Using Dynamic Binary Instrumentation to Generate Multi-platform SimPoints: Methodology and Accuracy

Vincent M. Weaver and Sally A. McKee

School of Electrical and Computer Engineering
Cornell University
{vince,sam}@csl.cornell.edu

Abstract. Modern benchmark suites (e.g., SPEC CPU2006) take months to simulate. Researchers and practitioners thus use partial simulation techniques for efficiency, and hope to avoid sacrificing accuracy. SimPoint is a popular method of choosing representative parts that approximate an application's entire behavior. The approach breaks an application into intervals, generates a Basic Block Vector (BBV) to represent instructions executed in each interval, clusters the BBVs according to similarity, and chooses a representative interval from the most important clusters. Unfortunately, tools to generate BBVs efficiently have heretofore been widely unavailable for many architectures, especially embedded ones.

We develop plugins for both the Qemu and Valgrind dynamic binary instrumentation (DBI) tools, and compare results to those generated by the PinPoints utility. All three methods can deliver under 6% average CPI error on both the SPEC CPU2000 and CPU2006 benchmarks while running under 0.4% of the total applications. Our tools increase the number of architectures for which BBVs can be generated efficiently and easily; they enable simulation points that include operating system activity; and they allow cross-platform collection of BBV information (e.g., generating MIPS SimPoints on IA32). We validate our tools via hardware performance counters on nine 32-bit Intel Linux platforms.

1 Introduction

Cycle-accurate simulators are slow. Using one to run a modern benchmark suite such as SPEC CPU2006 [16] can take months to complete when full reference inputs are used. This prohibitive slowdown prevents most modelers from using the full reference inputs. Yi et al. [18] investigate the six most common ways of speeding up simulations:

- Representative sampling (SimPoint [13]),
- Statistics based sampling (SMARTS [17]),
- Reduced input sets (MinneSPEC [6]),
- Simulating the first X Million instructions,
- Fast-forwarding Y Million instructions and simulating X Million, and
- Fast-forwarding Y Million, performing architectural warmup, then simulating X Million.

P. Stenström et al. (Eds.): HiPEAC 2008, LNCS 4917, pp. 305–319, 2008.

They conclude that SimPoint and SMARTS give the most accurate results. Over 70% of the previous 10 years of HPCA, ISCA, and MICRO papers (ending in 2005) use reduced simulation methods that are less accurate. Most remaining papers use full input sets. Sampling is thus an under-utilized technique that can greatly increase the breadth and accuracy of computer architecture research.

Collecting data needed by SimPoint is difficult and time consuming; we present two tools to more easily generate the Basic Block Vectors (BBVs) that SimPoint needs. Our tools greatly expand the platforms for which BBVs can be generated, including a number of embedded platforms. We implement the tools using dynamic binary instrumentation (DBI), a technique that generates BBVs much faster than simulation. DBI tools are easier to use than simulators, removing many barriers to wider SimPoint use. Features inherent in the tools we extend make it possible to collect data that previous tools cannot.This includes creating cross-platform BBV files (e.g., generating MIPS BBVs from MIPS binaries on an IA32 host), as well as collecting BBVs that include operating system information along with normal user-space information.

We validate the generated BBVs and compare them against the PinPoint [10] BBVs generated by the Pin utility. We validate all three methods using hardware performance counters while running the SPEC CPU2000 [15] and SPEC CPU2006 [16] benchmark suites on a variety of 32-bit Intel Linux system. Our website contains source code for our Qemu and Valgrind modifications.

2 Generating Simulation Points

SimPoint exploits phase behavior in programs. Many applications exhibit cyclic behavior: code executing at one point in time behaves similarly to code running at some other point. Entire program behavior can be approximated by modeling only a representative set of intervals (in our case, *simulation points* or SimPoints).

Figures 1, 2, and 3 show examples of program phase behavior at a granularity of 100M instructions; these are captured using hardware performance counters on the CPU2000 benchmarks. Each figure shows two metrics: the top is L1 D-Cache miss rate, and the bottom is cycles per instruction (CPI). Figure 1 shows `twolf`, which exhibits almost completely uniform behavior. For this type of program, one interval is enough to approximate whole-program behavior. Figure 2 shows the `mcf` benchmark, which has more complex behavior. Periodic behavior is evident: representative intervals from the various phases can be used to approximate total behavior. The last example, Figure 3, shows the extremely complex behavior of `gcc` running the `200.i` input set. Few patterns are apparent; this type of program is difficult to approximate with the SimPoint methodology (smaller phase intervals are needed to recognize patterns, and variable-size phases are possible, but choosing appropriate interval lengths is non-trivial). We run the CPU2000 benchmarks on nine implementations of architectures running the IA32 ISA, finding that phase behavior is consistent across all platforms when using the same binaries, despite large differences in hardware process and design.

