MonteSim: A Monte Carlo Performance Model for In-order Microachitectures

Ram Srinivasan
New Mexico State University
ram@nmsu.edu

Olaf Lubeck
Los Alamos National Laboratory
olubeck@lanl.gov

***Abstract*— In this paper, we present a predictive Monte Carlo based performance model for in-order microarchitectures that is validated against the Itanium-2 processor. In such architectures, we find that application specific characteristics such as *load* carried dependence and prefetching significantly impact performance. We parametrize these effects and use the PIN instrumentation tool to obtain them. We use Monte Carlo sampling techniques to model the processor core, memory hierarchy and application characteristics. These techniques are widely used in physical simulations but their application to computer architecture performance modeling is atypical and the application parameterization used in this work is also believed to be novel. Preliminary results indicate that the model predicts CPI with a high degree of accuracy as validated against real measurements. Unlike detailed cycle accurate simulation which is computationally infeasible for evaluating realistic scientific applications, the proposed Monte Carlo model converges to a prediction in a few seconds. The accuracy of the model given its simplicity is surprising.**

I. Introduction

Processor performance evaluation is a vital component in the design of future processors. Traditionally, cycle-accurate simulators have dominated the mainstream due to their inherent flexibility and accuracy. However, their long execution time is seen as a major drawback to a thorough design-space exploration. For example, cycle-accurate simulation of the complete SPEC CPU2000 suite using SimpleScalar [6], a popular academic simulator which models a relatively simple MIPS R10K pipeline, takes more than five processor months on a modern desktop. In the initial stages of design exploration, such high simulation time is often unaffordable. Rather a quick measure of relative performance improvements is often sufficient and as the design matures, cycle accurate simulation can be used for more precise performance estimates. In this paper, we propose a Monte Carlo performance model of the Itanium-2 processor. The model heavily relies on micro-architecture independent benchmark profiles that are easily obtained through binary instrumentation tools such as PIN [7].

We observe that scientific applications typically operate on large data sets and for such applications the memory hierarchy induced stalls are a more significant contributer to overall Cycles-Per-Instruction (CPI) than intrinsic pipeline stalls. Moreover for in-order architectures such as the Itanium, intrinsic pipeline stalls can be easily computed through dependence analysis. Therefore, we abstract out the details of the core CPU pipeline and model the memory hierarchy in significant detail. Preliminary results indicate that the model typically predicts CPI to within 10% of the measured value and does so in a few seconds even for large applications. The rest of this paper is organized as follows: Section II describes the experimental methodology, Section III explains the features of the Itanium architecture pertinent to our model, Section IV describes the model, Section V describes the initial results and, Section VI states our conclusions.

II. Experimental Methodology

All experiments were carried out on *bench1*: A dual Itanium-2 processor, shared memory system running Linux v2.6. Each processor operates at 1.3GHz and incorporates a 3MB unified level-3 cache. The memory access latency is 260 clocks. All OS related activities including interrupt handling are isolated to a single processor, leaving the other processor relatively clean for measurements. We use one LANL HPC benchmark:*sweep3d* and eight SPEC CPU2000 FP benchmarks [1] in the study. The benchmarks were compiled using the Intel ECC 7.0 compiler at -03 optimization. The PIN binary instrumentation tool [7] was used to gather application profiles while the PFMON tool [2] was used to access the hardware performance counters. Sensitivity studies were done on *hertz*: A four processor, Itanium-2 SMP, running Redhat Linux v2.4. Each processor in the machine operates at 900MHz and incorporates a 1.5MB level-3 cache. Due to improvements in the memory controller, the memory access latency in this machine is 112 clocks. The peak memory-cache transfer bandwidth on both *bench1* and *hertz* is 6.4GB/sec.

