Pinpointing Representative Portions of Large Intel® Itanium® Programs with Dynamic Instrumentation

Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi

Intel Corporation

{harish.patil, robert.s.cohn, mark.charney, rajiv.kapoor, andrew.y.sun, anand.karunanidhi}@intel.com

Abstract

Detailed modeling of the performance of commercial applications is difficult. The applications can take a very long time to run on real hardware and it is impractical to simulate them to completion on performance models. Furthermore, these applications have complex execution environments that cannot easily be reproduced on a simulator, making porting the applications to simulators difficult. We attack these problems using the well-known SimPoint methodology to find representative portions of an application to simulate, and a dynamic instrumentation framework called Pin to avoid porting altogether. Our system uses dynamic instrumentation instead of simulation to find representative portions — called PinPoints — for simulation. We have developed a toolkit that automatically detects PinPoints, validates whether they are representative using hardware performance counters, and generates traces for large Itanium® programs. We compared SimPoint-based selection to random selection of simulation points. We found for 95% of the SPEC2000 programs we tested, the PinPoints prediction was within 8% of the actual whole-program CPI, as opposed to 18% for random selection. We measure the end-to-end error, comparing real hardware to a performance model, and have a simple and efficient methodology to determine the step that introduced the error. Finally, we evaluate the system in the context of multiple configurations of real hardware, commercial applications, and industrial-strength performance models to understand the behavior of a complete and practical workload collection system. We have successfully used our system with many commercial Itanium® programs, some running for trillions of instructions, and have used the resulting traces for predicting performance of those applications on future Itanium processors.

1. Introduction

Designers rely on detailed simulation for evaluating features in future microprocessors. It is important to choose workloads for simulation that represent the increasingly complex and challenging problems future microprocessors must execute efficiently. Running real programs to completion on an accurate performance model is not practical because it can require months of execution time. Instead, microprocessor architects choose only a small portion of the execution of a program to run on a detailed cycle-accurate model and use extrapolation to predict the behavior of the entire program. Selecting representative portions of programs is difficult because the behavior of programs changes over time.

Another obstacle to obtaining representative workloads for simulation is the difficulty in reproducing the complex execution environments required by real applications. Applications can have run-time license checking, special device drivers or other kernel dependencies, large storage requirements, and elaborate installation procedures. Whole-system simulators [1],[2] allow a high degree of control and visibility, but it is usually time consuming to get the latest versions of complex applications running properly on such a system.

Collecting workloads is a never-ending task. Different applications become important, existing applications change, and compilers are tuned. It is desirable to update the workloads to ensure that new microprocessor features are not fixing yesterday's problems. Unfortunately, the difficulty of getting a new application to work in the simulation environment combined with the complexity of identifying and capturing a representative portion of a program means important workloads are often missing or out-of-date.

The entire process of collecting performance-modeling workloads requires expertise in several domains: operating systems, simulation technology, benchmark analysis, and knowledge about exercising the application.

Our goal is to automate the workload-collection process as much as possible and to simplify any tasks that must be done manually. Our system does not require expertise in benchmark characterization or simulation technology nor does it require an elaborate setup. If you can run the program, you can collect appropriately-sized regions and be confident that they are representative of the whole-program behavior.

To achieve our goal, we employed two key technologies: the SimPoint methodology from Sherwood *et al.* [3] and the Pin dynamic instrumentation tool [5]. SimPoint uses an execution profile to identify representative portions (slices) of an application. These slices — or simulation points — are validated against whole-program behavior and can be used to collect instruction traces or to drive an execution-driven simulation. Pin is a flexible tool for instrumenting Itanium®/Linux programs at run time. Running a program under Pin only requires a small change to the command line, so it can easily use an existing setup for an application. Since Pin relies on direct execution, it is much faster than traditional simulation technology. We use Pin-based tools to collect SimPoint profiles, validation statistics, and instruction traces. We call the

Table 1: Some applications processed with PinPoints methodology

Application (# runs)	Description	# Retired Instructions (billions)
AMBER-sander (5)	Assisted Model Building with Energy Refinement: A suite of bio-molecular simulation codes (Version 8, March, 2004) from University of California (SF) [6].	1300 — 3900
Fluent(8)	Computational Fluid Dynamics code (version 6.1.22, June 2003) from Fluent Inc.[7].	141 — 3900
LS-DYNA(1)	A general purpose transient dynamic finite element analysis program (SMP 970.3858 release for Itanium® linux OpenMP version) from Livermore Software Technology Corporation [8]. Public-domain benchmark 3cars: 3 vehicle collision.	4900
RenderMan®	A photo-realistic rendering application (version 11.5, un-released) from Pixar [9].	340 —797
ASIM (1)	An Itanium performance model built using the ASIM [19] framework. We ran PinPoints steps on the performance model binary while it was simulating a trace from a commercial database system.	8600
SPECOMP2001 (11 - medium)	Medium workloads from SPEC's benchmark suite for evaluating performance based on multi-threaded OpenMP applications [10].	830 — 4800
SPEC2000 (48:*ref*)	SPEC CPU2000 suite [11].	2 —723

representative portions of programs we find *PinPoints* and call our technique the *PinPoints methodology.*

We have used our system to collect traces for many commercial applications — including a database system, a computational fluid dynamics modeler, and a molecular dynamics simulation package. Some of the applications we processed with PinPoints are described in Table 1, where the last column shows the range of retired instruction counts encountered for multiple inputs to the applications. The total trace-collection time for these applications ranged from a few hours (for SPEC) to a few days (Fluent) which we think is very reasonable, especially since all the steps are automated. We successfully analyzed an application with more than 25 trillion dynamic instructions, which is two orders of magnitude longer than the longest SPEC2000 program run. With this work, we have made the following contributions:

1. a demonstration of a practical system for workload characterization and tracing in a production environment
2. an evaluation of its effectiveness when applied to commercial applications on an industrial performance model
3. a quick-and-easy method to determine if the PinPoints actually represent whole-program behavior
4. methods for handling non-determinism caused by data speculation on the Itanium® architecture
5. validation of an Itanium-2 performance model using PinPoints
6. an extension of SimPoint techniques to handle multi-threaded programs and multi-input program runs.

2. Background

In this section we describe the underlying technologies for the toolkit: SimPoint and Pin.

2.1. SimPoint

We use SimPoint analysis tools to identify representative portions of programs. We explain the algorithm briefly in this section. See Sherwood *et al.* [3] for more information.

SimPoint methodology divides the program execution into intervals of equal instruction counts. Each interval is called a *slice*. A *phase* in the program is a collection of slices with similar behavior. A representative slice, or a simulation point, is chosen from each phase.

For each slice, a basic block vector is collected. The vector has one count associated with each basic block—the number of instructions in the block multiplied by the number of times the block is executed during the slice. SimPoint algorithm clusters the basic block vectors. It normalizes each vector, and uses a random projection to map it into a 15-dimension space. Reducing the dimension of the vector makes the analysis efficient. Next, the *K-means* clustering algorithm is used to assign slices to clusters. Some points are chosen randomly to be cluster centers, and each slice is assigned to the cluster with the closest center. After the first clustering is finished, the algorithm picks new cluster centers by finding the centroid of the clusters and repeats the clustering until there is no change or a fixed iteration count is reached. This process is repeated for K=1 through 10 clusters. The *Bayesian Information Criterion* is used to find the smallest K that finds a clustering within 80% of the best clustering. In the end, we get K clusters of slices and a representative (called a simulation point) is chosen from each cluster. The weight of a simulation point is the relative size of the cluster it represents.