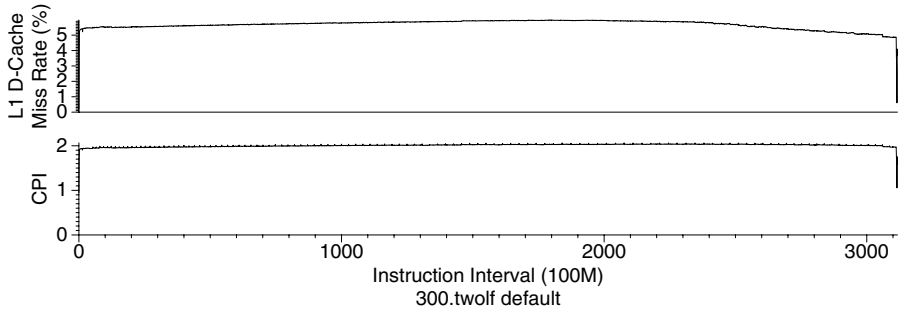

Fig. 1. L1 Data Cache and CPI behavior for `twolf`: behavior is uniform throughout, with one phase representing the entire program

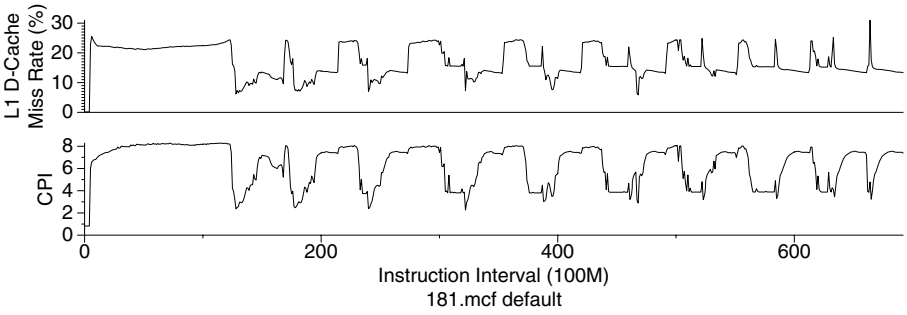

Fig. 2. L1 Data Cache and CPI behavior for `mcf`: several recurring phases are evident

To generate the simulation points for a program, the SimPoint tool needs a Basic Block Vector (BBV) describing the code's execution. Dynamic execution is split into intervals (often fixed size, although that is not strictly necessary). Interval size is measured by number of committed instructions, usually 1M-300M instructions. Smaller sizes enable finer grained phase detection; larger sizes mitigate warmup error when fast-forwarding (without explicit state warmup) in a simulator. We use 100M instruction intervals, which is a common compromise.

During execution, a list is kept of all basic blocks executed, along with a count of how many times each block is executed. The block count is weighted by the number of instructions in each block to ensure that instructions in smaller basic blocks are not given disproportionate significance. When total instruction count reaches the interval size, the basic block list and frequency count are appended to the BBV file. The SimPoint methodology uses the BBV file to find simulation points of interest by K-means clustering. The algorithm selects one representative interval from each phase identified by clustering. Number of phases can be specified directly, or the tool can search within a given range for an appropriate number of phases. The final step in using SimPoint is to gather statistics for all chosen simulation points. For multiple simulation points, the SimPoint tools

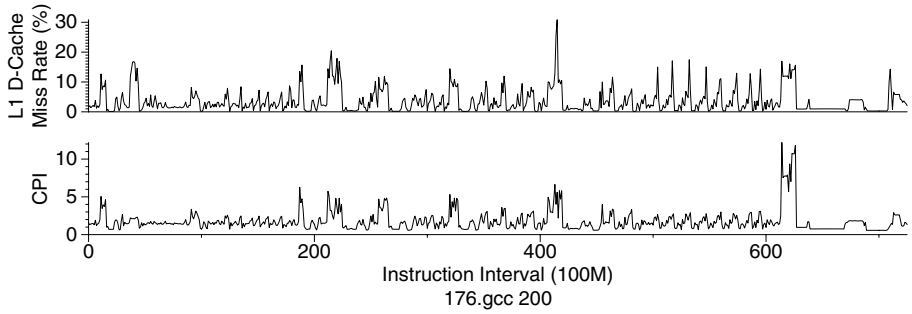

Fig. 3. L1 Data Cache and CPI behavior for `gcc.200`: this program exhibits complex behavior that is hard to capture with phase detection

generate weights to apply to the intervals (and several SimPoints must be modeled for accurate results). By scaling the statistics by the corresponding weights, an accurate approximation of entire program behavior can be estimated quickly (within a small fraction of whole-application simulation time).