III. Itanium architecture

The Itanium employs an eight stage, in-order, VLIW pipeline capable of issuing 6 instructions every cycle. However, this issue rate decreases in the presence of *group stops* [4], [3]. A group is a set of instructions that have no data hazards and could potentially be issued simultaneously. The architecture mandates that dependent instructions be placed in different groups. When a group stop is encountered, instruction issue for the current cycle stops and the next group begins issue from the subsequent cycle. The Itanium-2 has the following fully pipelined execution units: 2 integer, 2 floating-point, 4 memory and 3 branch. The cache hierarchy in the Itanium-2 is fully-pipelined and non-blocking, the details of which are shown in Table I. Floating-point loads and all store instructions bypass the L1 cache and directly access the L2 cache.

In VLIW architectures, the compiler is responsible for arranging instructions to avoid stalls. The Itanium pipeline

Level	Characteristics
L1 instruction	16KB, 64 byte line, 1 cycle
L1 data	16KB, 64 byte line, 1 cycle
L2 unified	256KB, 128 byte line, 6+ cycle for FP data
L3 unified	3/6/9MB, 128 byte line, 16+ cycles for FP data

TABLE I

CACHE HIERARCHY OF AN ITANIUM-2 PROCESSOR

will stall instruction issue on all unresolved RAW and WAW dependence. The architecture provides many advanced features such as control/data speculation, automatic register rotation for software pipelined loops, predication and explicit data/instruction prefetch instructions. These features allow the hardware to exploit ILP and help hide memory latency. Compilers are usually very effective at scheduling instructions to hide deterministic and often small latencies of the execution units. However, load-dependent instructions pose a problem. The latency associated with a load is often variable, depending on where in the memory hierarchy the load is satisfied and this can be as high as a few hundred cycles when the load misses all levels of the cache. Compilers are not very successful in hiding such large latencies due to limitations in the static predictability of such events. Figure 1 shows the percentage of overall CPI that can be attributed to memory stalls. The study was done on *bench1* and is explained further in Section V.

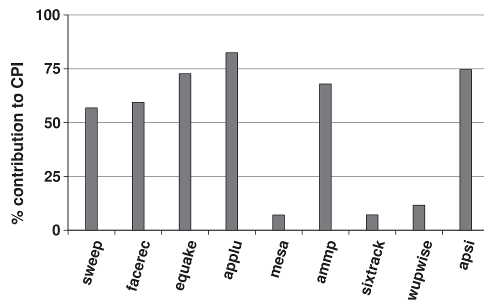

Fig. 1. Contribution of stalls induced by load-dependence to overall CPI

In six of the nine benchmarks, we observe that greater than 60% of the overall CPI is due to memory induced stalls. This can be as high as 80% in benchmarks such as *applu*. If the compiler schedules the load instructions early so as to separate the load and its consumer by several hundred instructions, memory latency can be completely hidden and instruction issue will not stall at the dependence. However, such early scheduling of loads is not possible in many scenarios such as pointer referenced data accesses. Thus, memory induced stalls can have a significant effect on performance in data intensive applications.

IV. THE MODEL

The model aims to accurately predict CPI for a given application of the Itanium ISA. The proposed model is based on the relation, CPI = CPI_0 + CPI_{ms}, where CPI_0 is the CPI of an application with an infinite L1 data cache and CPI_{ms} is the contribution of memory stalls to the overall CPI. Since scientific applications have low branch-mispredictions and I-cache miss rates, CPI_0 can be easily determined either through static code analysis or through regression techniques [8]. Note that CPI_0 has a theoretical lower bound of $\frac{1}{6}^{th}$ for the Itanium-2. This corresponds to the maximum issue rate of 6 instructions every cycle. The prediction of CPI_{ms} on the other hand is non-trivial due to the non-blocking nature of the Itanium-2's memory hierarchy and its ability to handle multiple loads under miss. Our model uses probability distributions of application characteristics and statistical methods based on Monte Carlo sampling to predict CPI_{ms} and hence the total CPI for a given application. We do not perform a discrete-cycle (or cycle-by-cycle) simulation wherein, every cycle of the processor is simulated. Instead, cycle accounting is performed based on probabilistically generated events.