We configured our tools to choose up to 10 slices of 250 million instructions. Even with our relatively large slice size, some programs exhibit wide variations in behavior from slice to slice. This makes it difficult to capture all the behavior of a program with just 10 points. Collecting more points would help, but would require too much simulation time. The choice of slice size of 250 million instructions was driven by an internal study we did with traces of size 100 million instructions. In the study we simulated the traces twice — once treating all *cold* misses as hits and the second time treating cold misses as ordinary misses and found the performance difference around 10% for SPEC but was up to 40% for some commercial programs. Of course, the performance with proper warm-up will be between the two extremes. We chose large 250 million instruction slices to ease warm-up problems — in the same spirit as Conte *et al.* [4] who proposed increasing *cluster size* to reduce *cold-start effect*.

Random sampling [4],[12] is an alternative that has the potential to provide higher accuracy with a smaller number of dynamic instructions. Wunderlich *et al.* [12], in a technique called SMARTS, use random sampling to collect 10000 slices of 1000 instructions each. Sampling from a much larger set of points in the program makes it easier to capture all the behavior of the program. However, a short slice size requires a warm-up period for processor structures such as the caches and branch predictors. It is difficult to know how many instructions are adequate for warm-up, and conservative estimates would unnecessarily increase simulation time. Wunderlich *et al.* also proposed using functional warm-up, where only the cache and branch predictors are simulated in the intervals between slices. This method of warm-up would require extensive changes to the performance models that consume the traces we generate. For these reasons, we do not believe that fine-grain sampling is an appropriate choice for the performance models we target.

2.2. Pin

Pin [5] is a tool for dynamic, user-defined instrumentation of Itanium®/Linux programs. It provides an API for inserting calls to user-defined analysis procedures at arbitrary points in a program, and allows the instrumentation to observe the architectural state of the processor. Pin provides an infrastructure for writing program analysis tools similar to Atom [13]. We use Pin-based tools to collect basic block profiles for SimPoint, to collect validation data, and to generate instruction traces.

Pin performs the instrumentation at run time. It uses a just-in-time compiler to translate and instrument an application, allowing it to seamlessly handle shared libraries, hand-written assembly, mixed code and data, and some forms of dynamically generated code. It also employs dynamic optimization to make instrumentation efficient. Pin does not require any special compiler or linker switches and can run Itanium®/Linux programs “out-of-the-box” without modification. It can instrument programs using multiple threads and signals. It provides a high degree of transparency — instrumenting a program does not change code or data addresses.

It is easier to dynamically instrument an application with Pin than to statically instrument it and it is faster to run an application under Pin than to run it on a simulator. This has enabled workload collection for more applications than would otherwise be possible. However, there are some limitations; Pin is unable to instrument operating system activity as Pin only operates in user mode. Second, inserting instrumentation slows down the program and can change its time-dependent behavior. For example, the locking behavior of parallel programs might change if the program runs slower. We hope to address some of these issues in future work. For this paper, we have chosen to focus on commercial applications (mostly from the high-performance computing (HPC) domain) and SPEC2000 programs that do not spend much time in the kernel and are not affected by the time-dilation effects of instrumentation.

3. Methodology

Our technique has four steps:

1. Collect a profile with a pin-based tool. 2. Analyze the profile with SimPoint tools to find representative portions (PinPoints). 3. Compare the behavior of the PinPoints to the whole program using Pin and *pfmon* [15]. 4. Generate a trace with a Pin-based tool or use the PinPoints to fast forward to representative portions in execution-based simulation.

3.1. Profile Collection

We profile the branches of an application. For each branch, we count the number of executed instructions since the last taken branch. After executing a fixed number of instructions (*e.g.* 250 million), the profiler outputs the counts for each branch, resets all the counters to zero, emits a marker to identify the point in the execution, and continues execution. The final profiling output is a list of vectors of counts, one vector per slice, in a format suitable for input to the SimPoint tools.

After SimPoint selects the simulation points, other tools such as the trace generator must be able to reliably find the same points in the execution in a later run. We identify a simulation point with a marker, which is an instruction address paired with the number of times it is executed. Pin relies on direct execution on real hardware for speed and simplicity. On the Itanium architecture the data and control speculation mechanisms can cause non-determinism in the program control flow, which can make the marker method unreliable.

With speculation, the compiler can move a load above a branch (control speculation) or a load above a store (data speculation). The Itanium architecture provides hardware mechanisms to detect if this code motion is invalid. Moving a load above a branch is invalid when the load causes an exception it originally did not cause. Moving a load above a store is invalid when the store and load use the same address. When it detects an invalid speculation, the processor branches to compiler-generated recovery code which usually repeats some of the instructions in the correct order and resumes execution. The transitions to recovery code are similar to conditional branches.

Even a correct speculation can sometimes trigger the execution of recovery code. A *translation look-aside buffer* miss can cause a control speculation to fail and a context switch can flush the *data speculation table* (*ALAT*), causing all in-progress data speculations to fail. In addition, Pin sometimes must explicitly flush data speculation table entries. Failed speculations can cause instruction counts and control-flow paths to change from run to run. The change in behavior makes simple instruction count or address/count markers unreliable. To solve this problem, we describe markers that are independent of the speculation recovery flows. Our profiler identifies recovery code dynamically. When the program reaches the end of a slice, the profiler defers emitting a marker until the recovery flow has rejoined the main flow in the application. At that point, we emit an address/count marker. As part of our validation step, we check that PinPoints are reached at the right point in the execution, and have found such checks to be very effective.

3.2. Profile Analysis

We use the SimPoint analysis tool to select representative slices from a profile. We mostly used SimPoint version 1.1 as that was the latest version available during the initial part of this work. However, we did repeat some of the experiments with SimPoint 2.0 when it became available. SimPoint groups the vectors of counts into clusters of similarly behaving slices, and then picks a single slice from the cluster to represent the behavior of the entire cluster. We call the representative slice a *PinPoint.* The number of slices in a cluster divided by the total number of slices determines the weight of the PinPoint.

For multi-threaded programs we simply concatenate the individual thread profiles and present the combined profile to the SimPoint analysis tool. This amounts to treating the multi-threaded run as one large single-threaded run. SimPoint analysis finds representative slices for the concatenated profile regardless of the fact that the representative slices may belong to different threads. The concatenated profile retains the identity of individual threads. PinPoints for multi-threaded programs belong to individual threads and the tracing tool needs to assign individual PinPoints to the right thread. When a PinPoint in a thread is reached, the trace generator may decide which thread(s) to actually trace.

3.3. Validation of PinPoints

After identifying PinPoints, a user of the PinPoints tool needs to know if they accurately represent the performance of the application on the performance model. The most basic performance measure is cycles-per-instruction (CPI), but we would also like the PinPoints to accurately model other behavior like cache misses. For validating PinPoints we collect performance statistics for the PinPoints and for the entire program and compare the whole-program statistics with those predicted by PinPoints, similar to SimPoint and SMARTS. However, SimPoint and SMARTS both use cycle-accurate simulation of the entire benchmark or portions of the benchmark during validation. Our performance models are very detailed and hence slow, making the use of simulation to get whole-program statistics unacceptably time consuming. We want to provide a quick turnaround when looking at a new application or see the effect of a new compiler on an existing application. To speed up the process, we split the validation process into three steps:

1. *Comparison of the performance of the entire application to the performance of PinPoints on real hardware*. We use *pfmon* [15] to sample hardware performance counters and then compute various metrics such as CPI or cache-misses -per-thousand-instructions (MPKI) for the whole program and for the PinPoints. We then use PinPoints statistics and weights to predict the statistics for the entire program. We compare these predictions to the actual values for the entire program. Note that while collecting performance counter data for the PinPoints we still need to run the entire program (at least until the last PinPoint). Hence, there is no cold-start effect on the performance metrics for the PinPoints *in this step*.
2. *Ensuring that the PinPoints are actually reached during trace generation/simulation.*We run a test program at least twice — first for profiling and second for trace generation/ simulation. We assume that the behavior of the program does not change between the two runs. We verify that PinPoints based on the profiling run of the program are actually encountered in the tracing/simulation run.
3. *Comparison of the PinPoints performance on real hardware to the PinPoints performance on a cycle accurate simulator.* Here we simulate the PinPoints traces/regions, get performance metrics and compare them to those obtained in Step 1. Obviously, this step can only be done if there is real hardware corresponding to the performance model; otherwise we do the comparison for as many real hardware platforms (with the same architecture) as possible to gain confidence in the PinPoints. In the results section, we present some data that suggests that cross-platform validation is sufficient. This confirms earlier results from Perelman *et al.* [16] showing that a single set of simulation points can accurately predict behavior across multiple configurations of the same architecture.

