Reverse State Reconstruction for Sampled Microarchitectural Simulation

Paul D. Bryan Michael C. Rosier Thomas M. Conte

Center for Efficient, Secure and Reliable Computing (CESR)
North Carolina State University, Raleigh, NC 27695
{pdbryan, mcrosier, conte}@ncsu.edu

Abstract

For simulation, a tradeoff exists between speed and accuracy. The more instructions simulated from the workload, the more accurate the results — but at a higher cost. To reduce processor simulation times, a variety of techniques have been introduced. Statistically sampled simulation is one method that mitigates the cost of simulation while retaining high accuracy. A contiguous group of instructions, called a cluster, is simulated and then a fast type of simulation is used to skip to the next group. As instructions are skipped, non-sampling bias is introduced and must be removed for accurate measurements to be taken.

In this paper, the *Reverse State Reconstruction* warm-up method is introduced. While skipping between clusters, the data necessary for reconstruction are recorded. Later, these data are scanned in reverse order so that processor state can be approximated without functionally applying every skipped instruction. By trading storage for speed, the proposed method introduces the concept of *on-demand* state reconstruction for sampled simulations. Using this technique, the method isolates ineffectual instructions from the skipped instructions without the use of profiling. Compared to SMARTS [19], *Reverse State Reconstruction* achieves a maximum and average speedup ratio of 2.45 and 1.64, respectively, with minimal sacrifice to accuracy (less than 0.3%).

1. Introduction

Contemporary research and development of computer architecture relies heavily on simulation. Whether designing a new system, or characterizing an existing one, simulation provides crucial insights regarding cycle behavior. Although necessary, cycle-accurate simulation of modern processor designs is extremely time consuming. It can take weeks or even months simulating a workload that can be executed in hardware within minutes. Given that simulation is a bottleneck, many researchers have devised methods to reduce the simulation time.

Various techniques exist to reduce the cycle-accurate simulation time. The criteria under which the instructions are selected varies according to the technique, but they all strive to run only a small subset of the workload to save time. Often researchers will execute small, arbitrary pieces of the workloads under inspection. Although dramatically faster than simulating the entire benchmark, the peril of selecting arbitrary pieces is that inferences potentially can be made on code that is unrepresentative of the entire execution. Other techniques, such as SimPoint [17], inform the user which instructions to execute up to some maximum threshold. Another common approach to limit the total number of simulated instructions involves statistical sampling of the workload. Sampled simulation can be accomplished via systematic sampling or random sampling. In systematic sampling, a metric is used to determine the instructions that will be executed. In random sampling, every instruction has the same chance of being selected.

Random sampling is the most accurate sampling technique for large populations. However, it is cost-prohibitive to select individual instructions at random for simulation. Consequently, cluster sampling, which is a less accurate but more cost-effective technique, is often used instead. In cluster sampling, contiguous groups of elements are selected at random intervals from the population. When relating this technique to sampled processor simulation, the elements are instructions and the population is the workload being simulated. Each of these randomly chosen clusters is then simulated in order to estimate any attribute desired by the user. In terms of statistics nomenclature, each of these clusters is called a "sampling unit" and the collection of clusters is called the sample. The larger the sample, the more likely the estimates obtained from that sample will be correct. However, as the sample size increases, so does the simulation time. Conversely, a sample that is too small can lead to inaccurate estimates. Care must be taken to select an appropriate sampling regimen. A sampling regimen simply defines the number of clusters and the size of the clusters for a particular workload.

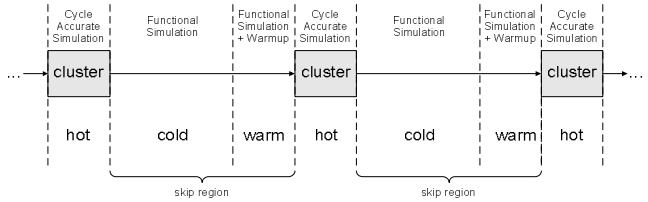

Figure 1: Cluster Sampling

Figure 1 depicts the general composition of a sampled simulation. After the cluster size and number of clusters has been selected, the starting positions of each cluster are randomly generated. Execution consists of three phases: hot, cold, and warm. Hot simulation refers to the complete cycle-accurate simulation of the system. The pipeline, memory

hierarchy, branch predictor, etc., are all simulated within the hot phase. Generally, hot execution consists of normal system simulation. Once the cluster has finished, execution continues in the cold phase. The cold phase consists of simple functional simulation. The purpose of cold simulation is to ensure correct architectural and functional memory state. At some point prior to the next cluster, the warm execution phase begins. In warm execution, data are functionally applied to high-state microarchitectural elements, such as the branch predictor and cache hierarchy. Functional simulation continues as in the cold phase but the elements are not as rigorously modelled as in the hot phase. The purpose of warm execution is to warm-up the state of the processor before measurements are taken from the next cluster.

Many algorithms have been proposed to reconstruct the processor state in warm execution. These algorithms are commonly referred to as warm-up methods. *SMARTS* [19], *MRRL* [7], and *BLRL* [5] are examples of current widely used warm-up methods.

This paper introduces a new warm-up method called *Reverse State Reconstruction*. While skipping between clusters, there are many instructions that do not affect the final processor state immediately prior to the next cluster. If these instructions can be isolated, they can be omitted during warm simulation with no adverse affect on sampling accuracy. The *Reverse State Reconstruction* algorithm reconstructs the state of the cache and branch predictors by iterating through the skip-region trace in reverse order and judiciously applying updates as needed. Unlike other techniques, such as BLRL [5], no analysis is performed between clusters except for logging the needed information for reconstruction. Using this algorithm, a novel concept of on-demand state reconstruction is introduced to achieve 64% speedup compared to *SMARTS,* while still passing confidence interval tests.

2. Related Work

The sampling of workloads has been used in a number of architectural simulation applications. Originally, sampling was applied to cache simulation [2],[6],[10],[20] and was later extended to the simulation of processors [4],[11],[16]. Most instances utilize some derivative of cluster sampling but other forms, such as stratified sampling [13] and set sampling [6],[9],[12], have been used with success. In stratified sampling, the population is classified into groups and elements from each group are chosen to be included in the sample.

Two different types of sampling are possible for caches: time sampling [3],[6],[10],[20] and set sampling [6],[9],[12]. Time sampling involves the extraction of time-contiguous memory references from different locations in an address trace. Set sampling is a form of stratified sampling when applied to caches and involves the inspection of particular cache sets. Thus, the memory references that affect chosen sets are not necessarily temporally adjacent.