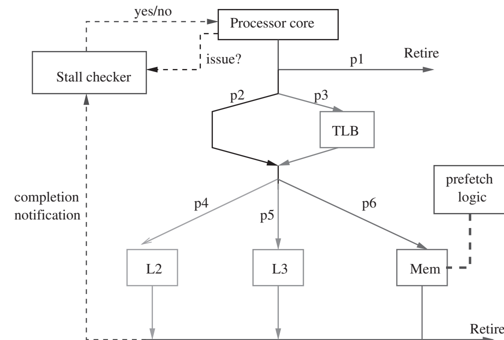

Fig. 2. The Itanium-2 performance model

Figure 2 gives an overview of the model. In this Monte Carlo model, the processor's core pipeline is treated as an abstract generator of tokens. This token generator nominally emits one token every CPI_0 cycles. The emitted tokens belong to one of two instruction classes: load or non-load. The non-load tokens are emitted with a probability of p_1 and these tokens retire immediately. Load tokens proceed through the memory hierarchy before retirement. This process of token emission can however stop when the *stall checker* detects a memory hierarchy induced hazard. The memory hierarchy components of the Itanium-2 are modeled as delay centers with deterministic service times. The delay-centers correspond to the L2 DTLB, L2 cache, L3 cache and system memory. The service time of the TLB, L2 and L3 delay centers are fixed at 31, 6, 16 cycles respectively and are obtained from the processor manuals [4], [3]. Memory access latency, on the other hand, is a system specific parameter. It is affected by prefetching, and is variable as explained later in this section. For every token issued to the memory delay center, the prefetch-logic block is invoked to determine the effective delay experienced by the token. After passing through the delay centers, the load tokens retire and, in the process, inform the stall checker of their completion. The following are the simplifying assumptions made in this model:

- we ignore the finite buffers at each level of the Itanium's memory hierarchy and assume that each delay center is capable of handling an unbounded number of requests.
- L2 TLB misses are satisfied in the L3 cache. That is, the page-tables are assumed to reside in L3 cache.

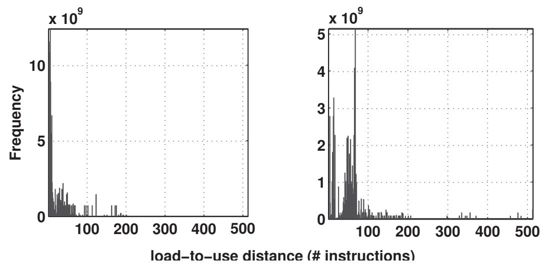

Fig. 3. Load-to-use distance histogram for *equake* (left) and *apsi* (right)

The transition probabilities, $p2 - p6$, correspond to the probability that a load is satisfied in the given level of the hierarchy. These directly correspond to cache and TLB miss rates and are presently obtained from hardware performance counters. Table II shows the calculation of p from the performance counter data. We can also leverage existing research for predicting cache hit/miss rates based on data-reuse characteristics of an application [5], [9]. Though *store* instructions access the

Probability of ..	Symbol	Formula
Non-load	p_1	$\frac{ia64_inst_retired - loads_retired}{ia64_inst_retired}$
L2 TLB miss	p_3	$\frac{l2dtlb_misses}{mem_ops}$
L2 TLB hit	p_2	$1 - p_3$
Load satisfied in L2	p_4	$\frac{mem_ops - l2_misses}{mem_ops}$
Load satisfied in L3	p_5	$\frac{l2_misses - l3_misses}{mem_ops}$
Load satisfied in Memory	p_6	$1 - (p_4 + p_5)$

TABLE II
CALCULATION OF TRANSITION PROBABILITIES FROM ITANIUM-2'S HARDWARE PERFORMANCE COUNTER EVENTS. NOTE, $mem_ops = loads_retired + stores_retired$

memory hierarchy and contend for resources, we treat them like any other non-load instruction that take the path along $p1$.