3.4. Using PinPoints

PinPoints are designed to drive processor performance models. We support two variants: instruction trace driven and execution driven simulation. Our toolkit provides an instruction trace generator that is implemented as a Pin-based tool, so generating a trace is as easy as collecting a profile. Execution driven simulators use the PinPoint file to decide when to switch between fast forward mode (not in a PinPoint) and detailed modeling mode (in a PinPoint). A PinPoint identifies the beginning and end of a dynamic sequence of instructions. We use an instruction address and a count for the number of times this address is executed to identify a unique point in the execution.

Regardless of the operating mode — trace-driven or execution-driven — the performance model only observes the instructions from the PinPoint, and none between PinPoints. This potentially creates a cold-start effect where structures such as caches and branch predictors are starting empty or untrained, which can cause inaccuracy in simulation. We chose a large slice, 250 million instructions so that cold-start effects should be negligible for all but the really large structures on a processor. The slice is also large enough for the consumer of a PinPoint to use some part of the beginning of a slice for warm-up,

Table 2: Three main machine configurations used for CPI validation

Configuration	***Config1***: IBM Intellistation Z Pro (2P)	***Config2:*** HP rx5670 (4P)	***Config3***: HP rx5670 (4P)
Processor, Frequency, Main Memory	Itanium® Processor, 800 MHz, 10 GB	Itanium® 2 processor, 900 MHz, 16 GB	Itanium® 2 processor, 1.3 GHz, 8GB
L1 D Cache: capacity, line size, associativity, latency	16 KB, 32 bytes, 2 cycles	16 KB, 64 bytes, 1 cycle	16 KB, 64 bytes, 1 cycle
L2 Unified Cache: capacity, line size, associativity, latency	***96 KB***, 64 bytes, 6-way, 6 cycles (int/min) 9 cycles (fp/min)	***256 KB***, 128 bytes, 8-way, 5 cycles (int/min) 6 cycles (fp/min)	256 KB, 128 bytes, 8-way, 5 cycles (int/min) 6 cycles (fp/min)
L3 Unified Cache: capacity, line size, associativity, latency	2 MB (off chip), 64 bytes, 4-way, 21cycles (int/min) 24 cycles (fp/min)	1.5 MB (on chip), 128 bytes, 4-ways per MB, 12 cycles (int/min) 13 cycles (fp/min)	3 MB (on chip), 128 bytes, 4-ways per MB, 12 cycles (int/min) 13 cycles (fp/min)

and the rest for detailed simulation. The decision about how to handle cold-start effects is left to the user of the PinPoints.

3.4.1. Simulating Multi-threaded Programs

For multi-threaded programs PinPoints may be scattered across different threads. Whenever a PinPoint is reached in a thread, our tracing tool starts tracing all threads in order to capture interactions between threads. Thus, multiple trace files are generated for each PinPoint. We have successfully traced four-threaded runs of some SPECOMP2001[10] programs this way.

We assume that all PinPoints will be reached during tracing. For that to happen, the behavior of the program during tracing has to be similar to the behavior during profiling. For multi-threaded programs such determinism is hard to guarantee. In particular, some multi-threaded programs may use dynamic scheduling for load balancing and various threads may do different work on different runs. While processing SPECOMP2001 programs we did find some cases where PinPoints were not encountered during tracing. We are investigating an approach based on the work by Ronsse *et al.* [18] to make multi-threaded programs repeatable.

4. Results and Discussion

We now present results for SPEC2000 programs and some commercial applications we processed. These applications are described in Table 1. During the course of this work we processed SPEC2000 programs built with different compilers/optimizations multiple times — we present results for one collection of SPEC2000 binaries with reference inputs. These binaries were generated with Intel's Electron Compiler (version 8.0 Beta) using a high level of optimization. Some of the machine configurations we used for testing are described in Table 2. The time to process *all* 48 SPEC2000 runs with various PinPoints steps (profile collection, analysis, and CPI validation but not trace generation) is around five days on a 4-processor machine with Config3 (see Table 2). Trace generation time is dominated by the time for on-the-fly compression

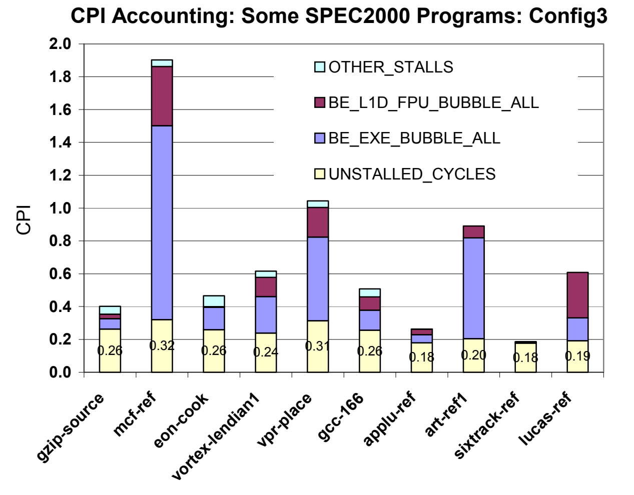

Figure 1 Whole-program CPI breakdown for some SPEC2000 programs.

(using *bzip2*) of the trace and formatting the output and is generally around three hours for a trace of 100 million instructions.

4.1. Program Characterization

We used Itanium-2 performance counters to identify the bottlenecks in the programs we tested. The counters allow us to characterize the program in terms of resource usage as described in the work by Jarp [20]. Such characterization allows us to identify the components of whole-program CPI that are not accurately predicted by PinPoints. We first present results from whole-program "Cycle Accounting" for some of our test programs in Figure 1 and Figure 2. Cycle accounting is a methodology that uses hardware performance counters to identify how cycles are spent during a program execution. We divide the overall (including NOPs) CPI for a given program into the following four sub-components (as shown from bottom to top in the figures): *1. Un-stalled cycles*: Cycles where useful work gets done. *2. BE_EXE_BUBBLE_ALL*: Stall cycles in the execution stage of the Itanium-2 pipeline when execution units do not find required data in registers allocated for the operation being executed. These are due to data access stalls or unsatis-

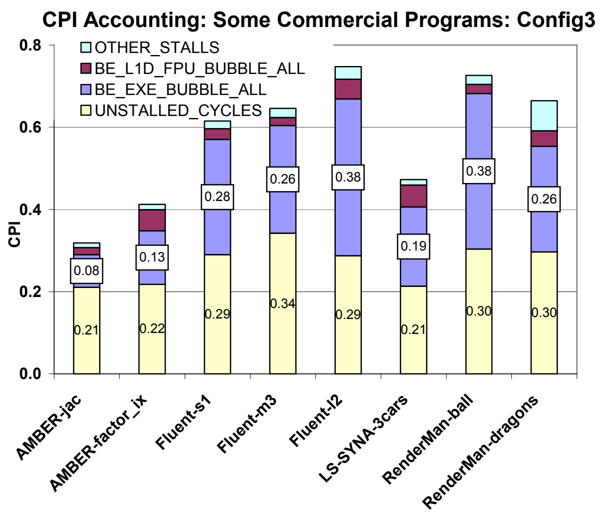

Figure 2 Whole-program CPI breakdown for some commercial programs.

fied register dependencies. *3. BE_L1D_FPU_BUBBLE_ALL*: Stall cycles due to various pipelines (L1 D-cache, floating point unit), and other memory system resources. 4. *Other stalls* (including those due to branch mispredictions).

