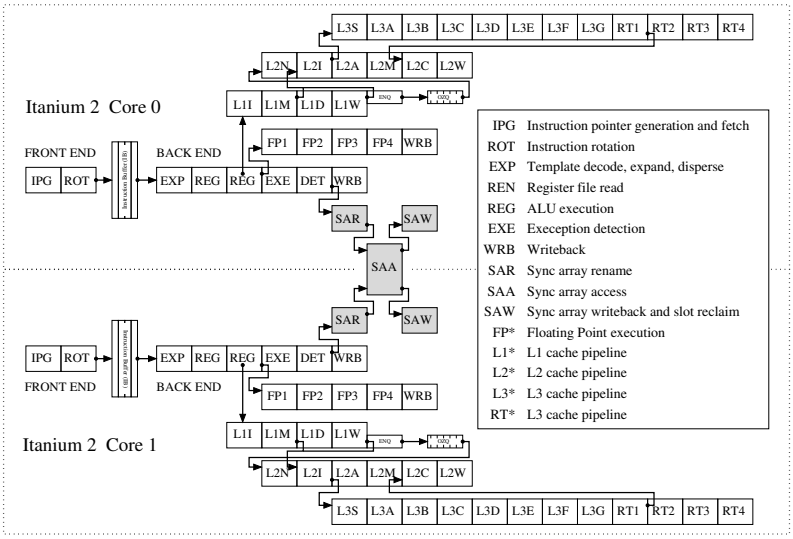


Figure 11: Dual-core CMP configuration with Synchronization Array. The individual cores are Itanium 2 cores. Shaded pipeline stages indicate extensions to the Itanium 2 datapath to support synchronization array instructions.

half of which was spent in investigation), overall constituent error decreased by 34%.

Notice that while certain benchmarks show worse overall behavior in this new model (though overall CPI error is reduced), the reduction in constituent errors makes it much more likely that the model will produce reliable conclusions when exploring new ideas which have impact on our targeted constituents. 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.

7 Using the Initial Model

Thus far, we have seen evidence that the Liberty modeling methodology allows rapid construction of validated models. However, while the refinement process provides evidence that these models are also flexible, the true test is to use these models to explore a wider space. We recount here two independent user experiences: one user (the original modeler) explored a novel pipeline organization, while the another user (previously unfamiliar with the model) used the model to explore a technique called dynamic software pipelining.

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 was made in only 4 weeks, 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 (CMP) interconnection mechanism. Specifically, he modeled a CMP system with a synchronization array (SA) to execute decoupled software pipelined (DSWP) codes [13]. Decoupled

software pipelining is a static multithreading technique that exposes thread-level parallelism in programs. In a dual-threaded decoupled software pipelined execution, one thread runs ahead, computes data values, and communicates them to another concurrently executing thread through the SA, a decoupling buffer [13]. The SA is part of the architectural state along with registers, program counters and memory. DSWPed codes use new `produce` and `consume` instructions that access the SA to effect all inter-thread communication and synchronization. The resulting CMP configuration is shown in Figure 11.

To build this model, the user performed the following tasks:

- Define functional semantics for the new `produce` and `consume` instructions and modify the pipeline to recognize these new instructions.
- Augment the Itanium 2 datapath to handle the synchronization array instructions (shown in dark in Figure 11)
- Package the main Itanium 2 pipeline and the L1 caches into a processor core template
- Instantiate two cores connecting them with the synchronization array and a shared L2/L3 memory hierarchy.

Finding unused instruction encodings in the IA64 ISA and adding them to the DSWP CMP model via the functional emulator extension interface was a fairly straightforward task, involving only an hour's work by the user. This functional emulator extension interface provides a transparent way to define new instructions. In this way, the base validated IA64 functional emulator can remain untainted by this specific experiment. Once the instruction decode semantics were defined, the *SA-decoder* was added to the end of the EXP stage to provide correct decode information for SA instructions. The SA-decoder was modeled with a simple pass-through module invoking functional emulator API calls to compute the decode information for SA instructions alone and letting the other instructions pass through untouched.

Once this was done, `produce` and `consume` instructions could flow through the Itanium 2 processor pipeline while almost all the bookkeeping (like register renaming, scoreboarding, and bypass calculation) for the new instructions is managed *automatically*. The only exception to this was the addition of 10-15 lines of behavioral code to handle scoreboarding for predicated `consume` instructions, which had slightly different writeback behavior than regular predicate writers in the IA64 ISA. Discovering and fixing this corner case, a case which would exist in real hardware, took a total of two days.