A. Stall checker

With no load-use dependence, tokens are emitted at the rate of one every CPI_0 cycles and thus, $CPI=CPI_0$. However, the presence of load dependence can stall the pipeline until the corresponding *load* completes. Such a pipeline stall forces all instructions following the dependent instruction to wait for the *load*. Since, stalls are directly caused by the instruction that uses the load data, we use an application specific metric to characterize this effect. The metric is called load-to-use distance and is defined as the number of instructions that separate a *load* and its dependent instruction. If this distance is large, the dependent instruction may not be encountered until the *load* completes, thereby completely hiding the memory hierarchy latency. Conversely, a small dependence distance is likely to result in frequent memory hierarchy induced stalls and hence larger is the overall CPI. Since a load-to-use distance is associated with every executed load instruction of the application, a histogram of the distances seen over the complete execution is captured. This is called the *load-to-use* histogram. This histogram can be easily generated through binary instrumentation tools such as PIN or through a combination of static code analysis and function call profiles from tools like *gprof*. In this work, we use the PIN tool to capture the load-to-use histogram.

Figure 3 shows the load-to-use distance histograms for two SPEC benchmarks *equake* and *apsi*. *Equake*'s histogram exhibits very high density at small load-to-use distances, while *apsi* has high density concentration in the distance range [50-100]. Thus, we would expect the average number of stall cycles per committed load to be lower in *apsi* than in *equake* due to its ability to hide memory hierarchy latency better. This is observed to be true, with *apsi* stalling on an average of 2.44 cycles for every committed load and *equake* stalling 5.69 cycles per load. These numbers are determined using the model and are explained further in Section V.

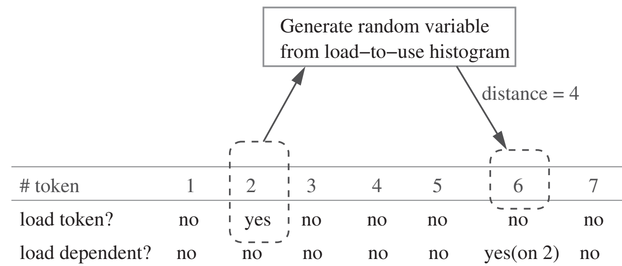

Fig. 4. Illustrates how the load-to-use histogram is used by the simulator

Figure 4 illustrates how the load-to-use histogram is incorporated into the simulator. For every *load* token generated by the core, the simulator samples the load-to-use histogram to determine the instruction distance to the dependent token. This information is passed on to the stall-checker that tracks all issued *load* tokens and their dependences. In our example, the stall checker records that token#2 is a load token and, at a distance of four tokens (or token#6), the load data is consumed. After issuing the *load* token, the processor core continues to emit one token every CPI_0 cycles, until the load-dependent token#6 is reached. At this point, the stall checker instructs the processor core to stall token issue if the producing load (or token#2) is still in flight. Based on the path taken by token#2 in the memory hierarchy, we can compute its completion time and hence the number of cycles for which token#6 must stall. For example, if token#2 hit in TLB (along path p_2), missed in L2 and hit in L3 (along path p_5), the load will not complete until 16 cycles after it was issued. Therefore, token#6 would be forced to stall for ($16 - 4 \times CPI_0$) cycles. After stall completion, subsequent tokens continue to be issued at the rate of one every CPI_0 cycles.

B. Prefetch logic

Prefetching is another aspect of the architecture that can greatly influence performance. Based on how far in advance prefetching is done prior to the load, the service time of the memory delay center changes. This change in service time is determined by the prefetch logic using application specific parameters. The Itanium architecture supports both explicit and implicit data prefetch operations. Implicit data prefetch is done in the *base_update_form* of memory related instructions, while, explicit data prefetch happens when an *lfetch* instruction