Whenever sampling is used, a sample estimate is affected primarily by two kinds of error: sampling bias and non-sampling bias [8]. Sampling bias occurs when sampling units chosen from the population are unrepresentative of the entire population. A properly chosen sampling regimen will have small sampling bias. Technically, non-sampling bias is defined as any bias other than sampling bias. More specifically, non-sampling bias can be described as the difference in state between the sampled simulator and the full simulator at the start of a given cluster. For example, if 20 million instructions were skipped between two clusters, the state of the processor would be much different than if the skipped instructions had been simulated. The difference in state is commonly referred to as the *cold-start problem*. In the previous section, it was stated that warm-up methods are used to warm processor state before the next cluster is executed. These warm-up methods are used to reduce non-sampling bias so accurate measurements can be obtained from the clusters.

Many different approaches have been used to remove non-sampling bias from sampled simulation. Laha, et al., [10] took sampling units immediately following context switches to ensure consistent state. By assuming the cache contents are flushed after a context switch for small caches, the contents are empty and therefore identical to the full execution trace. For larger cache designs, the idea of primed cache sets was introduced by Fu, et al. [6] and Laha, et al. [10]. Once the execution of a new cluster begins, a set in the cache is considered primed after it has been filled with unique references. Only information gathered from primed sets are used to record measurements. The *Reverse State Reconstruction* algorithm for cache warm-up is similar to the notion of a primed set. Before a cache set is simulated in the next cluster, its state must first be reconstructed. Other warm-up techniques proposed by Wood, et al. [20] use probability to distinguish misses at the beginning of a cluster between *compulsory* and *cold-start* misses.

Of all of the warm-up methods, perhaps the most accurate in removing non-sampling bias is SMARTS [19], proposed by Wunderlich, et al. When skipping instructions between clusters, the entire skip region of instructions is executed in a warm phase. Thus, every branch and memory operation is functionally applied to the branch predictor and cache hierarchy. The SMARTS warm-up policy has been applied in cache simulations [2] and to processor simulations [3],[4]. The SMARTS, or full functional, warm-up method is extremely accurate, but at a cost. SMARTS is heavy-handed because many instructions within the skip region do not affect the final state prior to the next cluster. The *Reverse State Reconstruction* algorithm is able to dynamically isolate and remove such ineffectual instructions to realize faster simulation times than SMARTS.

Because SMARTS is demanding in terms of simulation time, other warm-up methods have been proposed that approximate the SMARTS accuracy at a lower cost. Haskins, et al. [7] proposed the *Memory Reference Reuse Latency (MRRL)* algorithm for warm-up. MRRL profiles the skip regions in between clusters (see Figure 1) to determine the number of pre-cluster instructions to execute for a given percentage warm-up. This work was later extended by Eeckhout, et al. [5] with the *Boundary Line Reuse Latency (BLRL)* algorithm. Unlike MRRL, BLRL only considers

memory references from instructions that originate in the cluster. In this study, cluster and pre-cluster pairs are profiled for memory references. Only references in the pre-cluster that affect memory operations in the cluster are applied to the cache. The *Reverse State Reconstruction* algorithm proposed in this paper, unlike the aforementioned techniques, requires no profiling or analysis of skip region instructions. Although effective, the MRRL and BLRL techniques pin down the cluster locations and require profiling analysis whenever the cluster positions are changed.

One widely popular approach used in lieu of statistical sampling was proposed by Sherwood, et al. [17]. This technique, called SimPoint, analyzes the frequency at which basic blocks are executed within a workload. From this heuristic, SimPoint identifies a region or set of regions that can be simulated to approximate the entire program behavior. This technique is hardware independent. While effective, critics of SimPoint note that the heuristic by which the regions are selected utilizes systematic sampling. Since the probability of selection is not random, statistical tests such as the confidence interval cannot be used. An extension to SimPoint, called Variance SimPoint [15], has been introduced to calculate error bounds for sampled clusters. Such error bounds can be calculated if SimPoint selects clusters of execution at random.

3. Reverse State Reconstruction

While skipping between clusters, many instructions do not affect the processor state immediately prior to the next cluster. For example, cache blocks generated by memory operations at the beginning of the skip region will likely be overwritten by those at the end. If such instructions can be identified, they can safely be omitted during warm execution with no adverse affects on sampling accuracy.

The *Reverse State Reconstruction* algorithm reconstructs the state of the cache and branch predictors by iterating through the logged skip-region trace in ***reverse order*** and judiciously applying updates as needed. Once a cluster is finished, cold-phase execution continues to the next cluster boundary. During this phase, certain instructions are logged from the functional simulator. Branch information is saved for reconstruction of the branch predictor, and memory operations are saved for the caches. To minimize the storage requirements of the algorithm, data are kept only for the current cluster of execution. When the current cluster finishes, any saved information is discarded to accommodate data in the next skip region. The contributions of this technique are threefold:

1. The proposed method isolates ineffectual instructions from skip regions between clusters without the use of profiling.
2. The proposed method achieves a maximum and average speedup ratio of 2.45 and 1.64, respectively, over SMARTS with minimal sacrifice to accuracy (less than 0.3%).
3. By trading storage for speed, the proposed method introduces the concept of on-demand state reconstruction for sampled simulations.

The two most important elements to warm-up in sampled processor simulation are the branch predictor and cache hierarchy. The following sections describe how these pieces are reconstructed.

3.1. Cache Reconstruction

Cache reconstruction begins by logging memory operations during cold simulation. During logging, the state of the cache is left *stale*. A stale cache contains the same data that was present after the execution of the previous cluster. The current PC, next PC, the address of the data or instruction, and two Boolean values specifying the entry type (instruction or data) and reference type (load or store) are buffered. Immediately before the next cluster, the reference stream is scanned in reverse order and the cache state is updated. Temporal locality is exploited by applying updates to the cache for only those references that would have affected the final state. References that occurred at the beginning of the skip region, which subsequently are evicted by future references, can be safely removed from warm-up. Redundant references (i.e., references to data already reconstructed in the cache) can also be ignored since their effect on the set has already been processed. Each cache block contains a bit that indicates if it has been reconstructed. These bits are cleared before the logged data are used to warm the cache. Whenever a cache block is reconstructed, its associated reconstructed bit is set.

For each logged reference, a lookup is performed to determine if its corresponding set has been reconstructed. *Remember that since the logged data are processed in reverse order, redundant accesses to a reconstructed set actually occurred earlier and can be ignored*. If the set has been reconstructed, then all subsequent accesses can be ignored. If not, then the reference is classified as present or absent. If present, the LRU bits are updated only if they are stale. If absent, the reference is inserted into the least recently used stale block. After deciding where the block should be placed, the LRU bits of the reconstructed blocks are assigned in ascending order. The first reconstructed block of a set is assigned the most recently used reference. Additional unique references reconstructed into the same set are assigned increasing LRU values. The last reconstructed block of a set is assigned the least recently used reference.