2.1 BBV Generation

The BBV file format looks like:

```
T:45:1024 :189:99343
T:11:78573 :15:1353 :56:1
T:18:45 :12:135353 :56:78 314:4324263
```

A `T` signifies the start of an interval, and is followed by a series of colon separated pairs; the first is a unique number specifying a basic block, and the second is the scaled frequency count. There are many methods for gathering information needed to create such BBV files. Requirements are that the tool count the number of committed instructions and track entries into every basic block. The SimPoint website only provides BBV generation tools using ATOM [14] and SimpleScalar [1] sim-alpha. These are useful for experiments involving the Alpha processor, but that architecture has declined in significance. There remains a need for tools to generate BBVs on a wider range of platforms.

Our first attempt used DynInst [3], which supports many platforms, operating systems, and architectures. Unfortunately, the tool is not designed for generating BBVs, and memory overhead for instrumenting some of the benchmarks exceeds 4GB. Furthermore, the tool works with dynamically linked applications. We hope to use future versions, and work with the DynInst developers to generate BBVs without undue overheads. In contrast, Qemu [2] and Valgrind [9] already provide capabilities needed with acceptable overhead, and we modify these two DBI tools to generate BBVs. To validate our methods, we compare results to those from the Pin [7] tool. Figure 4 shows architectures supported for each tool; since all run on Intel platforms, we use them as a common reference.

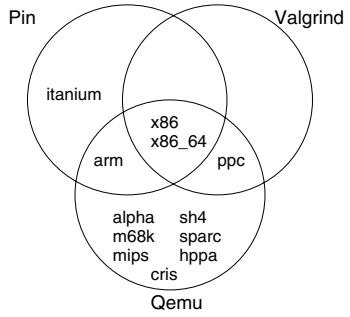

Fig. 4. Architectures supported by Pin, Qemu, and Valgrind: IA32 is the ideal platform for comparison, as it is supported by all of three tools

2.2 Pin

Pin [7] is a dynamic binary instrumentation tool that runs on Intel architectures (including IA32, Intel 64, Itanium, and Xscale), and it supports Linux, Windows, and Macintosh OSX operating systems. We use the PinPoint [10] BBV generation tool bundled with version pin-2.0-10520-gcc.4.0.0-ia32-linux. Pin analysis routines are written in C++, and the instrumentation happens just-in-time, with the resulting instrumented code cached for performance. The core of Pin is proprietary, so internals must be treated as a black box. PinPoint analyses run from 1.5 (`swim`) to 20 (`vortex`) times slower than the binary run on native hardware.

2.3 Qemu

Qemu [2] is a portable dynamic translator. It is commonly used to run a full operating system under hardware emulation, but it also has a Linux user-space emulator that runs stand-alone Linux binaries using system-call translation. Qemu supports the Alpha, SPARC, PowerPC, sh4, IA32, AMD64, MIPS, m68k, and ARM architectures. The user-mode translation we use is currently supported on Linux. Ongoing work will support more operating systems. Qemu uses `gcc` to compile code corresponding to each intermediate language micro-operation. At translation time, these pre-compiled micro-operations are chained together to create translated basic blocks that are cached.

Qemu is not designed for DBI. Using it for our purposes requires intrusive changes to Qemu source. Our code is a patch applied on top of the Qemu 0.9.0 release. We add a unique identifier field to the internal *TargetBlock* basic block structure, which is set the first time a BB is translated. At translation time, we instrument every instruction to call our BBV tracking routine to update BBV counts and total instruction count. Once the interval size is reached, the BBV file is updated, and all counters are reset. Qemu runs from between 4 (`art`) to 40 (`vortex`) times slower than native execution. This makes it slower than Pin but faster than our Valgrind implementation.

Note that `gcc` uses an extremely large stack. By default Qemu only emulates a 512KB stack, but the `-s` command-line option enables at least 8MB of stack space, which allows all `gcc` benchmarks to run to completion.

2.4 Valgrind

Valgrind [9] is a dynamic binary instrumentation tool for the PowerPC, IA32, and AMD64 architectures. It was originally designed to detect application memory allocation errors, but it has developed into a generic and flexible DBI utility. Valgrind translates native processor code into a RISC-like intermediate code. Instrumentation occurs on this intermediate code, which is then recompiled back to the native instruction set. Translated blocks are cached.

Our BBV generation code is a plugin to Valgrind 3.2.3. By default, Valgrind instruments at a *super-block* level rather than the basic block level. A super-block only has one entrance, but can have multiple exit points. We use the `--vex-guest-chase-thresh=0` option to force Valgrind to use basic blocks, although our experiments show that using super-blocks yields similar results. Valgrind implements just-in-time translation of the program being run. We instrument every instruction to call our BBV generation routine. It would be more efficient to call only the routine once per block, but in order to work around some problems with the "rep" instruction prefix (described later) we must instrument every instruction. When calling our instruction routine, we look up the current basic block in a hash table to find a data structure that holds the relevant statistics. We increment the basic block counter and the total instruction count. If we finish an interval by overflowing the committed instruction count, we update BBV information and clear all counts. Valgrind runs from 5 (`art`) to 114 (`vortex`) times slower than native execution, making it the slowest of the tools we evaluate.