is issued [4], [3]. Prefetching can have two effects depending on how early the prefetch is issued relative to its target load. If prefetching is done early enough to provide sufficient time for the data transfer from memory to cache, we call it *early-prefetch*. In early-prefetch, the corresponding *load* instruction will either hit in cache if the prefetched line is not evicted or will miss in cache if it was evicted. The transition probabilities, derived from the cache hit rates, automatically capture this effect. The second scenario is referred to as *late-prefetch* and happens when the prefetch is issued in close proximity of the target *load*. In this case, the *load* will miss cache, but its effective memory access time is decreased due to the prefetch. It is possible that a late-prefetch will have a detrimental effect on performance. This scenario results from the fact that the target load must wait for the duration of a full cache line transfer. In the case of a load with no prefetch, the critical word is delivered directly to a register and the dependent instruction using the load-data need not wait for the entire cache line. The difference in wait time is due to the finite bandwidth between memory and cache. To illustrate this effect, we use synthetic assembly traces to measure the memory access time (in cycles), first without prefetch (*no pf*) then with an explicit prefetch issued one cycle prior to the load instruction (*late pf*). This was done for both *bench1* and *hertz* and the data is shown in Table III. With late prefetch, the access time is longer. For example, in *bench1*, when no prefetching is done, the load causes a 260 cycle stall. Due to the critical-word-first order of data delivery, this stall is equal to the memory access latency. When a prefetch is issued one cycle prior to the load, the load experiences a 290 cycle stall. The 30 cycle additional wait, in this case, corresponds to the time it takes to transfer one line of data (128 bytes) from memory to L2 cache.

machine	no pf	late pf
bench1	260	290
hertz	112	131

TABLE III
MEMORY ACCESS TIME (IN CYCLES) WITH NO PREFETCH (NO PF) AND PREFETCH ISSUED ONE CYCLE PRIOR TO THE LOAD (LATE PF)

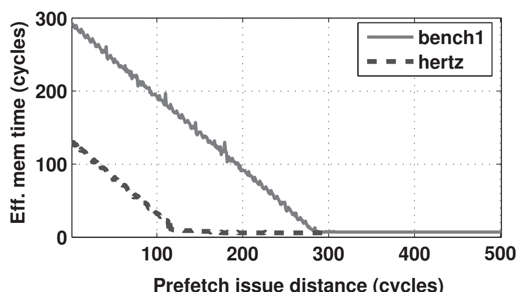

Fig. 5. Effect of late prefetch on memory access time

Using the synthetic assembly trace, we next analyze the effect of increasing the separation between the prefetch and the load instruction. This can be easily achieved by incrementally inserting NOP bundles (with group stops) between the prefetch and the load instruction. Each NOP bundle adds a cycle of separation. As the separation is increased, we expect the memory access time to proportionally drop since the prefetch instruction initiates the data transfer from memory to cache, prior to the load. Figure 5 illustrates this effect for *bench1* and *hertz*. Clearly as the separation increases, the effective memory access time drops with a slope of -1. This trend continues until the load instruction hits in L2 cache due to the completion of the prefetch operation, after which, the access time becomes 6 cycles. Therefore the service time of the memory delay center is variable in the range [0-290] for *bench1* and [0-131] for *hertz*.

Clearly, the effect of prefetching on the load's memory access time depends on the distance between the prefetch and the corresponding load instruction. Again this is an application specific parameter and we capture this using a *prefetch-to-load* distance histogram. A prefetch-to-load distance is a count of the number of instructions between a prefetch and the matching *load* instruction to the same cache line. Just as the load-to-use distance, the prefetch-to-load distance is also a micro-architecture independent metric that can be easily captured through the PIN tool. Figure 6 shows the cumulative

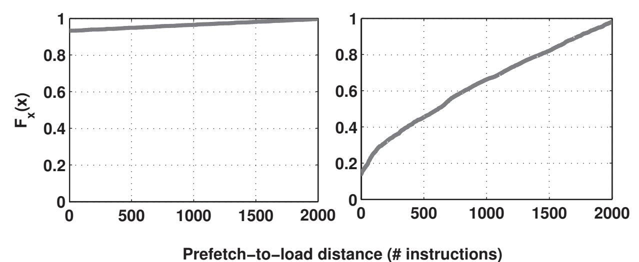

Fig. 6. Cumulative distribution function of prefetch-to-load distance for *equake* (left) and *apsi* (right)

distribution function (CDF) of the prefetch-to-load distance.