Itanium-2 can execute up to six instructions per cycle, so the best possible CPI value is 1/6. *sixtrack* from SPECFP2000 actually approaches this ideal CPI value and as Figure 1 shows, most of the cycles in *sixtrack* are un-stalled cycles doing useful work. We note that BE_EXE_BUBBLE_ALL is the dominant reason for stalls in most of the test programs (except *swim*). Further monitoring of sub-events revealed that the major reason for stalls is (not surprisingly) the memory sub-system. Monitoring sub-events of BE_L1D_FPU_BUBBLE_ALL indicates that L1-D-cache related stalls are more prominent than the FPU related stalls. We also notice that SPEC programs show a wide variety in terms of CPI values and sub-component distribution. For commercial programs, we found that memory subsystem is the major reason for stalls (except for AMBER).

4.2. PinPoints Generated

Table 3 lists the PinPoints generated for some of the applications we processed with the PinPoints kit. The second column lists the number of retired instructions (predicated on and off) as counted by our Pin-based profiling tool. The third column shows the total number of slices of size 250 Million instructions each seen during profiling and the last column shows the number of representative slices or PinPoints chosen by SimPoint analysis. For brevity we list the average values for SPECINT and SPECFP. We see that the PinPoints constitute less than one percent of the total number of slices, a drastic reduction in simulation time for the programs. We also note that Fluent-l1 gets the maximum allowed number of PinPoints (10) and possibly has more than 10 phases.

Table 3: PinPoints generated for some programs tested

Program	#Retired Instructions (Billions)	# Slices (250 Million Insts).	#Pinpoints
AMBER-rt	3994	15975	6
Fluent-m3	2625	10499	8
Fluent-l1	3340	13360	10
LS-DYNA-3cars	4932	19729	6
RenderMan®-ball	541	2163	3
SPECINT (Avg.)	142	567	4
SPECFP (Avg.)	373	1491	5

4.3. Validation of PinPoints

As discussed in section Section 3.3, there are three components to validation. We check that (1) the whole-program behavior matches the PinPoint behavior as reported by the performance monitoring counters, (2) we can actually capture the desired PinPoint during tracing/simulation, and (3) the PinPoint behavior on the performance model and on real hardware match. We look at each step individually and discuss the causes of error.

4.3.1. Predicting Whole-program Behavior

Our goal is to determine how closely PinPoints predict whole-program CPI. We compute CPI for PinPoint regions and the whole program with *pfmon* as described in Section 3.3. We compute a predicted CPI for the whole program with PinPoints CPI values and weights. To assess whether one set of PinPoints can be used for multiple configurations of a processor, we repeated the process with the same set of binaries and PinPoints on three different machines using different implementations of the Itanium® architecture. Some of the key differences between the machines we used are listed in Table 2. Config1 uses a processor from the first implementation of the Itanium architecture. Whereas, Config2 and Config3 use processors from the second implementation of the Itanium architecture. Thus, the microarchitecture in Config1 is different from the microarchitecture in Config2 and Config3. The differences between the 2nd and 3rd configurations are the processor frequency, the capacity of L3 Cache, and memory size. We also used a Config4 machine for some experiments. Config4 (*not* in Table 2) is similar to Config3 except for the frequency (1.5 GHz) and the L3 capacity (6 MB). All the machines we used were running Red Hat Advanced Server 2.1 with *pfmon*-enabled variants of Linux kernel 2.4. For RenderMan evaluations, we used machines similar to Config2 and Config3 but running Red Hat Enterprise Linux 3.0.

We first look at actual and predicted CPI values for various test programs on Config3 in Figure 3 and Figure 4. There are two Y-axes — one for CPI values (on the left) and the other for percent difference in CPI values (on the right). The two lines show actual and predicted CPI for the test programs plotted in increasing values of CPI. The bars (hanging upside down

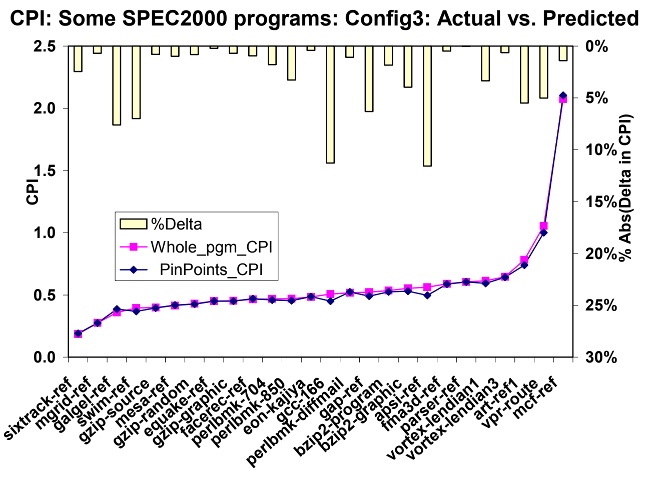

Figure 3 Actual vs. predicted CPI and % delta for SPEC200((Config3)

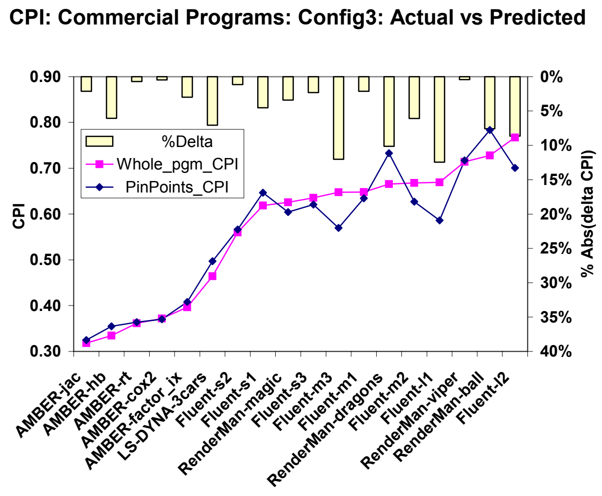

Figure 4 Actual vs. predicted CPI and % delta for commercial programs

in the figures) show percent delta between actual and predicted CPI. We notice that, in general, the predicted CPI values track actual CPI values well and the delta is less than 10% in most cases. Further, the % delta does not seem to be correlated with CPI values indicating that the prediction is working across the board not just for programs with low CPI.