3 Evaluation

To evaluate the BBV generation tools, we use the SPEC CPU2000 [15] and CPU2006 [16] benchmarks with full reference inputs. We compile the benchmarks on SuSE Linux 10.2 with gcc 4.1 and "-O2" optimization (except for `vortex`, which we compile without optimization because it crashes, otherwise). We link binaries statically to avoid library differences on the machines we use to gather data. The choice to use static linking is not due to tool dependencies; all three handle both dynamic and static executables.

We choose IA32 as our test platform because it is widely used and because all three tools support it. We use the Perfmon2 [5] interface to gather hardware performance counter results for the platforms described in Table 1.

The performance counters are set to write out the the relevant statistics every 100M instructions. The data collected are used in conjunction with simulation points and weights generated by SimPoint to calculate estimated CPI. We calculate actual CPI for the benchmarks by using the performance counter data, and use this as a basis for our error calculations. Note that calculated statistics

Table 1. Machines used

machine	processor	memory	L1 I/D	L2/L3 Cache	performance counters used
nestle	400MHz Pentium II	256MB	16KB/16KB	512KB	inst_retired, cpu_clk_unhalted
spruengli	550MHz Pentium III	512MB	16KB/16KB	512KB	inst_retired, cpu_clk_unhalted
itanium	800MHz Itanium	1GB	16KB/16KB	96KB/3MB	ia32_inst_retired, cpu_cycles
chocovic	1.66GHz Core Duo	1GB	32KB/32KB	1MB	instructions_retired, unhalted_core_cycles
milka	1.733MHz Athlon MP	512MB	64KB/64KB	256KB	retired_instructions, cpu_clk_unhalted
gallais	1.8GHz Pentium 4	256MB	12Kμ/16KB	256KB	instr_retired:nbogusntag, global_power_events:running
jennifer	2GHz Athlon64 X2	1GB	64KB/64KB	512KB	retired_instructions, cpu_clk_unhalted
sampaka12	2.8GHz Pentium 4	2GB	12Kμ/16KB	512KB	instr_retired:nbogusntag, global_power_events:running
domori25	3.46GHz Pentium D	4GB	12Kμ/16KB	2MB	instr_retired:nbogusntag, global_power_events:running

are ideal, with full warmup. If we were analyzing via a simulation, the results would likely vary in accuracy depending on how architectural state is warmed up after fast-forwarding between simulation points. We use SimPoint version 3.2, the newest version from the SimPoint website, to generate our simulation points.

3.1 The `Rep` Prefix

When validating against actual hardware, total retired instruction counts closely match Pin results, but Qemu and Valgrind results diverge on certain benchmarks. We find the cause of this problem to be the IA32 `rep` prefix. This prefix appears before string instructions (which typically implement a memory operation followed by a pointer auto-increment). The prefix causes the string instruction to repeat, decrementing the `ecx` register until it reaches zero. A naive implementation of the `rep` prefix treats each *repetition* as a committed instruction. In actual hardware, this instruction is grouped in multiples of 4096, so only every 4096^{th} repetition counts as one committed instruction. The performance counters and Pin both show this behavior. Our Valgrind and Qemu plugins are modified to compensate for this, so that we achieve consistent committed instruction counts across all of the BBV generators and actual hardware.

3.2 The `Art` Benchmark

Under Valgrind, the `art` floating point benchmark finishes with half the number of instructions committed by actual hardware. Valgrind uses 64-bit floating point arithmetic for portability reasons, but by default on Linux IA32, programs use 80-bit floating point operations. The `art` benchmark unwisely uses the "==" C operator to compare two floating point numbers, and due to rounding errors between the 80-bit and 64-bit versions, the 64-bit version can finish early, while still generating the proper reference output.

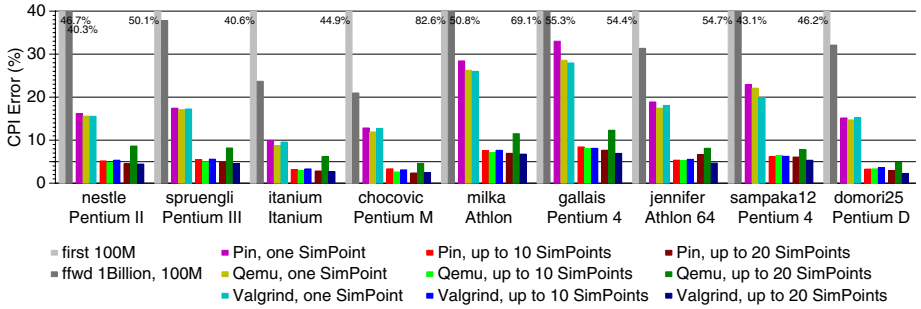

Fig. 5. Average CPI error for CPU2000 when using first, blind fast-forward, and SimPoint selected intervals on various IA32 machines: when using up to 10 simulation points per benchmark, average error is 5.3% for Pin, 5.0% for Qemu, and 5.4% for Valgrind