A distance of zero implies no prefetch was issued. For example, in *equake*, greater than 90% of the load instructions have no prior prefetch issued, while it is only 10% in *apsi*. This further explains *apsi*'s lower cycle cost per load.

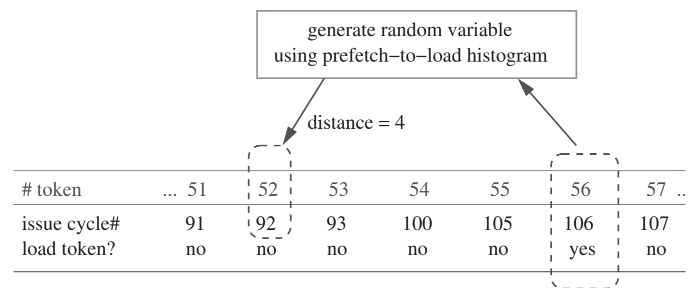

Fig. 7. Working of the prefetch logic

Figure 7 illustrates how the simulator uses the prefetch-to-load distance histogram. For every load token proceeding along path p_6 (in Figure 2), the prefetch-logic samples the prefetch-to-load distance histogram to determine how far in

advance a prefetch was issued, if any. Using the upstream history of when every prior instruction was issued, the prefetch-logic converts this distance into cycles. The distance in cycles can then be used by the simulator to determine the service time that the load token experiences at the memory delay center. For example, in Figure 7, token#56 that is issued in cycle 106 is assumed to be a load token which misses in cache and accesses memory. At this point, the prefetch-logic samples the prefetch-to-load distance histogram and generates a prefetch distance of 4. Therefore, token#52 is assumed to be a prefetch request and the issue cycle of this token is used to compute the distance in cycles to the load (token#56). In this example, it is 14 cycles. This distance is then mapped to a memory service time that the load token will experience. Figure 8 illustrates this mapping

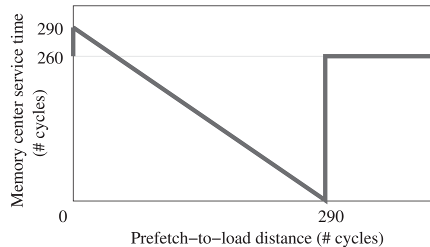

Fig. 8. Effect of prefetch-to-load distance in cycles on memory service time in *bench1*

for *bench1*. The x-axis shows the prefetch-to-load distance in cycles and the y-axis represents the effective delay at the memory center. With no prefetching (indicated by a prefetch distance of 0), the effective memory service time is 260 cycles. With prefetching, the memory service time drops proportional to the prefetch-to-load distance as seen in Figure 5. Beyond a distance of 290 cycles, the service time jumps back to 260 cycles. This covers the case where the prefetch operation completes but the cache line is subsequently evicted. Since we have already determined, based on transition probabilities, that the current load-token missed in cache and is requesting data from memory, we must infer that the prefetched line has been evicted from cache, and the load incurs a full latency of 260 cycles.

V. RESULTS

The model parameters are CPI_0, transition probabilities, and prefetch-to-load and load-to-use histograms. CPI_0 is obtained using the regression method described in [8]. For each benchmark, CPI and cache hit data are obtained from 10 runs, each with different input sets. The runs are used for the regression. As mentioned before, CPI_0 can be also be determined by dependence analysis of the binary. The transition probabilities are computed from cache, TLB hit rates and are obtained through the PFMON tool. The prefetch-to-load and load-to-use histograms are obtained using the PIN tool. The simulator, written in C, uses these parameters as explained in Section IV, to predict CPI. The simplicity of the simulator makes it extremely fast and converges to a result in a few seconds. For example, Figure 9 shows the variation of CPI for the first 3 Million tokens generated for *apsi*. We notice that CPI reasonably converges after just 1 Million tokens. The simulation is continued until the change in CPI over two successive intervals (of a million tokens each) is no greater than 10^{-4}.