We want to use PinPoints to evaluate performance and microarchitectural changes of *future* Itanium processors. For that, we would first like to know if PinPoints can predict performance differences among *existing* implementations of the Itanium processor. Various machine configurations we used have a variety of processor implementations, cache sizes and parameters, and microarchitectures. That gives us an opportunity to find out how well PinPoints predict performance of the same set of binaries across different configurations. Figure 5 shows results for some SPEC2000 programs. We computed the actual CPI and CPI predicted by the same set of PinPoints on

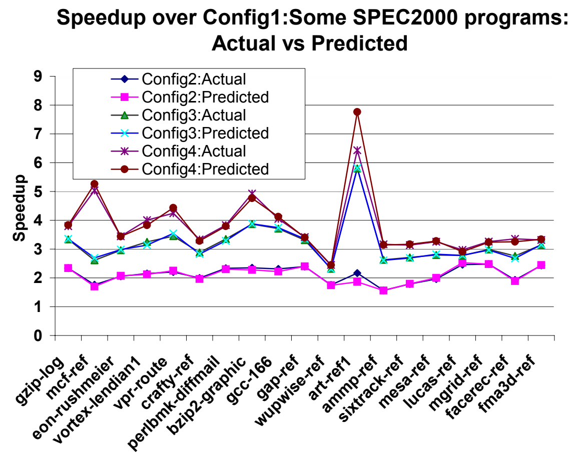

Figure 5 SPEC2000: Speedup prediction using the same set of PinPoints for multiple configurations.

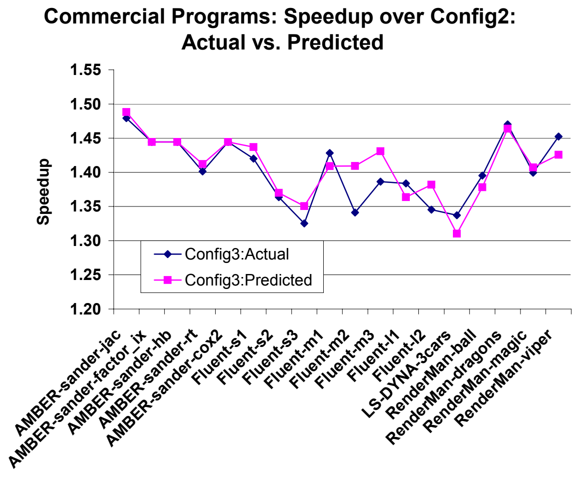

Figure 6 Commercial programs: Speedup prediction using the same set of PinPoints.

four machine configurations (three listed in Table 2). We converted the CPI values to run-time and then computed the speedup on Config2,3,4 over Config1. Figure 5 shows two lines for each configuration — one for actual speedup and the other for PinPoints-predicted speedup. We notice that predicted speedup tracks actual speedup in most cases for all the configurations we tested. That indicates that PinPoints based on branch counts are sufficient to capture whole-program behavior across various implementations of the Itanium architecture for most programs we tested. This is a significant result as it gives designers confidence in using PinPoints to evaluate *future* implementations of the Itanium architecture. We show similar results for 18 commercial applications (speedup over Config2) in Figure 6.

We have used *pfmon* to validate a number of program statistics other than CPI. We now present results from validation of some cache-related statistics. The metric we chose for cache statistics is misses per thousand instructions (MPKI). Figure 7

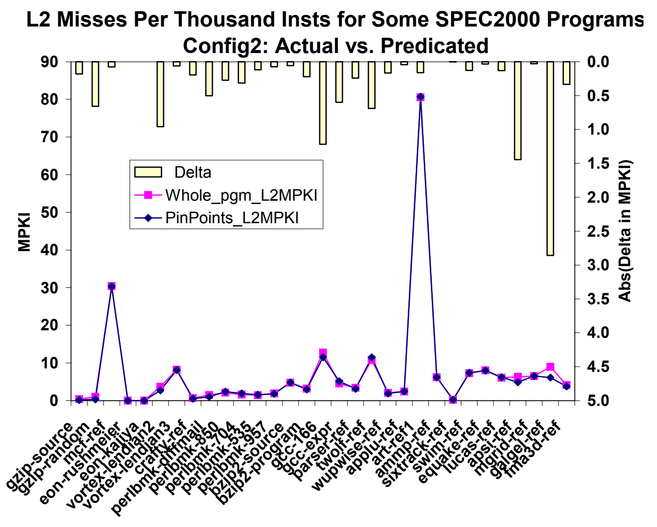

Figure 7 SPEC2000: Actual vs. predicted L2 MPKI and delta in L2 MPKI (Config2).

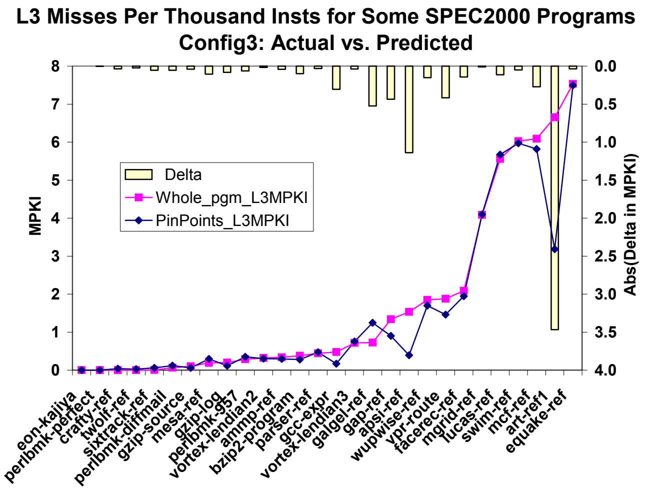

Figure 8 SPEC2000: Actual vs. predicted L3 MPKI and delta in L3 MPKI (Config3).

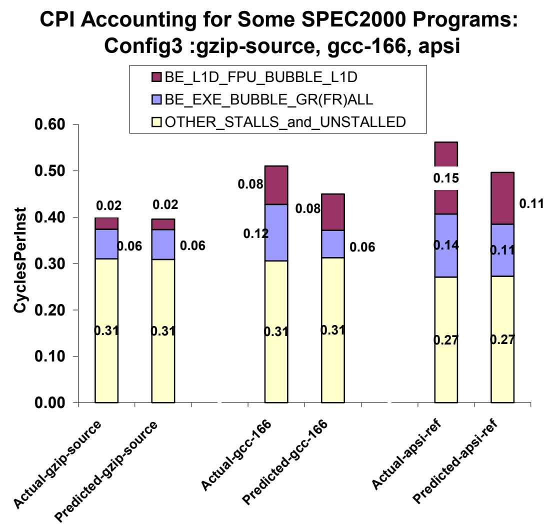

Figure 9 (gzip-source, gcc-166, and apsi) Config3: Contributors to CPI:Actual vs. predicted.

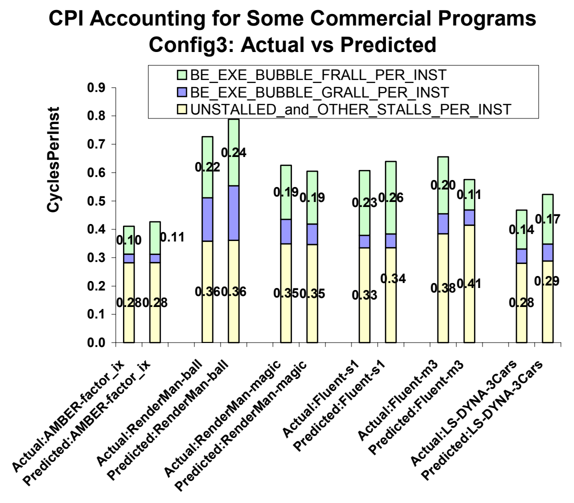

Figure 10 Commercial programs: Config3: Contributors to CPI: Actual vs. predicted.

shows L2-MPKI values for some SPEC2000 programs on Config2. The lines show the actual and predicted L2-MPKI values and the bars (hanging upside down) show *absolute* value of the delta between actual and predicted L2-MPKIs. We show absolute delta values instead of percentages because unlike CPI values, MPKI values can be arbitrarily small, even zero, and a small delta in a small MPKI value shows up as a large percentage distorting its contribution to performance. L2-MPKI prediction does work well across the entire range of L2-MPKI values. However, if we look at a similar plot for L3-MPKI (on Config3) in Figure 8, we see that predicted L3-MPKI deviates from the actual L3-MPKI for a number of programs. The delta for *art-ref1* is unusually large and the large delta in L3-MPKI prediction for *art-ref1* is not reflected in the corresponding delta in CPI prediction (see Figure 3). *art-ref1* is often stalled for memory subsystem as indicated by the relatively large BE_EXE_BUBBLE_ALL component in the CPI for *art-ref1* in Figure 1. Figure 8 shows L3_MISSES which includes all prefetches, instruction misses, L2 write-backs in addition to data misses. Many of these accesses occur in parallel. Further, prefetch and L2-writeback misses do not stall the processor pipeline. Therefore, a large delta in L3 MPKI is not causing a corresponding large delta in CPI for *art-ref1*.