Having vastly different numbers of completed instructions interferes with simulation point generation, since it limits SimPoint intervals to only part of the complete execution. In order to have the benchmark finish with the same number of instructions, we modify `art` to execute an IA32 assembly instruction to force the FPU to use 64-bit arithmetic. This small change makes the performance counter, Pin, and Valgrind results match. Unfortunately, this does not work for Qemu, which ignores the settings and always uses 80-bit operations.

There are solutions to this problem. One is to use the `-msse2` option of `gcc` to use the 64-bit SSE2 unit instead of the 80-bit x87 floating point unit. Not all of our machines support SSE2, so that workaround is not available. Another option is to use another compiler, such as the Intel C Compiler, which has specific compiler options to enable 64-bit floating point. This does not work with Qemu, which uses 80-bit operations regardless. Therefore we modify the benchmark, and let Qemu generate skewed results.

4 Results

Figure 5 shows results for the SPEC CPU2000 benchmarks. When allowing SimPoint to choose up to 10 simulation points per benchmark, the average error across all machines for CPI is 5.32% for Pin, 5.04% for Qemu, and 5.38% for Valgrind. Pin chooses 354 SimPoints, Qemu 363, and Valgrind 346; this represents only 0.4% of the total execution length, making the simulations finish 250 times faster than if run to completion. It is reassuring that all three BBV methods pick a similar number of intervals, and in many cases they pick the same intervals.

Figure 5 also shows results when SimPoint is allowed to pick up to 20 simulation points. The results are better: error is 4.96% for Pin, 8.00% for Qemu, and 4.45% for Valgrind. This requires less than twice as much simulation — around 0.7% of the total execution length. The increase in error for Qemu is due to poor SimPoint choices in the `gcc` benchmark with the `166.i` input: on many of the architectures, chosen intervals give over 100% error.

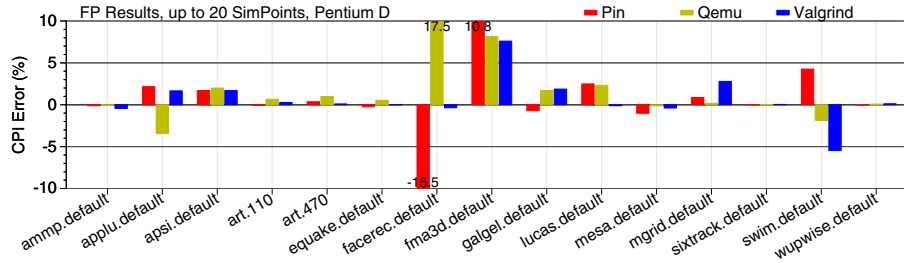

Fig. 6. Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2000 FP: the error with `facerec` and `fma3d` is due to extreme swings in the phase behavior that SimPoint has trouble capturing

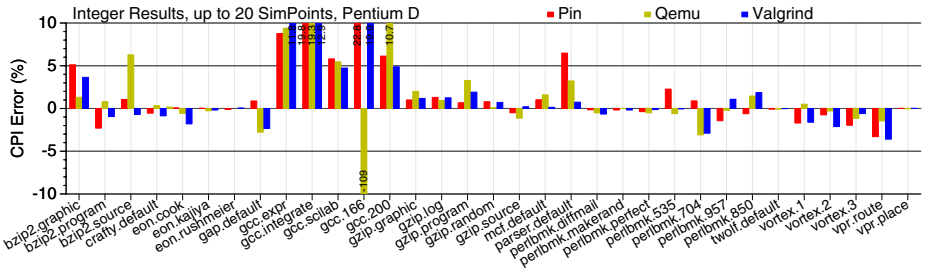

Fig. 7. Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2000 INT: the large error with the `gcc` benchmarks is due to spikes in the phase behavior that SimPoint does not capture well

In addition to the degree of error when using multiple simulation points, Figure 5 shows error for other common methods of interval selection. The first column shows error when running only the first 100M instructions from each benchmark. This method of picking points is poor: error averages around 54% for CPI. Another common method is fast-forwarding 1B instructions and then simulating an interval beginning there (this is equivalent to always choosing the 10th interval as a single SimPoint). This produces better results than using the first interval, but at 37%, the error is still large. Using SimPoint analysis but only choosing one representative interval is a way to use the same amount of simulation time as the previous two methods, but attempts to make a more intelligent choice of which interval to run. As the graph shows, this behaves much better than the blind methods, but the error is twice as large as that from using up to 10 SimPoints.