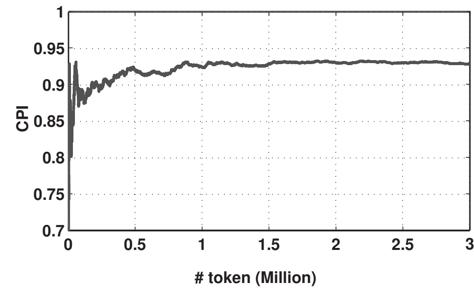

Fig. 9. CPI convergence in *apsi*

First, we analyze the ability of the model to predict CPI. Figures 10 and 11 show CPI_0, measured CPI and the predicted CPI for various benchmarks on *bench1* and *hertz*. CPI_0 for

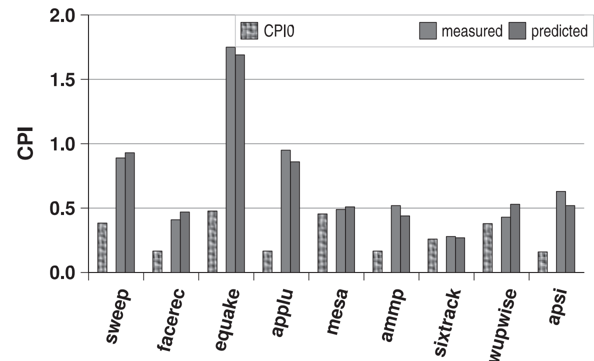

Fig. 10. CPI predictions for *bench1*

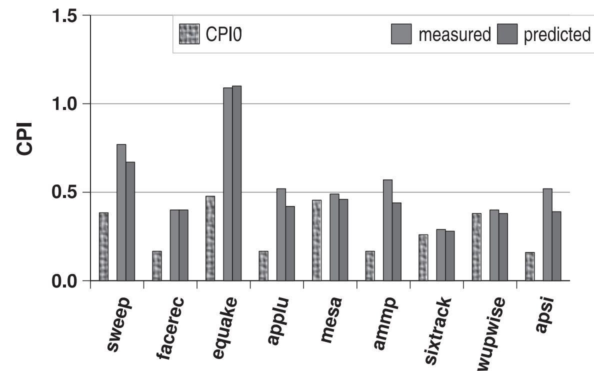

Fig. 11. CPI predictions for *hertz*

three of the benchmarks is the minimum possible of 0.17 and the average across all benchmarks is just 0.29. This is due to efficient code scheduling by the compiler and the inherently low I-cache and branch prediction miss rates of scientific applications. We observe that predicted CPI closely tracks the measured CPI. In *bench1*, a maximum difference of 0.11 between predicted and measured CPI was observed for *apsi*. In *hetrz*, both *ammp* and *apsi* exhibit the maximum

deviation of 0.13. Typically the error in CPI prediction is less than 10%. However, in *hertz*, the benchmarks *ammp*, *apsi* and *applu* exhibit higher errors. We are presently investigating the reason for this. Using CPI_0 and measured CPI, we can easily compute the fraction of CPI due to memory hierarchy induced stalls as was illustrated earlier in Figure 1. In applications like *equake*, memory hierarchy stalls can significantly affect CPI, while in applications like *sixtrack* almost all of the CPI is due to inherent pipeline stalls.

The model can also be used to easily determine the average effective latency experienced by loads at each level of the memory hierarchy. That is, the CPI_{ms} in our fundamental equation, CPI = CPI_0 + CPI_{ms}, can be further decomposed into, $CPI_{ms} = h_2 t_2 + h_3 t_3 + h_m t_m$, where h_x indicate the hit rate to each level of memory hierarchy and the t_x correspond to the effective stall time for accessing a particular level of the hierarchy. Note that t_x can be very different from the fixed access latency of each level of the memory hierarchy (Table I) due to the latency hiding features of the architecture and compiler. The ts are a good indicator of how well the architecture/compiler is able to hide the latency associated with each level of the hierarchy. Table IV lists the ts for various benchmarks on *bench1* and *hertz*. From the t_2 column of both machines, we observe that on an average 75% of the 6 cycle L2 latency is hidden by the architecture/compiler. Similarly we observe that about 60% of the 16 cycle L3 latency and less than 20% of the memory latency is hidden in both machines. The lp column is the average number of cycles for which every executed load stalls. This is obtained by dividing CPI_{ms} by the number of load tokens generated. Also, the ratio of CPI_{ms} to CPI indicates the fraction of overall CPI that can be attributed to memory-hierarchy stalls. This ratio was illustrated earlier in Figure 1.