As the logged trace is consumed, sets within the cache are reconstructed. References to reconstructed blocks can be safely ignored because they do not affect the final cache state. During cache reconstruction, updates are applied to both the L1 and L2 caches. For caches with WTNA policies, the block is allocated even if the access is a write to avoid history looking for a previous read.

Studies of this technique show it can sufficiently approximate the cache state without functionally simulating the cache for every reference in the skip region. Despite the buffering of the data reference stream during functional

simulation, it has been shown that reducing the total number of updates to the cache results in faster simulation times.

Figure 2 shows an example of the reverse cache reconstruction algorithm. In this figure, the number above the cache block indicates its LRU value. The letter R indicates that the block has been reconstructed. In this example, a forward reference stream E, A, F, C is applied to a particular cache set. Two columns are used to show how the reverse trace algorithm approximates regular cache simulation for a particular cache set. The left column shows normal cache simulation. As the references are applied on the left, the least recently used element is evicted from the cache, and replaced by the incoming reference. The newly placed reference is the most recently used element, and the LRU bits of the other blocks in the set are updated. After all accesses have been applied, the final state of the cache set is shown.

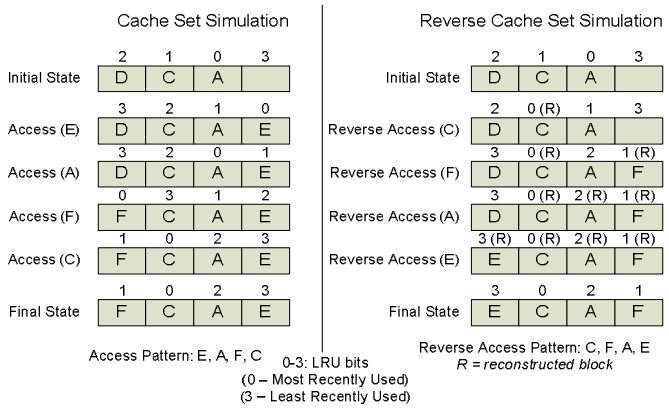

Figure 2: Reverse reconstruction of an individual cache line.

On the right column of Figure 2, the details of the reverse cache reconstruction method are shown. The LRU bits for stale elements are used to determine the way of the inserted block. Reconstructed blocks are placed into the *least recently used stale element*. The set is searched for the maximum LRU value for all reconstructed blocks. If no blocks have been reconstructed, the newly reconstructed block becomes the most recently used. If blocks have been reconstructed, the LRU values will increase. The last reconstructed block becomes the least recently used. As shown in Figure 2, *Reverse Trace Cache Reconstruction* can closely approximate normal cache simulation.

3.2. Branch Predictor Reconstruction

Branch Predictor reconstruction involves state repair in the prediction tables, branch target buffer (BTB), and return address stack (RAS). Branch predictor reconstruction begins by logging branch information during cold simulation. Buffered data includes the current PC, next PC, branch outcome, and other accounting information relevant to determine the final branch effects. This includes the instruction opcode, source register, and any instruction flags. A BTB element in the branch predictor is reconstructed using the address logged during functional simulation. BTB reconstruction is accomplished similar to the cache reconstruction since the BTB can be viewed as a direct mapped cache indicating the taken branch target.

Unlike cache reconstruction, the branch predictor is updated *on-demand* in the next cluster of execution. Specifically, as branches are encountered in the next cluster, the branch predictor is probed to determine if the entry has been reconstructed. If the entry has been reconstructed, then execution in the cluster continues as normal. If not, the entry is first reconstructed before hot execution continues. During the traversal, branches that reference entries that are not relevant to the current entry (i.e., branches that do not index into the same entry) also are reconstructed. By reconstructing other branches in this manner, the logged data does not need to be rescanned from the beginning for each uniquely indexed branch. Because a Gshare predictor is used, the global history register must first be reconstructed using the last *n* branches of the skip-region trace (where *n* is the width of the global history register). Once the global history register has been reconstructed, branch entries can be accurately determined. Like cache reconstruction, the contents of the branch predictor are left stale prior to reconstruction.

Figure 3 shows the normal operation of a 2-bit saturating counter entry indexed within a branch predictor. Each counter value indicates a prediction state. When an instruction is retired, the initial prediction is updated with its outcome. Taken branches cause the counter to increment, and not taken branches cause the counter to decrement. Since the 2-bit counter has a limited number of values, usually only a small amount of history is needed to approximately reconstruct a particular branch predictor entry. In other words, the logged branch history can be used to sufficiently isolate the exact counter value, or narrow the counter value to a set of possible states.

During reconstruction, a series of possible states are tracked for each prediction table entry. Initially, the set of possible states includes all possible counter values: 0, 1, 2, or 3. As references to the same entry are encountered, a reverse branch history is generated. *Remember that the reverse branch history field in the table constitutes that branch history for a particular set in the reverse order*. Therefore, the first outcome in the reverse history was the last outcome for that branch table entry in the skip region. The logged branch history is searched until the counter state for the branch can be determined or until the history has been consumed. Rather than performing this computation at execution time, a table was built *a priori* so that reconstruction can be implemented through a table lookup.

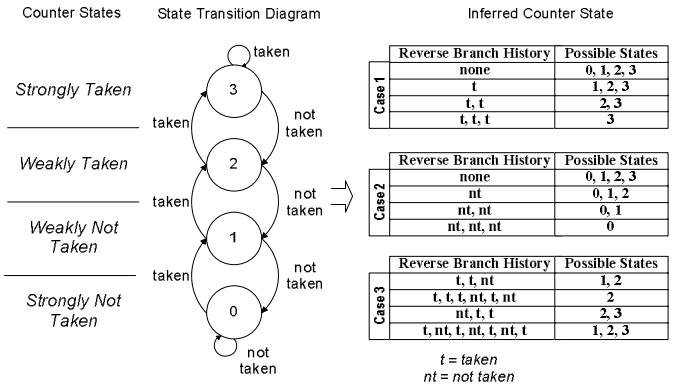

Figure 3: Prediction of Branch Counters

Figure 3 shows several examples of how the reverse branch histories isolated for a particular entry can be used to infer a branch counter or set of branch counters. If the last three consecutive outcomes for a particular branch entry are taken, or not taken, then the exact counter state can be determined. No matter what the original counter state, three taken branches in a row will cause the counter state to become three, and three not taken branches will cause the counter state to become zero (see cases 1 and 2 in Figure 3). Furthermore, if these patterns exist anywhere within the branch history, then the exact counter state can also be determined (see case 3 in Figure 3). However, the branch history does not always yield an exact counter state. Case 3 shows some instances where the exact state cannot be inferred. In this instance, the outcome is predicted based on the remaining set of possible states. If the branch is biased in one direction (taken or not taken) the predictor is set to the weak form. If three states exist, the middle state is predicted. For example, if the remaining possible states include strongly not taken, weakly not taken, and weakly taken, then the state of weakly not taken is predicted. No more than three states can exist for an entry that has a history of one branch. If no history for a branch is produced, then the counter value is left stale.