Figures 6 and 7 show the CPI error for the individual benchmarks on the Pentium D system. For floating point applications, `facerec` and `fma3d` have significantly more error than the others. This is because those programs feature phases which exhibit extreme shifts in CPI from interval to interval, a behavior

often has trouble capturing. The integer benchmarks have the biggest source of error, which is the `gcc` benchmarks. The reason `gcc` behaves so poorly is that there are intervals during its execution where the CPI and other metrics spike. These huge spikes do not repeat, and only happen for one interval; because of this, SimPoint does not weight them as being important, and they therefore are omitted from the chosen simulation points. These high peaks are what cause the actual average results to be much higher than what is predicted by SimPoint. It might be possible to work around this problem by choosing a smaller interval size, which would break the problematic intervals into multiple smaller ones that would be more easily seen by SimPoint.

We also use our BBV tools on the SPEC CPU2006 benchmarks. These runs use the same tools as for CPU2000, without any modifications. These tools yield good results without requiring any special knowledge of the newer benchmarks. We do not have results for the `zeusmp` benchmark: it would not run under any of the DBI tools. Unlike the CPU2000 results, we only have performance counter data from four of the machines. Many of the CPU2006 benchmarks have working sets of over 1GB, and many of our machines have less RAM than that. On those machines the benchmarks take months to run, with the operating system paging constantly to disk. The CPU2006 results shown in Figure 8 are as favorable as the CPU2000 results. When allowing SimPoint to choose up to 10 simulation points per benchmark, the average error for CPI is 5.58% for Pin, 5.30% for Qemu and 5.28% for Valgrind. Pin chooses 420 simulation points, Qemu 433, and Valgrind. This would require simulating only 0.056% of the total benchmark suite. This is an impressive speedup, considering the long running time of these benchmarks. Figure 8 also shows the results when SimPoint is allowed to pick up to 20 simulation points, which requires simulating only 0.1% of the total benchmarks. Average error for CPI is 3.39% for Pin, 4.04% for Qemu, and 3.68% for Valgrind.

Error when simulating the first 100M instructions averages 102%, showing that this continues to be a poor way to choose simulation intervals. Fast-forwarding 1B instructions and then simulating 100M produces an average error of 31%. Using only a single simulation point again has error over twice that of using up to 10 SimPoints. Figures 9 and 10 show CPI errors for individual benchmarks on the Pentium D machine. For floating point applications, there are outlying results for `cactusADM`, `dealII`, and `GemsFDTD`. For these benchmarks, total number of committed instructions measured by the DBI tools differs from that measured with the performance counters. Improving the BBV tools should fix these outliers.

As with the CPU2000 results, the biggest source of error is from `gcc` in the integer benchmarks. The reasons are the same as described previously: SimPoint cannot handle the spikes in the phase behavior. The `bzip2` benchmarks in CPU2006 exhibit the same problem that `gcc` has. Inputs used in CPU2006 have spiky behavior that the CPU2000 inputs do not. The other outliers, `perlbench` and `astar` require further investigation.

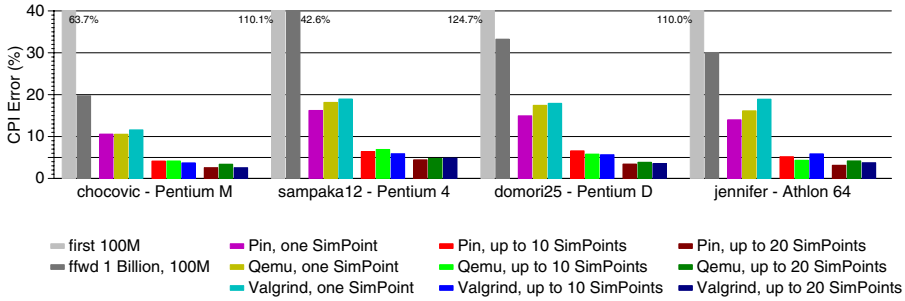

Fig. 8. Average CPI error for CPU2006 on a selection of IA32 machines when using first, blind fast-forward, and SimPoint selected intervals: when using up to 10 simulation points per benchmark, average error is 5.6% for Pin, 5.30% for Qemu, and 5.3% for Valgrind