VI. Conclusion

Despite many innovative features in modern processors, memory latency is a major obstacle to exploiting ILP. Memory induced stalls account for a big fraction of overall CPI in many scientific applications. We build a Monte Carlo performance model of the Itanium-2 to predict CPI. Using cache hit/miss rates and load, prefetch characteristics, the model predicts CPI typically to within 10% of the measured value. The inherent model simplicity makes it extremely fast and well suited for performance studies, particularly, of evolving architectural features. For example, we intend to extend and apply the model to multi-core chips where the basic core is unchanged. The accuracy of the model given its simplicity is surprising. In the future, we would like to extend the model by:

- using data-reuse distance to predict cache/TLB miss rate and hence the ps of the model.
- adding support for chip-multiprocessing (CMP).
- applying the model to non-scientific applications.

VII. Acknowledgments

We thank our fellow members of the CCS-3 PAL team at LANL for their suggestions that helped improve this work. In particular, we extend our gratitude to Greg Johnson, Micheal Lang, Eitan Frachtenberg and Scott Pakin for helping us understand the effects of prefetching on the Itanium-2 better and for helping us with the Linux OS.

App	Bench1				Hertz			
	t_2	t_3	t_m	lp	t_2	t_3	t_m	lp
sweep	0.26	2.00	211.19	4.08	0.24	1.96	79.47	2.13
facerec	1.34	7.83	234.27	2.77	1.32	7.86	96.16	2.17
equake	2.26	8.66	225.85	5.69	2.25	8.38	91.80	2.93
applu	2.37	6.53	200.87	5.98	0.87	5.48	84.37	2.50
mesa	2.63	9.20	242.13	0.56	2.73	9.96	102.63	0.52
ammp	2.91	9.08	233.85	2.80	2.72	8.84	90.00	2.77
sixtrack	0.36	3.18	231.95	0.10	0.51	5.35	83.68	0.38
wupwise	0.99	4.32	222.12	1.59	1.11	6.11	91.67	0.92
apsi	1.16	3.95	202.77	2.47	1.03	3.80	81.50	1.43

TABLE IV

The ts indicate the effective access time at L2 cache, L3 cache and memory. The lp column indicates the average number stall cycles per committed load. Both t and lp are in cycles

References

[1] CPU2000 benchmark suite from Standard Performance Evaluation Corporation. *http://www.spec.org/cpu2000/*.

[2] HP pfmon tool. *http://www.hpl.hp.com/research/linux/perfmon/pfmon.php4*.

[3] Intel Itanium-2 processor reference manual for software development and optimization. *http://www.intel.com/design/itanium2/documentation.htm*.

[4] Intel Itanium Architecture software developer's manual. *http://www.intel.com/design/itanium2/documentation.htm*.

[5] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. *In Proceedings of PDCS'01, pages 617-662, Aug 2001*.

[6] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *Technical Report 1342, Computer Sciences Department, University of Wisconsin*, June 1997.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[8] Y. Luo, O. M. Lubeck, F. Wasserman, Harvey J.and Basseti, and K. w. Cameron. Development and Validation of Hierarchical Memory Model Incorporating CPU- and Memory Operation Overlap. *In the proceeding of the first international workshop on software and performance*, October 1998.

[9] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. *In Proceedings of the joint ACM SIGMETRICS-Performance 2004 Conference on Measurement and Modeling of Computer Systems*.