Reconstruction of a finite size return address stack is accomplished through the following algorithm. Whenever a pop is encountered in the reverse history, a single counter is incremented. If a push is encountered, and the counter is equal to zero, the next PC is placed at the end of the RAS. Otherwise, whenever a push is seen, the counter is decremented. Once the return address stack has been filled, reconstruction is complete. Figure 4 shows an example of a forward and a reverse call sequence. The numbers next to the reverse call sequence indicate the counter value after the push/pop has been processed.

4. Experimental Framework

The model used in this study is an execution-driven simulator based on SimpleScalar [1]. Unlike trace-driven simulation, the processor model fetches instructions from a compiled binary. The front end of the processor can fetch and dispatch eight instructions per cycle, and can issue and retire four instructions per cycle. The model includes eight universal function units that are fully pipelined. The maximum number of in flight instructions is 64. The issue queue size is 32, and there is a load-store queue of 64 elements. The pipeline depth is seven stages. The minimum branch miss-prediction penalty is five cycles. The processor frequency is assumed to be 2 GHz. The branch predictor is a 64K entry Gshare with an eight-entry return address stack. The BTB consists of 4K entries. Architectural checkpoints are utilized to allow the processor to speculatively execute beyond eight branches.

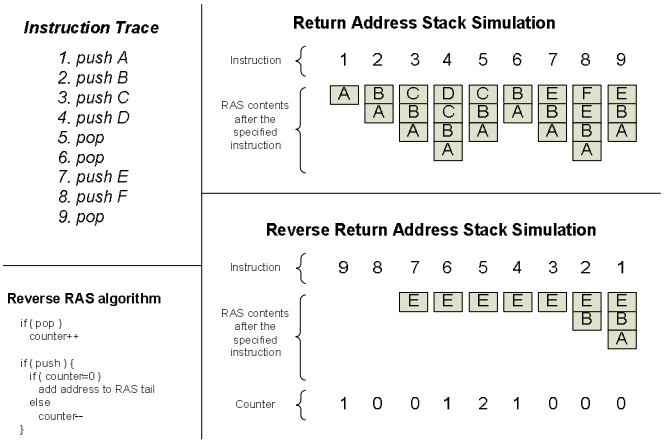

Figure 4: Reverse RAS reconstruction

A substantive memory hierarchy is modelled within the simulator. The first level data cache is 4-way and contains 32 KB with a 64-byte line size. The first level instruction cache is also 4-way and contains 64 KB with a 64-byte line size. The instruction and data caches are implemented using a write-through no-write allocate policy. The second level cache is 8-way and contains 1 MB with a 64-byte line size, and is implemented using a write-back write-allocate policy. A bus model also is incorporated in order to emulate arbitration, contention, and transfer delay between the levels of memory. The first level bus is shared between the first level data and instruction caches, and connects the first level caches to the second level cache. The first level bus has a width of 16 bytes and operates at 1GHz. The second level bus connects the second level cache to main memory, has a width of 32 bytes, and operates at 2 GHz.

The model includes both a functional and a timing simulator. The functional simulator is useful for many reasons. First, the functional simulator is used to validate the results of the timing simulator. If the timing simulator attempts to commit a wrong value, the functional simulator will assert an error. However, in the context of sampled simulation, the functional simulator has additional uses. Second, as instructions in the dynamic stream are skipped (either in cold or warm simulation), the functional simulator retains valid architectural state. When hot execution continues in the next cluster, the values of the registers contained in the functional simulator are copied to the timing simulator.

For processor simulations, the standard performance metric is IPC, which is measured as the number of instructions

retired per execution cycle. The *Reverse Trace Reconstruction* algorithm described above was tested against a number of other techniques for accuracy, speed, and statistical confidence.

5. Experimental Results

Experiments were conducted using the SPEC2000 benchmarks. Integer benchmarks used include *gcc*, *mcf*, *parser*, *perl*, *vortex*, *vpr*, and *twolf*. Floating point benchmarks used include *ammp* and *art*. The first six billion instructions from each benchmark were simulated with reference input sets. Table 1 shows the true IPC of each benchmark simulated during experimentation. The true IPC was used to serve as a baseline for comparison to the various sampling techniques. Sampling regimens were constructed for each workload and are included in the table. All sampling techniques from each compared benchmark utilize the specified sampling regimen. The starting positions of each cluster were then randomly generated according to a uniform distribution. The same starting cluster positions were used for each sampling algorithm (except SimPoint) to keep the sampling bias constant.

benchmark	True IPC	Number clusters	Cluster size
ammp	0.24811	1024	1000
art	0.77980	512	1000
gcc	0.87314	1000	10000
mcf	0.20854	1000	9000
parser	1.07389	2048	5000
perl	1.28956	600	21000
twolf	0.97398	1024	1000
vortex	0.92672	1000	2000
vpr	1.18062	256	6000

Table 1. True IPC and sampling regimen data for each workload

Using this framework, a number of different techniques were compared to measure the effectiveness of non-sampling bias removal. As discussed previously, non-sampling bias is caused by the loss of state information during skipped periods. After a cluster is executed and instructions are skipped, the potential for state loss is high and likely will affect the performance of the next cluster. State in a processor is kept in a number of areas including: the scheduling queues, the reorder buffer, the functional unit pipelines, the branch prediction hardware, instruction caches, data caches, load/store queues, and control transfer instruction queues.

Each warm-up method or policy was then passed through a 95% confidence interval test in order to determine if it correctly predicted the true IPC. The standard deviation S_{IPC} and standard error $S_{\overline{IPC}}$ for a cluster sampling design is given by,

$$S_{IPC} = \sqrt{\frac{\sum_{i=1}^{N_{cluster}} (\mu_{IPC}^{i} - \mu_{IPC}^{sample})^2}{N_{cluster} - 1}}, \qquad S_{\overline{IPC}} = \frac{S_{IPC}}{\sqrt{N_{cluster}}},$$

where μ_{IPC}^{i} is the mean IPC for the i_{th} cluster in the sample. The estimated standard error is used calculate the *error bounds* and *confidence interval*. Using the properties of the normal distribution, the 95% confidence interval is given by $\mu_{IPC}^{sample} \pm 1.96\, S_{IPC}$, where the error bound is $\pm 1.96 S_{IPC}$. A confidence interval of 95% implies that 95 out of 100 sample estimates may be expected to fit into this interval. Moreover, for a well-designed sample, the true mean of the population may also be expected to fall within this range. Low standard errors imply relatively small variation in repeated estimates and consequently result in higher precision. For each warm-up policy, the relative error was calculated as follows:

$$RE(IPC) = \frac{\left| \mu_{IPC}^{true} - \mu_{IPC}^{sample} \right|}{\mu_{IPC}^{true}},$$

where μ_{IPC}^{true} is the true population mean IPC, and μ_{IPC}^{sample} is the IPC estimate obtained from the sample. Relative error relies on μ_{IPC}^{true} from a full-trace simulation of each test benchmark.