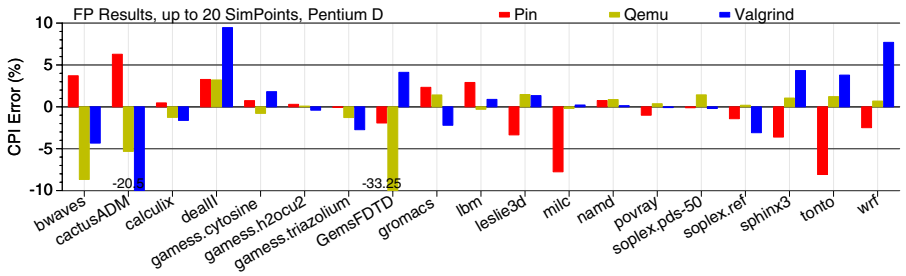

Fig. 9. Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU 2006 FP: the large variation in results for *cactusADM*, *dealII* and *GemsFDRD* are due to unresolved inaccuracies in the way the tools count instructions

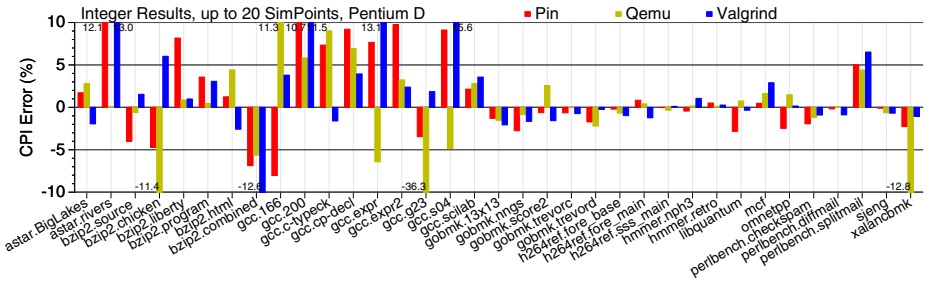

Fig. 10. Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU 2006 INT: the large error with the `gcc` and `bzip2` benchmarks is due to spikes in the phase behavior not captured by SimPoint

5 Related Work

Sherwood, Perelman, and Calder [12] introduce the use of basic block distribution to investigate phase behavior. They use SimpleScalar [4] to generate the BBVs, as well as to evaluate the results for the Alpha architecture. They show preliminary results for three of the SPEC95 benchmarks and three of the SPEC CPU2000 benchmarks. They build on this work and introduce the original SimPoint tool [13]. They use ATOM [14] to collect the BBVs and SimpleScalar to evaluate the results for the SPEC CPU2000 benchmark suite. They use an interval of 10M instructions, and find an average 18% IPC error for using one simulation point for each benchmark, and 3% IPC error using between 6 to 10 simulation points. These results roughly match ours. The benchmarks that require the most simulation points are *ammp* and *bzip2*, which is different from the *gcc* bottleneck we find on the IA32 architecture. This is most likely due to the different ISAs, as well as differences in the memory hierarchy.

Perelman, Hamerly and Calder [11] investigate finding "early" simulation points that can minimize fast-forwarding in the simulator. This paper does not investigate early points because that functionality is not available in current versions of the SimPoint utility. When they look at a configuration similar to ours, with 43 of the SPEC2000 reference input combinations, 100M instruction intervals, and up to 10 simulations per benchmark, they find an average CPI error of 2.6%. This is better than our results, but again this was done on the Alpha architecture, which apparently lacks the *gcc* benchmark problems that appear on the IA32 architectures. They collect BBVs and evaluate results with SimpleScalar, showing that the results on one architectural configuration track the results on other configurations while using the same simulation points. We also find this to be true, but in our case we compare the results from various real hardware platforms.

While many people use SimPoint in their research, often no mention is made of how the BBV files are collected. If not specified, it is usually assumed that the original method described by Sherwood et al. [13] is used, which involves ATOM [14] or SimpleScalar [4]. Alternatively, the SimPoint website has a known set of simulation points provided for pre-compiled Alpha SPEC CPU2000 binaries, so that recalculating using SimPoint is not necessary. Other work sometimes mentions BBV generation briefly, with no indication of any validation. For example, Nagpurkar and Krintz [8] implement BBV collection in a modified Java Virtual Machine in order to analyze Java phase behavior, but do not specify the accuracy of the resulting phase detection.

Patil et al.'s work on PinPoints [10] is most similar to ours. They use the Pin [7] tool to gather BBVs, and then validate the results on the Itanium architecture using performance counters. This work predates the existence of Pin for IA32, so no IA32 results are shown. Their results show that 95% of the SPEC CPU2000 benchmarks have under 8% CPI error when using up to ten 250M instruction intervals. All their benchmarks complete with under 12% error, which is more accurate than our results. One reason for this is that they use much longer simulation points, so they are simulating more of each benchmark. They also investigate commercial

benchmarks, and find that the results are not as accurate as the SPEC results. These Itanium results, as in other previous studies, do not suffer from the huge errors we find in the *gcc* benchmarks. This is probably due to the vastly different architectures and memory hierarchies. Even for the minimally configured machine they use, the cache is much larger than on most of our test machines. The benefit of our study is that we investigate three different methods of BBV generation, whereas they only look at Itanium results generated with Pin.