We now take a closer look at some of the cases where percentage delta during CPI validation is relatively high. First, we used *pfmon* to capture Itanium performance counters for "stall analysis" to figure out the bottlenecks for the whole program [20]. Once the major contributors to processor stalls for the entire program were identified, we captured values for those events for the PinPoint regions. Just like CPI validation we then got a "predicted" breakup in stall contributions and compared it with the actual breakup in stall contribution.

For example, when we did the stall analysis of *gcc-166* on machine Config3 we found that the major bottlenecks are related to pipeline and the memory-system resources (event BE_L1D_FPU_BUBBLE_L1D) and to register-register or register-load dependencies (BE_EXE_BUBBLE_GRALL). We then sampled these events for PinPoints and did a comparison of predicted and actual stall contributors. The comparison is illustrated in Figure 9. We see two bars: for each program: one for actual CPI contributors and another for CPI contributors predicted from PinPoint samples. We see that both un-stalled cycles and L1 D-cache related CPI contributions are well predicted by PinPoints. However, register-(register/load) dependency related stalls are not predicted well. Further selective sampling showed that register-load dependency related stalls are not predicted well by PinPoints. We then focused on L1, L2 and L3 cache misses for *gcc-166* and found that the MPKI as predicted by PinPoints was off by 1.5 for L1, 1.4 for L2, and 0.3 for L3. These deltas (using average latency values for L2/L3 and memory accesses) explain why the predicted CPI is off by 0.06 for gcc-166.

Figure 9 also compares CPI break-down for *gzip-source* and *apsi-ref* for Config3. The CPI delta for *gzip-source* on Config3 is very low (0.82%) and Figure 9 shows that the CPI break-down for gzip-source is predicted very well by the PinPoints. For *apsi-ref* both BE_L1D_FPU_BUBBLE_L1D and BE_EXE_BUBBLE_FR_ALL are not predicted well. All other cases with high delta in CPI can be explained in a similar fashion. Figure 10 shows how various components of CPI for some commercial programs are predicted by PinPoints. There are two bars for each program — the first (on the left) is for the actual CPI and the second is for the predicted CPI. Note that the un-stalled portion of the CPI (bottom region, which shows CPI for useful work) is very well predicted across all commercial programs. That is the case even with SPEC2000 programs.

4.3.2. Capturing PinPoints

The previous results determined whether the PinPoints are representative of whole-program behavior. We need to capture the PinPoints as traces. A Pin-based tool called *pingtrace* reads a file that lists the markers for the begin and end of PinPoints, and then runs the program and collects traces. A marker is an address of an instruction and a count of the number of times to execute the instruction before triggering a trace begin or end. After trace collection is complete, we verify that the markers were seen in the proper order at approximately the same instruction counts we had seen during the profiling phase.

Our initial implementation suffered from a marker order problems. This occurred when a marker was placed in the recovery code for speculation. The number of times recovery code is executed can change between the profiling run and the trace collecting run, which means that an address and count marker might never be reached if speculation failures occur less often, or might be reached too soon if speculation failures occur more often during trace generation. Avoiding markers in recovery code eliminates this problem.

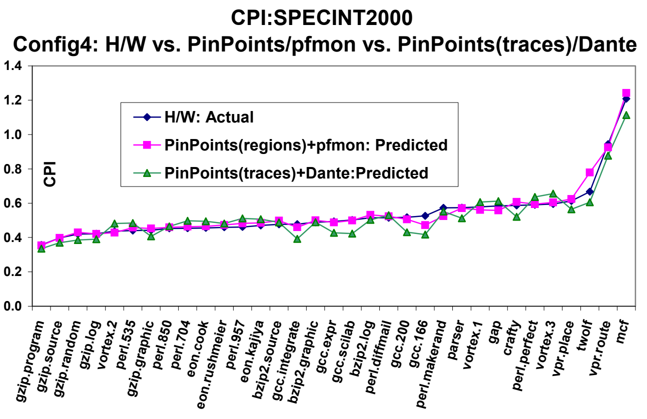

Figure 11 SPECINT2000: Prediction of CPI using PinPoints traces on a Dante/Itanium-2 model.

An additional issue is the skew between the instruction counts in the profile and trace collection phases. *pingtrace* logs actual instruction counts as PinPoints begin and end. Those counts can be contrasted with the intended instruction counts for PinPoint regions in the PinPoints files. Ideally, *pingtrace* should begin tracing PinPoints at the intended beginnings. However, the per-instruction instrumentation required for tracing is rather intrusive and results in more data speculation failures and hence more recovery code execution with *pingtrace* than during profiling. As a result, the actual instruction counts at PinPoint beginnings during tracing is higher than during profiling, and the trace lengths vary from the ideal 250 million.

We looked at the logs from *pingtrace* for SPEC programs and found that the skew between actual and intended beginning dynamic instruction counts of PinPoints is on an average 1.4% for SPECINT and 0.02% for SPECFP. We found a strong correlation between the use of data speculation in the program and the skew. Most SPEC binaries made very little use of data speculation. The *gap* binary has significant use of data speculation and correspondingly the skew between actual PinPoint beginnings and trace lengths for tracing of *gap* is large (30%+). However, this skew does not cause a problem because all the extra instructions are from recovery code, which the simulator can choose to ignore. Since the data speculation related resources are implementation dependent, traces capture data speculation behavior *on the machine used for tracing*. We believe the simulator *should* completely ignore the portions for recovery code in the incoming traces and should implement its own strategy (e.g. assigning a fixed CPI penalty for data speculation) depending on the microarchitecture being simulated.

4.3.3. Matching Performance Model with Hardware

Our final test was to determine whether a performance model and hardware agree on the CPI of the PinPoints. For that, we used a model for a currently shipping Itanium-2 processor (1.5GHz, 6MB L3) based on an Itanium performance simulator called *Dante*.

We simulated the SPEC PinPoints traces on Dante/Itanium-2 and obtained CPIs for the traces. Since the Itanium processor we simulated (Confg4: Itanium-2, 1.5 GHz, 6MB L3) is already available, we could validate our SPEC2000 CPI projections from Dante/Itanium-2 against the actual SPEC2000 CPI from real hardware. The current results from Dante/Itanium-2 CPI projection are presented in Figure 11.