The next major step was adding the SA instruction datapath. Writing the synchronization array model and the synchronization array renamer module took one full day. The `produce` and `consume` instructions are treated as memory operations in the backend until the beginning of the EXE stage. The baseline model handles this automatically. At the beginning of the EXE stage, the SA instructions have to be steered into the SA datapath. They are then staged through two pipeline latches. Only non-speculative predicate-on SA instructions get renamed at the SA Rename (SAR) stage. These instructions then go on to access the synchronization array at the SAA stage. Finally, at the SAW stage, `produce` and `consume` instructions write back, mark themselves as "completed", and reclaim their synchronization array slots. Once the SA datapath was added, the user found it straightforward to package the augmented Itanium 2 pipeline and the L1 caches into a *dswp_core* module. The final CMP configuration was put together with two such cores, a shared L2/L3 hierarchy and the synchronization array.

Overall, it took the second user two weeks to build a fully working CMP model to execute DSWP codes. This time included both the development/debugging time for the model configuration and time spent debugging errors in the compiler-generated DSWP binaries. Only two custom modules (recall that a module is an LSE component template) were written

specifically for this project, the SA Renamer and the SA itself. All other components were customized and reused from LSE's standard module library. The extremely generic nature of the key pieces of logic in the Itanium 2 pipeline model, like the register remapping, scoreboarding, and bypassing, made the introduction and handling of new instructions very straightforward for the user. The second user found the model sufficiently well-designed as to allow introduction of new back-pressure stalls at the SAR stage without breaking the model in any way for execution of non-DSWP codes. For DSWPed codes to work, a minor modification had to be done in the misspeculation-handling logic. The clear signals for each of the instructions in the REN-REG and REG-EXE pipeline latches had to be logically ANDed with the output of a comparator that checked if the particular instruction is younger than the instruction redirecting the control flow. This modification was important in the augmented Itanium 2 model because back-pressure created in the SAR stage could force SA instructions to remain in these latches until the back-pressure eases, even though branches from the respective instruction groups could have been resolved long ago, a condition not seen in the baseline configuration. The user felt that support for hierarchical modeling and the structural nature of LSE modules made the transition from a uni-processor configuration to a multi-processor configuration with various shared structures very easy. This experience indicates that the validated Itanium 2 model, despite being very detailed, can be reused, customized and extended in interesting ways with minimal effort, allowing rapid exploration of multiple design points.

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 [10, 5], too difficult to develop due to lack of published information [9], or too time-consuming to modify for wide ranging design-space exploration [10]. Some have claimed that the Liberty modeling methodology [8, 11, 12] addresses these issues, but only anecdotal and secondary evidence exists to support it.

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 Vachharajani et al.'s first set of claims: validated models can be constructed rapidly.

We learned three lessons the hard way. First, we learned that documentation is often in error and sometimes 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 learned (and show in this paper) that building a model that is validated according to a single aggregate metric does *not* necessarily provide a credible baseline or an adequate platform 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 6 person-weeks in total. The speed with which these modifications were made is support for Vachharajani et al.'s second claim: that validated models built using the Liberty modeling methodology can be rapidly modified for design-space exploration.

References

[1] A. Ghosh and T. Givargis, "Cache optimization for embedded processor cores: An analytical approach," *ACM Transactions on Design Automation of Electrical Systems (TODAES)*, vol. 9, pp. 419–440, October 2004.

[2] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pp. 52–62, November 1994.

[3] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood, "Analytic evaluation of shared-memory systems with ILP processors," in *Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA)*, pp. 380–391, April 1998.

[4] H. S. Shahhoseini, M. Naderi, and S. Nemati, "Achieving the best performance on superscalar processors," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 4, pp. 6–11, 1999.

[5] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, pp. 59–65, May 1998.

[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)*, pp. 49–58, November 2000.

[7] 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)*, pp. 266–277, July 2001.

[8] 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)*, pp. 271–282, November 2002.

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

[10] 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*, vol. 30, pp. 2–4, March 2002.

[11] 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)*, pp. 195–206, June 2004.

[12] M. Vachharajani, *Microarchitectural Modeling for Design-space Exploration*. PhD thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, November 2004.

[13] 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*, pp. 177–188, September 2004.

[14] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pp. 84–97, June 2003.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[16] D. Penry and D. I. August, “Optimizations for a simulator construction system supporting reusable components,” in *Proceedings of the 40th Design Automation Conference (DAC)*, June 2003.

[17] M. Vachharajani, N. Vachharajani, S. Malik, and D. I. August, “Facilitating reuse in hardware models with enhanced type inference,” in *Proceedings of the 2004 Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, September 2004.

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

[19] M. S. Schlansker and B. R. Rau, “EPIC: Explicitly parallel instruction computing,” *Computer*, vol. 33, no. 2, pp. 37–45, 2000.

[20] D. Weaver, *SPARC-V9 Architecture Specification*. SPARC International Inc., 1992.

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

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

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

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

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

[26] C. McNairy and D. Soltis, “Itanium 2 processor microarchitecture,” *IEEE MICRO*, vol. 23, pp. 44–55, March-April 2003.

[27] 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*, vol. 37, pp. 1433–1440, November 2002.

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

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