Table 2 shows the various warm-up methods used during experimentation. In *no* warm-up, no state repair techniques were used in the skip region. After the execution of a cluster, the caches and branch predictor were left stale. In the *fixed period warm-up* method, a specified percentage of the skip regions immediately prior to the next cluster were used for warm-up. Three variations of SMARTS warm-up were also conducted. The first two consisted of selectively warming only the cache hierarchy or branch predictor. These simulations were used to determine the accuracy of the *Reverse Trace Reconstruction* algorithms when selectively applied to the cache and branch predictor alone. The third variant of SMARTS warmed both the cache and branch predictor for comparison when the reverse trace algorithm also warms the cache and branch predictor. All warm-up methods requiring percentage parameters were conducted using 20, 40, and 80 percent. Finally, a detailed comparison with SimPoint also was performed.

Experiment		Description
None	None	No warm-up was performed during cold execution.
Fixed Period	FP	A specified percentage of the skip region was used for warm execution.
SMARTS cache	$S_{\$}$	All memory operations were functionally applied to the cache in the skip region.
SMARTS branch	S_{BP}	All branches were functionally applied to the cache in the skip region.
SMARTS cache/branch	$S_{\$BP}$	All branches and memory operations were functionally applied in the skip region.
REWIND cache	$R_{\$}$	Reverse State Cache Reconstruction for a specified percentage of the skip region.
REWIND branch	R_{BP}	Reverse State Branch Reconstruction for a specified percentage of the skip region.
REWIND cache/branch	$R_{\$BP}$	The combined Reverse State Reconstruction algorithm.
SimPoint	SimPoint	Multiple simulation point analysis utilizing SimPoint v3.2.

Table 2: Warm-up method experiments

Each of the tested warm-up methods were compared based on accuracy, speed, and statistical confidence. Because the data were too voluminous to compare each individual benchmark, the average performance for each technique was analyzed. Specific workloads will be discussed in greater detail. For the interested reader, all data used to create the graphs are included in the appendix.

Figure 5 shows the relative error and simulation time results for all simulations that selectively warm-up only the cache. As shown, the *Reverse Trace Cache Reconstruction*

algorithm performs closely to SMARTS cache warm-up. The average relative error for SMARTS cache is 3.1%, while the reverse cache warm-up is approximately 3.3%. Although the simulation times for cache warm-up are highly similar, the simulation times vary significantly. Full functional simulation of the cache in the skip region takes an average of 1443 seconds, while the 20% reverse cache warm-up takes 1086 seconds. By applying the last 20% of the memory references to the cache hierarchy a speedup ratio of 1.41 was achieved for cache warm-up. For these simulations, *gcc* had the largest speedup ratio of 1.93 while *parser* had the smallest speedup ratio of 1.03. Therefore, reverse cache reconstruction at 20% always reduced simulation time when compared to SMARTS. As the warm-up percentages increased, the speedup ratio was degraded. At 40 and 80 percent, the speedup ratios are 1.27 and 1.05, respectively. At 40%, most workloads performances were improved, but *mcf* exhibits degradation in simulation speed with a speedup ratio of 0.97. At 80%, all workloads show speedup except *mcf*, *parser*, and *vortex*.

Little additional benefit was obtained by executing more than 20% of the logged cache data. This is consistent with temporal locality, such that the cache blocks at the beginning of the skip-region will be evicted by subsequent references.

Figure 6 shows the relative error and simulation time results for all simulations that selectively warm-up only the branch predictor. As shown, the *Reverse Trace Branch Predictor Reconstruction* algorithm performs similarly to SMARTS. Both the reverse algorithm and SMARTS warm-up achieve an average relative error of 22.3 and 22.2 percent, respectively. However, the average speedup ratio of the reverse technique over SMARTS branch prediction warm-up is 1.48. *Gcc* exhibits the highest speedup ratio of 2.26, while *mcf* has the lowest of 1.10.

As shown in Figures 5 and 6, the cache hierarchy has the greatest impact on non-sampling bias for sampled simulation. Warming the branch predictor alone produced an average relative error of 23% while warming the cache alone produced an average relative error of 3.1%. Although it may seem advantageous to only warm the cache structures in sampled simulation, non-sampling bias produced by cold state in the branch predictor is sufficient to cause many simulations to fail the confidence interval tests (see appendix).

Figure 7 shows the relative error and simulation time results for all simulations that incorporated both the cache and branch predictor in warm-up. No warm-up had the least overhead of all techniques, and thus had the lowest simulation time but produced the highest error at 23%. Of the remaining techniques, SMARTS had the lowest error at 0.9%, but had the highest simulation time. *Reverse Trace Reconstruction* achieved speedup ratios of 1.64, 1.51, and 1.25 for 20, 40, and 80 percent, respectively. At 20 and 40 percent, all workloads executed faster using the proposed algorithm than SMARTS. At 80 percent, *mcf* was the only workload that suffered in simulation time with a speedup ratio of 0.918.