6 Conclusions and Future Work

We have develop two new BBV generation tools and show that they deliver similar performance to that of existing BBV generation methods. Our Valgrind and Qemu code can provide an average of under 6% CPI error while only running 0.4% of the total SPEC CPU2000 suite on full reference inputs. This is similar to results from the existing PinPoints tool. Our code generates under 6% CPI error when running under 0.06% of SPEC CPU2006 (excepting `zeusmp`) with full reference inputs. The CPU2006 results are obtained without any special tuning for those benchmarks, which indicates that these methods should be adaptable to other benchmark workloads.

We show that our results are better than those obtained with other common sampling methods, such as simulating the beginning of a program, simulating after fast-forwarding 1B instructions, or only simulating one simulation point. All of our results are validated with performance counters on a range of IA32 Linux systems. In addition, our work vastly increases the number of architectures for which efficient BBV generation is now available. With Valgrind, we can generate PowerPC BBVs. Qemu makes it possible to generate BBVs for m68k, MIPS, sh4, CRIS, SPARC, and HPPA architectures. This means that many embedded platforms can now make use of SimPoint methodologies.

The potential benefits of Qemu should be further explored, since it can simulate entire operating systems. This enables collection of BBVs that include full-system effects, not just user-space activity. Furthermore, Qemu enables simulation of binaries from one architecture directly on top of another. This allows gathering BBVs for architectures where actual hardware is not available or is excessively slow, and for experimental ISAs that do not exist yet in hardware.

Valgrind has explicit support for profiling MPI applications. It would be interesting to investigate whether this can be extended to generate BBVs for parallel programs, and to attempt to use SimPoint to speed up parallel workload design studies. Note that we would have to omit synchronization activity from the BBVs in order to capture true phase behavior.

The poor results for `gcc` indicate that some benchmarks lack sufficient phase behavior for SimPoint to generate useful simulation points. It might be necessary to simulate these particular benchmarks fully in order to obtain sufficiently accurate results, or to decrease the interval size. Determining why the poor results only occur on IA32, and do not occur on Alpha and Itanium architectures, requires further investigation.

Overall, these tools show great promise in encouraging use of SimPoint for architectural studies. Our tools make generating simulation points fast and easy, and will help others in generating more accurate results in their experiments.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants 0509406, 0444413, and 0325536.

References

1. Austin, T.: Simplescalar 4.0 release note `http://www.simplescalar.com/`
2. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proc. 2005 USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (April 2005)
3. Buck, B., Hollingsworth, J.: An API for runtime code patching. The International Journal of High Performance Computing Applications 14(4), 317–329 (2000)
4. Burger, D., Austin, T.: The simplescalar toolset, version 2.0. Technical Report 1342, University of Wisconsin (June 1997)
5. Eranian, S.: Perfmon2: A flexible performance monitoring interface for Linux. In: Proc. of the 2006 Ottawa Linux Symposium, pp. 269–288 (July 2006)
6. KleinOsowski, A., Lilja, D.: MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. Computer Architecture Letters 1 (June 2002)
7. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (June 2005)
8. Nagpurkar, P., Krintz, C.: Visualization and analysis of phased behavior in Java programs. In: Proc. ACM 3rd international symposium on Principles and practice of programming in Java, pp. 27–33 (2004)
9. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89–100 (June 2007)
10. Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., Karunanidhi, A.: Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In: Proc. IEEE/ACM 37th Annual International Symposium on Microarchitecture, pp. 81–93 (December 2004)
11. Perelman, E., Hamerly, G., Calder, B.: Picking statistically valid and early simulation points. In: Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques, pp. 244–256 (September 2003)
12. Sherwood, T., Perelman, E., Calder, B.: Basic block distribution analysis to find periodic behavior and simulation points in applications. In: Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques, pp. 3–14 (September 2001)
13. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 45–57 (October 2002)

14. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 196–205 (June 1994)
15. Standard Performance Evaluation Corporation. SPEC CPU benchmark suite (2000), http://www.specbench.org/osg/cpu2000/
16. Standard Performance Evaluation Corporation. SPEC CPU benchmark suite (2006), http://www.specbench.org/osg/cpu2006/
17. Wunderlich, R., Wenish, T., Falsafi, B., Hoe, J.: SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In: Proc. 30th IEEE/ACM International Symposium on Computer Architecture, pp. 84–95 (June 2003)
18. Yi, J., Kodakara, S., Sendag, R., Lilja, D., Hawkins, D.: Characterizing and comparing prevailing simulation techniques. In: Proc. 11th IEEE Symposium on High Performance Computer Architecture, pp. 266–277 (February 2005)