The delta between actual and Dante-predicted SPEC CPIs shown in Figure 11 depends on three factors (1) how closely PinPoints capture whole-program behavior, (2) how accurately the simulator models actual hardware, and (3) cold-start effects in the simulator. Error due to (1) is bounded by the results presented earlier in Section 4.3. The error due to modeling inaccuracies should be corrected. Finally, the cold-start effects are reduced by choosing a large slice size (250 million instructions). The PinPoints toolkit is a valuable resource for identifying modeling inaccuracies. We can collect useful statistics such as cache misses and branch mispredictions on the real hardware for various PinPoints. We then compare those statistics with those from the simulation of corresponding traces. That allows us to narrow down the possible architecture features (cache, branch predictor) that may not be accurately modeled in the simulator. For *mcf* we first observed a large delta between actual and *Dante*-predicted CPI. When we compared statistics for PinPoints from *pfmon* to those from the simulation of corresponding traces we noticed that a large number of prefetches were missing in the simulator. We tracked this down to incorrect effective addresses for data-prefetching instructions. The compiler had inserted prefetch instructions incorrectly requesting semi-random 64-bit addresses. These prefetches caused faults in the Virtual Hash Page Table (VHPT) (used for virtual-to-physical translation of the Itanium architecture on a DTLB miss) and then were aborted. However, Dante does not model the VHPT (which is normally maintained by an OS,) and hence did not cancel the prefetches. This caused thrashing in the cache and hence a difference in Dante-predicted CPI and actual hardware CPI. The solution was to cancel prefetches that missed DTLB translations. We are continuing these efforts to bridge the gap between actual and predicted performance for the SPEC programs.

4.4. Merging PinPoints for Multi-input Programs

Many users of PinPoints traces have expressed concern over long simulation times for multiple traces from multiple inputs for the same SPEC binary (5 for *gcc*, 7 for *perlbmk*…). Hence, we experimented with reducing the number of PinPoints for programs with multiple inputs.

The idea is that some aspects of program behavior will not change with changing input so we should capture PinPoints for a 'merged' run with all the inputs. We profile individual runs as before. We wrote a utility called 'bbmerge' that merges profile vectors from multiple runs into a single profile file (keeping the identity of individual runs intact). We then run the combined file through SimPoint tools increasing the maximum number of allowable PinPoints to 20 instead of the default 10.

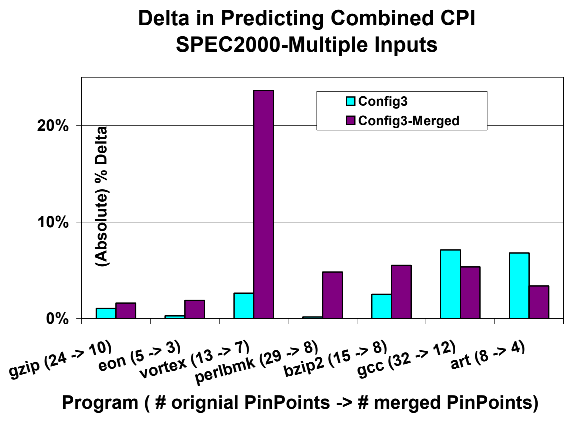

Figure 12 Effectiveness of PinPoints from profiles merged from multiple runs of SPEC2000 programs.

Thus, we get up to 20 'merged' PinPoints. The PinPoint generation script then creates PinPoints files for individual runs. The only difference is that the weights for the PinPoints for a given input no longer sum to 100%, although the weights for all PinPoints across multiple runs do sum to 100%.

To evaluate the effectiveness of merged PinPoints we look at the combined CPI for all the runs using (1) per-input PinPoints and (2) merged PinPoints. In each case we compare predicted and actual CPI. We show the results for some multi-input SPEC programs in on Config3 in Figure 12. There are two bars for each program – the one on the left shows % delta in CPI as predicted by original PinPoints and the one on the right shows the % delta in CPI as predicted by the merged set of PinPoints. On the X-axis we show program name followed by the number of original and merged PinPoints. The total number of PinPoints goes from 126 down to 52. We expect that this reduction in the number of PinPoints will be accompanied by an increase in delta between actual and predicted CPI. The CPI delta is reasonable in most cases. In fact, CPI delta for *gcc* and *art* improved slightly with PinPoint merging. Merged PinPoints for *vortex* show a large % delta indicating that the phase behavior of *vortex* is quite different for different inputs – in particular, we noticed that the CPI for *vortex* with input *lendian2* is markedly lower than the CPI for inputs *lendian1* and *lendian3*. We do not think merged PinPoints are well-suited for projecting absolute SPEC scores. However, they can be used for *relative* performance studies for evaluating new architectural features.

4.5. Comparison with Random Selection of Simulation Points

To assess the effectiveness of SimPoint in selecting slices, we compare it against selecting slices by random sampling. If the results for SimPoint are no better than sampling, then there is no benefit for the extra work it requires. For each program and input, we randomly select the same number and size of

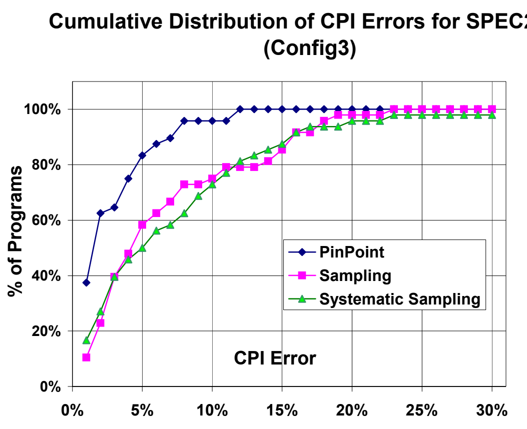

Figure 13 SPEC2000: Comparison of PinPoints with random selection of simulation points.

slices for simulation as SimPoint. The selection is done two ways, random sampling and systematic sampling. For random sampling, every slice can be anywhere in the execution of the program. For systematic sampling, we pick *n* slices by dividing the program execution into *n* intervals and randomly pick one slice from each interval. This method spreads the slices out over the entire execution of the program.

Both the SimPoint and sampling methods are stochastic, so it is not meaningful to compare one selection of slices by SimPoint for a program against one selection of slices by sampling. The sampling methods use random selection of slices, and for a single run you may of may not be lucky in the choice. SimPoint uses a random linear projection to reduce the basic block vector to 15 dimensions and uses random seeds in the *K-means* algorithm. It re-runs with different seeds and takes the best result, so we would expect it to be less sensitive to random effects. However, we have observed that regenerating a profile and rerunning SimPoint can lead to a very different simulation point selection.

Instead of comparing the results of individual programs, we compare a set of experiments with SimPoint against a set of experiments with sampling. For SPEC2000, the set consists of 48 unique runs (unique *program,input* combinations).

For SimPoint and sampling, we compute the cumulative distribution function for the error in CPI. The data for SPEC2000 is presented in Figure 13. The horizontal axis is the error in CPI and the vertical axis is the percentage of programs that have that error or less. For SimPoint on Spec, 95% of the program/input pairs have 8% or less error, but for sampling, 95% of the runs have 18% or less error. This is a clear advantage of SimPoint over random sampling and suggests that the SimPoint algorithm is effective in identifying good simulation points. The standard deviation for SimPoint, random sampling, and systematic sampling are 3%, 9%, and 10% respectively. It does not appear that systematic sampling is significantly better than random sampling.