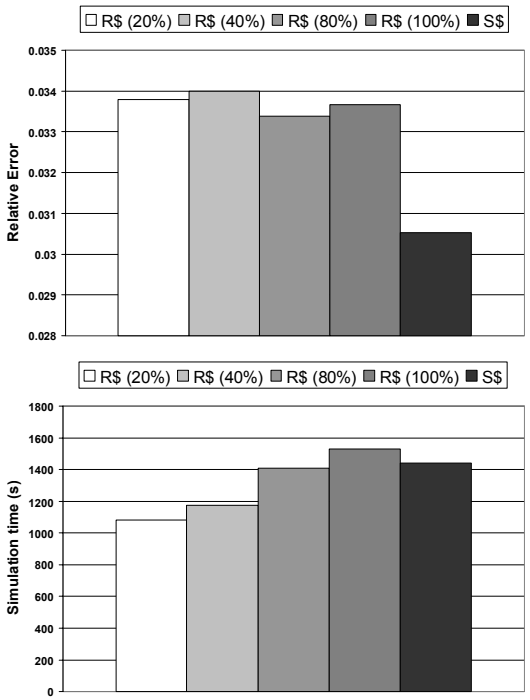

Figure 5: Cache warm-up only

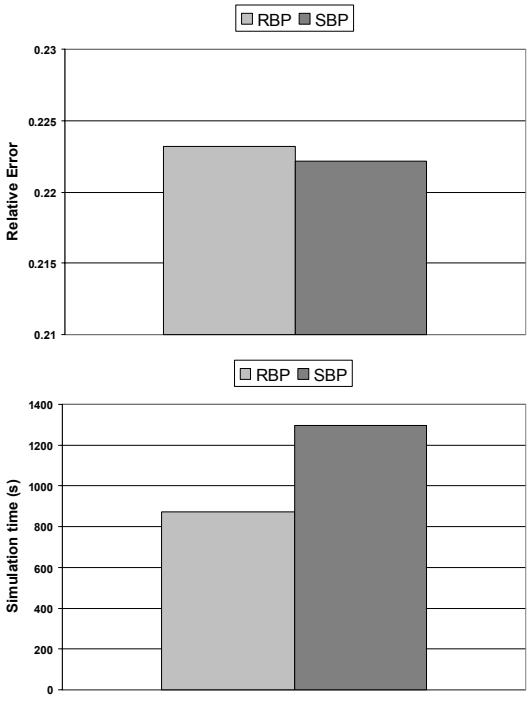

Figure 6: Branch Prediction warm-up only

Fixed period simulations performed highly similar to the proposed methods at the specified percentages. At 20%, fixed period has a lower simulation time. However, as the percentages increase to 40 and 80% the reverse techniques run faster. One explanation is that all accounting information necessary for reconstruction is logged in the skip region, regardless of the warm-up percentage. As the reconstruction

percentages increase, the data buffering cost are amortized over the reconstruction time.

Figure 8 shows the relative error and simulation time results for the *Reverse State Reconstruction* compared to SMARTS warm-up. At 20% warm-up, the average relative error with respect to SMARTS for all simulated workloads is 0.3%. At 20% warm-up the minimum and maximum relative errors with respect to SMARTS are 0.01% and 1.9%, respectively. Since SMARTS is the most accurate, it is expected that SMARTS should have the lowest error. The average behavior for the tested workloads is shown in Figure 7. Figure 8 shows these results by individual benchmark. As expected, the simulation time increases as the specified warm-up percentage increases. Figure 9 shows the average relative error and simulation times for SimPoint with the *Reverse Trace State Reconstruction* at 20%. In order to fairly compare SimPoint with sampled simulation, a variety of different interval sizes were incorporated. All SimPoint comparisons were conducted utilizing multiple simulation points (30), at varying interval sizes. SimPoint v3.2 [17] was used in these experiments.

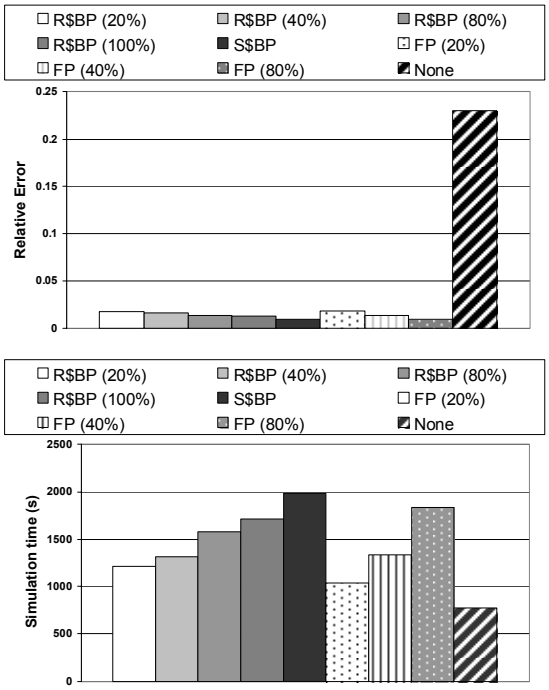

Figure 7: Cache and Branch Prediction warm-up

SimPoint allows the user to specify an interval size that defines the granularity at which basic block vector profiling is conducted. Originally, an interval size of 50K was selected in order to keep the number of instructions in hot execution constant. As shown below, SimPoint produces an average error of 20% when an interval size of 50K is used. One reason for this is that not all SimPoint variants incorporate warm-up while skipping to the next simulation point, or cluster. Without warm-up, measurements taken from small clusters are greatly affected by non-sampling bias. Therefore, SMARTS warm-up was incorporated into the SimPoint simulations. In 50K-SMARTS, the SMARTS warm-up policy was used to warm-up processor state while skipping instructions to the next simulation point indicated by SimPoint. As shown below, the error rate dropped to 8% when warm-up was included.

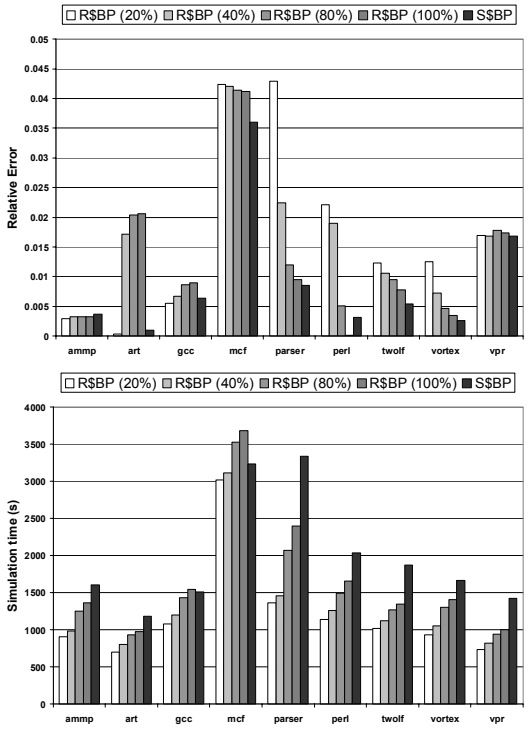

Figure 8: Reverse State Reconstruction vs SMARTS

Although an interval size of 50K was selected for sample size consistency, the authors of SimPoint do not suggest such a small interval size. As a result, the interval size was increased to 10M. Using a 10M-interval size, the relative error of SimPoint was 4.2%. For symmetry, SMARTS warm-up with an interval size of 10M was also performed, and had an average error of 5.9%. With an interval size of 50K the introduction of a warm-up method helped simulation accuracy. However, with an interval size of 10M its accuracy was degraded. No conclusions can be drawn from the addition of warm-up to the SimPoint method.