We performed the same experiment for the 18 runs of the commercial programs and show the data in Figure 14. This is a much smaller set of runs compared to SPEC2000, so it is risky

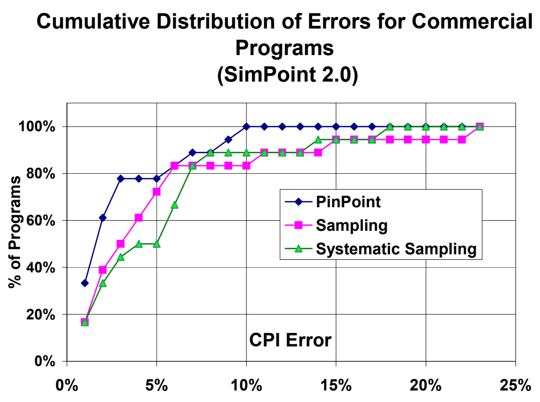

Figure 14 Commercial programs: Comparison of PinPoints with random selection of simulation points.

to draw conclusions because a few uncharacteristic results can distort the graph. SimPoint appears to be better overall than sampling. We note that for SimPoint, 95% of the program/input pairs have a CPI error of 8% or less, while only ~85% of the sampling results have similar error.

5. Related Work

The goal of our work is to engineer a practical system for workload characterization and tracing, therefore we have employed previously-developed technology where possible.

Sherwood *et al.* [3] presented an architecture-independent way of finding representative portions of Alpha programs. They used a static instrumentation tool called Atom [13] to generate samples of execution counts of various basic blocks in Alpha programs. These samples, called basic block vectors, were then fed to cluster finding tools to detect clusters of similar basic block vectors. A representative sample was chosen from each cluster and the corresponding program portion, or slice, was then called a simulation point, to be used for simulation. Each simulation point also got a weight, depending on the relative size of the cluster for it was chosen.

Wunderlich *et al.* described [12] a random sampling technique for SPEC2000 programs called SMARTS. Their analysis showed that they can sample fewer than 30 million instructions to estimate CPI within 3% with 99.7% confidence for 41 SPEC2000 benchmarks. Their implementation showed the actual CPI error to be 0.64% for SPEC2000. This work was built on statistical distribution theory and they applied fixed interval sampling and warm-up techniques. We have commented more on this work in Section 2.1.

Desikan *et al.* [21] did correlation of an Alpha EV6 model against real hardware. They first used some micro-benchmarks where they exhibited high accuracy. Then they used 10 SPEC2000 programs with *test* inputs (called *macro-benchmarks*) and found the mean error in double digits. Our experience with Dante/Itanium-2 shows reasonably low error when compared to real hardware. Further, PinPoints have enabled us to do the correlation for the *reference* runs of *all* SPEC2000 programs – this makes the correlation more realistic.

6. Summary

In this paper we present our experience capturing Itanium/ Linux workloads for microprocessor performance modeling using the PinPoints system. The system employs the SimPoint technique to find representative execution slices. Dynamic instrumentation is used to collect profiles and instruction traces, and to aid the validation of the workload. On-chip hardware counters allow us to measure and isolate any error. The system is evaluated by using it to predict the performance of commercial applications on an industrial performance model, and is measured against several configurations of real hardware. We found that the same set of PinPoints predict whole-program statistics reasonably well (within < 10%) across multiple machine configurations using different implementations of the Itanium processor.

We compared SimPoint-based selection to random selection of simulation points. We found for 95% of the SPEC2000 programs we tested, PinPoints predict actual CPI within 8% of the actual whole-program CPI as opposed to 18% for the random selection. We found similar results for a small set of commercial programs, showing a clear advantage for SimPoint-based selection of simulation points over random selection.

Other interesting lessons are harder to quantify. We think it is incorrect to assume that a workload is collected once and simulated many times or to assume that collection time and difficulty are not important. Often, it is desirable to get a quick prediction of performance for a potential customer or a software vendor, and it must be done at the customer site and not disrupt ongoing work. We believe our dynamic instrumentation based profiling and trace collecting system enables us to collect workloads quicker and more easily than other systems. Previous work focuses on the single step of selecting execution slices, comparing real hardware to performance models. We have learned that some errors are only apparent when putting an entire system together and that it is important to provide a simple methodology to isolate errors.

For future work, in addition to improving accuracy, there are many areas where we would like to extend the capabilities of PinPoints. These include modeling operating system behavior and better support for parallel programs. We are also actively working on porting Pin and PinPoints tools to Intel® IA-32 architecture.

Acknowledgments

We thank the authors (Sherwood *et al.*) of the SimPoint paper for making their analysis tools freely available and especially Prof. Brad Calder for his consultation during this work. Thanks to Michael Greenfield for championing the use of PinPoints for commercial applications. We thank the vendors of AMBER, Fluent, RenderMan, and LS-DYNA for permitting us to cite data from their applications and Ram Ramanujam, Timothy Prince, and Roger Golliver for providing support for those applications. We also thank Allan Knies, Chris Gianos, Geoff Lowney, James Vash, Hideki Saito, Nechama Katan, Youngsoo Choi, Michael Adler, Chris Weaver, and Kim Hazelwood for their help with various aspects of this work.

References

[1] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J.Högberg, F. Larsson, A. Moestedt, B. Werner, “Simics: A Full System Simulation Platform”, IEEE Computer, 35(2):50-58, February 2002.

[2] M. Rosenblum, S. A. Herrod, E. Witchell, and A. Gupta, "Complete Computer Simulation: The SimOS Approach", IEEE Parallel and Distributed Technology, 1995.

[3] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior", 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2002.

[4] T. Conte, M. Hirsch, and K. Menezes, “Reducing State Loss for Effective Trace Sampling of Superscalar Processors”, International Conference on Computer Design (ICCD), October 1996.

[5] PIN home page: http://rogue.colorado.edu/Pin/.

[6] AMBER home page: http://AMBER.scripps.edu/.

[7] Fluent home page http://www.fluent.com/.

[8] LS-DYNA home page http://www.lstc.com/.

[9] RenderMan® home page: https://RenderMan.pixar.com/.

[10] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, M. van Waveren, and B. Whitney, “Large System Performance of SPEC OMP2001 Benchmarks”, the Workshop on OpenMP: Experiences and Implementations, May 2002.

[11] J. Henning, “SPEC CPU2000: Measuring CPU Performance in the New Millennium”, IEEE *Computer*, July 2000.

[12] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling", 30th International Symposium on Computer Architecture (ISCA), June 2003.

[13] A. Srivastava and A. Eustance, "ATOM: A System for Building Customized Program Analysis Tools", Programming Language Design and Implementation (PLDI), 1994, pp. 196-205.

[14] Intel Corporation, “Intel Itanium Architecture Software Developer’s Manual”, Vol. 1: Application Architecture, October 2002.

[15] D. Mosberger and S. Eranian, “IA-64 Linux Kernel Design and Implementation”, chapter 9.3: Kernel Support for Performance Monitoring. Hewlett-Packard Company, 2002. *Pfmon* home page: http://www.hpl.hp.com/research/linux/perfmon/.

[16] E. Perelman, G. Hamerly, and B. Calder, “Picking Statistically Valid and Early Simulation Points”, Parallel Architectures and Compilation Techniques (PACT), September 2003.

[17] Intel Corporation, “Intel Itanium-2 Processor Reference Manual for Software Development and Optimization”, May 2002.

[18] M. Ronsse and K. Bosschere, “RecPlay: A Fully Integrated Practical Record/Replay System”, ACM Transactions on Computer Systems. Vol. 17 (2). 1999. pp. 133-152

[19] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace, “Asim: A Performance Model Framework,” IEEE Computer, 35(2):68-76, Feb. 2002.

[20] S. Jarp, “A Methodology for using the Itanium-2 Performance Counters for Bottleneck Analysis”, HP Labs, August 2002, http://www.gelato.org/pdf/Performance_counters_final.pdf.

[21] R. Desikan, D. Burger, and S. Keckler. "Measuring Experimental Error in Microprocessor Simulation”, 28th International Symposium on Computer Architecture (ISCA), pp. 266-277, July, 2001.