At the lowest interval size, SimPoint was faster than sampled simulation, but at a higher cost in accuracy. Increasing the interval size increased the accuracy, but at a high simulation cost. The *Reverse Trace Reconstruction* algorithm had an average relative error of 1.7%

All warm-up methods were then tested for statistical confidence (see appendix). Using a 95% confidence interval, the variability of each sample was tested to determine if it could correctly predict the actual IPC. At 20% warm-up, the reverse trace reconstruction correctly predicted the true IPC for seven of the nine workloads. The remaining two also were predicted at 80%.

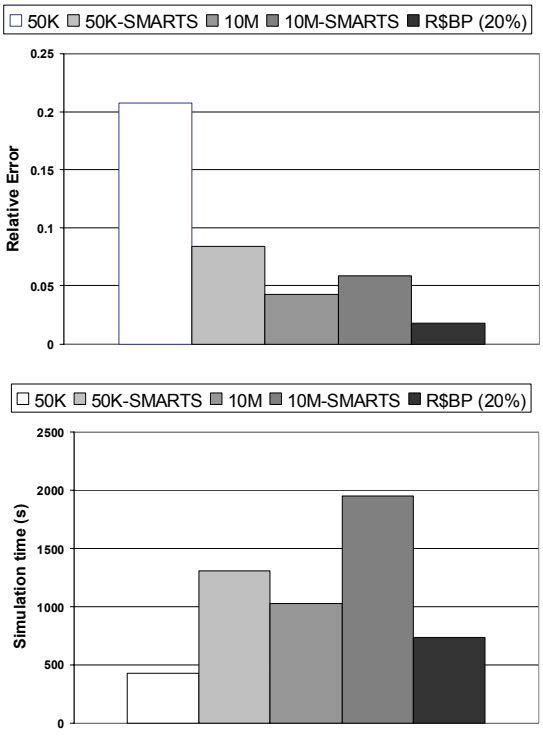

Figure 9: SimPoint comparison

6. Conclusion

In this paper, a new *Reverse State Reconstruction* warm-up method was introduced for sampled simulation. Using this method, considerable speedups were achieved relative to SMARTS, with negligible accuracy loss. Maximum and average speedup ratios of 2.45 and 1.64, respectively, were obtained with an accuracy loss of less than 0.3%. By recording data while skipping instructions, processor state can be reconstructed *on-demand* rather than naively applying every memory addresses and branch instructions functionally. From the experiments conducted in this study, it is shown that ineffectual instructions can be selectively removed from warm-up to reduce simulation time.

7. Acknowledgements

We would like to thank Dr. Suleyman Sair, one of the original contributors to SimPoint, for insights regarding SimPoint evaluation.

8. References

[1] Burger, D. C., and Austin, T. M. *The Simplescalar Toolset, version 2.0*. Computer Architecture News, 25(3):13-25, June 1997.

[2] Conte, T. M., Hirsch, M. A., and Hwu, W. W. *Combining Trace Sampling With Single Pass Methods for Efficient Cache Simulation*. IEEE Transactions on Computers, vol. C-47, no. 6, Jun. 1998.

[3] Conte, T. M. *Systematic computer architecture prototyping*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1992.

[4] Conte, T. M., Hirsch, M. A., and Menezes, K. N. *Reducing State Loss for Effective Trace Sampling of Superscalar Processors*. In Proc of the 1996 International Conference on Computer Design, (Austin, TX), Oct. 1996.

[5] EeckHout, L., Luo, Y., Bosschere, K. D., and John, L. K. *BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation*. The Computer Journal, 2005 Oxford University Press. Vol. 48 (4). 2005. pp. 451-459.

[6] Fu, J. W. C., and Patel, J. H. *Trace driven simulation using sampled traces*. In Proc. 27th Hawaii Int'l. Conf. on System Sciences, (Maui, HI), Jan. 1994.

[7] Haskins, J. W., and Skadron, K. *Memory Reference Reuse Latency: Accelerated Sampled Microarchitecture Simulation*. In Proc of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 195-203, Mar. 2003.

[8] Henry, G. T. *Practical sampling*. Newbury Park, CA: Sage Publications, 1990.

[9] Kessler, R. E., Hill, M. D., and Wood, D. A. *A comparison of trace-sampling techniques for multi-megabyte caches*. IEEE Trans. Comput., vol. C-43, pp. 664–675, June 1994.

[10] Laha, S., Patel, J. A., and Iyer, R. K. *Accurate low-cost methods for performance evaluation of cache memory systems*. IEEE Trans. Comput., vol. C-37, pp. 1325–1336, Feb. 1988.

[11] Lauterbach, G. *Accelerating architectural simulation by parallel execution*. In Proc. 27th Hawaii Int'l. Conf. on System Sciences, (Maui, HI), Jan. 1994.

[12] Lui, L., and Peir, J. *Cache sampling by sets*. IEEE Trans. VLSI Systems, vol. 1, pp. 98-105, June 1993.

[13] Mangione-Smith, W. H., Abraham, S. G., and Davidson, E. S. *Architectural vs Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1*. In Proc. 24th Hawaii International Conference on System Sciences, January 1991.

[14] McCall, J. C. H. *Sampling and statistics handbook for research*. Ames, Iowa: Iowa State University Press, 1982.

[15] Perelman, E., Hamerly G., and Calder, B. *Picking Statistically Valid and Early Simulation Points*. In the International Conference on Parallel Architectures and Compilation Techniques, September 2003.

[16] Poursepanj. *The PowerPC performance modeling methodology*. Communications ACM, vol. 37, pp. 47–55, June 1994.

[17] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. *Automatically Characterizing Large Scale Program Behavior*. In the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002.

[18] Wenisch, T. F, Wunderlich, R. E., Falsafi, B., and Hoe, J. C. *Simulation Sampling with Live-Points*. IEEE International Symposium on Performance Analysis of Systems and Software, Mar. 2006.

[19] Wunderlich, R. E., Wenish, T. F, Falsafi, B., and Hoe, J. C. *SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling*. Proc. 30th ISCA, 2003.

[20] Wood, D. A., Hill, M. D., and Kessler, R. E. *A model for estimating trace-sample miss ratios*. In Proc. ACM SIGMETRICS '91 Conf. on Measurement and Modeling of Comput. Sys., pp. 79–89, May 1991.

Appendix

Relative Error

	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr	AVG
FP (20%)	0.0035	0.02	0.0032	0.0386	0.0402	0.0226	0.0094	0.0114	0.0162	0.0184
FP (40%)	0.0037	0.0037	0.0041	0.0364	0.021	0.0203	0.0082	0.0066	0.0164	0.0134
FP (80%)	0.0038	0.0009	0.006	0.0361	0.0101	0.0011	0.007	0.0039	0.0173	0.0096
None	0.0025	0.1665	0.2001	0.1472	0.4764	0.0897	0.4836	0.1795	0.3267	0.2302
$S_{\$}$	0.0202	0.0125	0.0397	0.0302	0.0625	0.0107	0.0517	0.0126	0.0347	0.0305
S_{BP}	0.0188	0.1574	0.1804	0.1525	0.4554	0.083	0.4648	0.1696	0.3176	0.2222
$S_{\$BP}$	0.0037	0.0009	0.0063	0.036	0.0085	0.0032	0.0054	0.0026	0.0168	0.0093
$R_{\$ (20\%)}$	0.0194	0.0118	0.0383	0.037	0.0919	0.0106	0.058	0.0028	0.0344	0.0338
$R_{\$ (40\%)}$	0.0197	0.0287	0.0392	0.0365	0.074	0.008	0.0574	0.008	0.0345	0.034
$R_{\$ (80\%)}$	0.0197	0.0319	0.041	0.0359	0.0644	0.0052	0.0561	0.0106	0.0357	0.0334
$R_{\$ (100\%)}$	0.0197	0.0321	0.0413	0.0356	0.0623	0.0105	0.0545	0.0118	0.0352	0.0337
R_{BP}	0.0186	0.1635	0.1907	0.1382	0.4597	0.0781	0.4691	0.1716	0.3189	0.2232
$R_{\$BP (20\%)}$	0.0029	0.0003	0.0055	0.0423	0.0429	0.0221	0.0123	0.0125	0.0169	0.0175
$R_{\$BP (40\%)}$	0.0033	0.0171	0.0067	0.0421	0.0224	0.019	0.0105	0.0072	0.0168	0.0161
$R_{\$BP (80\%)}$	0.0033	0.0203	0.0086	0.0414	0.0119	0.005	0.0095	0.0046	0.0178	0.0136
$R_{\$BP (100\%)}$	0.0033	0.0206	0.009	0.0411	0.0095	0.0002	0.0078	0.0035	0.0173	0.0125

Time

	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr	AVG
FP (20%)	759.35	632	1336.1	2331.6	953.45	1004.1	911.33	809.54	653.31	1043.4
FP (40%)	934.81	780.96	1717.6	3046.3	1210	1256.4	1142	1060.5	862.36	1334.5
FP (80%)	1336.7	1030.3	2354.5	4012.7	1785.5	1762.8	1616.1	1436.3	1217.8	1839.2
None	548.4	523.65	913.86	1631.9	700.78	803.7	650.43	637.25	542.16	772.46
$S_{\$}$	1199.8	1016.5	1899.1	2773.4	1292.5	1428.8	1254.4	1188	936.63	1443.2
S_{BP}	945.11	646.44	1806.8	2435	1361.1	1302.3	1234.2	1012.4	926.27	1296.6
$S_{\$BP}$	1603.5	1181.4	1508.8	3235.8	3338.5	2038.3	1874.4	1662.2	1419.7	1984.7
$R_{\$ (20\%)}$	792.56	681.95	979.64	2664.4	1246.8	1064.6	892.36	804.75	643.2	1085.6
$R_{\$ (40\%)}$	896.78	765.98	1205.8	2830.1	1105.5	1115.6	983.88	955.88	731.06	1176.7
$R_{\$ (80\%)}$	1136.9	997.38	1285	3240	1428.9	1342.3	1134.8	1294.9	833.27	1410.4
$R_{\$ (100\%)}$	1244.5	925.12	1493.3	3429.2	1734	1535.9	1229.6	1276.1	919.3	1531.9
R_{BP}	650.84	505.1	800.77	2203.7	867.04	845.74	769.1	653.87	558.01	872.68
$R_{\$BP (20\%)}$	905.82	697.58	1076.4	3023.1	1360.5	1141.6	1018.4	930.69	735.63	1210
$R_{\$BP (40\%)}$	984.67	807.6	1196.6	3116.9	1461.6	1263.3	1122.1	1049.9	817.94	1313.4
$R_{\$BP (80\%)}$	1251	928.69	1428.6	3523.9	2068.9	1496.9	1270.5	1305.5	944.81	1579.9
$R_{\$BP (100\%)}$	1368.4	976.37	1544.5	3683.7	2396.4	1656	1346.2	1410.5	1003.8	1709.5

Confidence tests

	ammp	art	gcc	mcf	parser	perl	twolf	vortex	vpr
FP (20%)	yes	yes	yes	yes	no	no	yes	yes	yes
FP (40%)	yes	yes	yes	yes	no	no	yes	yes	yes
FP (80%)	yes	yes	yes	yes	yes	yes	yes	yes	yes
None	yes	no	no	no	no	no	no	no	no
$S_{\$}$	yes	yes	no	yes	no	no	no	yes	no
S_{BP}	yes	no	no	no	no	no	no	no	no
$S_{\$BP}$	yes	yes	yes	yes	yes	yes	yes	yes	yes
$R_{\$ (20\%)}$	yes	yes	no	yes	no	no	no	yes	no
$R_{\$ (40\%)}$	yes	yes	no	yes	no	yes	no	yes	no
$R_{\$ (80\%)}$	yes	yes	no	yes	no	yes	no	yes	no
$R_{\$ (100\%)}$	yes	yes	no	yes	no	no	no	yes	no
R_{BP}	yes	no	no	yes	no	no	no	no	no
$R_{\$BP (20\%)}$	yes	yes	yes	yes	no	no	yes	yes	yes
$R_{\$BP (40\%)}$	yes	yes	yes	yes	no	no	yes	yes	yes
$R_{\$BP (80\%)}$	yes	yes	yes	yes	yes	yes	yes	yes	yes
$R_{\$BP (100\%)}$	yes	yes	yes	yes	yes	yes	yes	yes	yes

SimPoint Relative Error

	ammp	art	mcf	gcc	parser	perl	twolf	vortex	vpr	AVG
50K	0.0215	0.0406	0.0923	0.2569	0.4103	0.2278	0.3408	0.1537	0.3262	0.2078
50K-SMARTS	0.2171	0.3206	0.0435	0.0235	0.0565	0.0226	0.0057	0.0636	0.0037	0.0841
10M	0.0485	0.003	0.0066	0.0246	0.0521	0.2308	0.0035	0.0117	0.0052	0.0429
10M-SMARTS	0.0485	0.008	0.0066	0.0193	0.1205	0.2303	0.0612	0.0121	0.0258	0.0591

SimPoint time

	ammp	art	mcf	gcc	parser	perl	twolf	vortex	vpr	AVG
50K	501	856	850	1030	925	491	594	545	429	691.22
50K-SMARTS	1841	1119	3497	2576	3007	1451	1680	1561	1303	2003.9
10M	2686	1535	9389	2548	1444	979	669	1254	1026	2392.2
10M-SMARTS	3549	2179	12154	4421	3191	2205	1245	2279	1954	3686.3