Long-term Performance Bottleneck Analysis and Prediction

Fei Gao and Suleyman Sair
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, NC 27695
Email: {fgao, ssair}@ncsu.edu

Abstract— **Identifying performance bottlenecks is important for microarchitects and application developers to produce high performance microprocessor designs and application software. Many techniques are used for this purpose, including simulation, software profiling and hardware event counters. Recently long-term program behavior has been getting more attention from researchers because of its potential applications in system-level, as well as program-level optimizations.**

In this paper, we study performance bottlenecks from a long-term program behavior viewpoint by classifying dynamic program execution into bottleneck phases - the portions of execution that have similar performance bottlenecks. We propose an event counter based performance model that can accurately estimate the performance cost for critical system events. Based on this model, we propose the bottleneck vector as the basis of long-term performance bottleneck analysis and a runtime bottleneck phase tracking scheme. In addition, three bottleneck phase prediction schemes are studied. Finally, we present an application of our performance bottleneck analysis model - an adaptive value predictor, which improves average performance by 7% when compared to the original value predictor design.

I. Introduction

The performance of a modern microprocessor benefits from improvements in circuit technology as well as microarchitecture. Over the past decade, a typical microarchitecture has evolved from in-order and scalar to out-of-order and superscalar with speculative execution. However, as processors enjoy high instruction level parallelism from modern microarchitectures, system-level performance becomes harder to understand and the question, "Where have the cycles gone ?", becomes harder to answer, because many events overlap and interact with each other. Effective performance analysis becomes essential for microarchitects and application developers to diagnose the code behavior and provide optimizations.

There are many approaches developed to analyze program performance. The most convenient way is software-based analysis tools, such as simulation [9], [2], [12] and instrumentation [17]. Simulation can provide details of program behavior, but causes several orders of magnitude slowdown. Instrumenting original code can catch dynamic events, but there still is code overhead and also the instrumentation code may change the behavior of the original program [17]. On the hardware side, modern microprocessors provide hardware support for performance analysis, called event counters [18], [16]. These counters give an inside view of how the program interacts with the underlying hardware. Users can access the counters with operating system and library support.

In addition to performance, understanding long-term program behavior interests researchers because of the potential long-term optimizations, such as power and thermal management. It has been shown that programs exhibit periods of similar behavior during their execution, called program phases [1], [6], [15]. The reason behind the phase phenomena is the regularity of code execution. Tracking the footprint of instructions [15] and working sets [6] are typical approaches to capture program phases.

Even though discovering general program behavior gives us hints with regards to areas of similar performance, it does not actually identify the problematic regions of execution. In this paper, we analyze performance bottlenecks from a long-term point of view. Performance bottlenecks are both hardware and software dependent. Hardware limitations, such as limited cache capacity, can slowdown program execution. The characteristics of a program impacts performance as well, such as the amount of available inherent code parallelism. Unlike previous long-term work that catch the regular patterns of program behavior to predict performance changes, our work provides a system-wide performance diagnosis to find out what causes the current performance change and what the next performance bottleneck will be. This inside view of system performance can guide runtime optimization directly without searching the optimization design space to find the most beneficial choice as prior program phase work does.

The contributions of this paper are:

- A counter based long-term performance model to quantify the performance impact of different system events,
- A vector based bottleneck phase tracking scheme to capture program bottleneck behavior at runtime,
- Performance bottleneck phase prediction schemes to guide system optimization.
- Runtime performance bottleneck analysis guided adaptive value predictor.

The rest of the paper is organized as follows. We first present related work on performance analysis and long-term program behavior analysis in Section II. General performance issues in a modern microprocessor are discussed in Section III. Then the experimental methodology is described in Section IV. In Sections V, VI and VII, we present our long-term performance model, bottleneck phase tracking, and prediction schemes respectively. In Section VIII we present an application to demonstrate the effectiveness of our model. Finally Section IX concludes our paper.

1-4244-9707-X/06/$20.00 ©2006 IEEE

II. Related work

In this section, we discuss previous work on performance analysis and long-term program behavior analysis.

Software-based analysis tools are very useful to get the basic information about a program's execution. Simulators [9], [2], [12] and instrumentation tools [17] are widely used for this purpose.

Meanwhile modern microprocessors provide hardware support for performance analysis with event counters. These counters are built into the chip to measure corresponding events [18], [16]. These event counter based performance analysis tools provide accurate and valuable inside view of dynamic program behavior. However, event counters can not accurately attribute those events to instructions, especially for out-of-order machines. For this reason, Dean et al. [5] proposed an instruction-level profiling technique - ProfileMe that samples instructions. As a sampled instruction moves through the processor pipeline, a detailed record of all interesting events and pipeline stage latencies is collected. ProfileMe also supports paired sampling, which captures information about the interactions between concurrent instructions. Similarly, to reveal the interactions of different instructions, Fields et al. [8] proposed an interaction cost model that can associate execution cycles to one instruction, or multiple instructions processed in parallel. In our work, we focus on long-term performance behavior rather than fine-grain analysis. In addition, we propose a low-overhead performance bottleneck model based on event counters instead of expensive instruction-level statistics.

Long-term program behavior became an active research area because of its potential applications for performance, power and thermal management optimization. Balasubramonian et al. [1] use a conditional branch counter to detect program phase changes. Dhodapkar et al. [6] define a program phase as the instruction working set of the program (i.e. the set of instructions touched in a fixed interval of time). Program phase changes are detected by comparing consecutive instruction working sets using a similarity metric called the relative working set distance. Sherwood et al. [15], [14] propose the use of basic block vectors (BBVs) to detect program phase changes. BBVs keep track of execution frequencies of basic blocks touched in a particular execution interval. Phase changes are detected when the Manhattan distance between consecutive BBVs exceeds a preset threshold. Duesterwald et al. [7] observed the repeating program behavior on IBM Power3 and Power4 processors and proposed table-based predictors that use performance metrics in past intervals as an index into the table to predict future behavior instead of tracking instructions. The difference between our work and previous long-term program behavior analysis is that we focus on performance bottleneck analysis - the cause of performance behavior - to provide system-wide performance diagnosis from a long-term viewpoint. In addition, our scheme can identify the most beneficial optimization choice directly, without searching the optimization design space as prior phase work does.

III. Performance bottlenecks

Performance becomes hard to understand in a modern superscalar out-of-order microprocessor because of the increasing number of inflight instructions and the interactions between them. Therefore, it is necessary to review the instruction flow of a modern microarchitecture and the potential factors that could cause performance slowdowns, before we do further performance bottleneck analysis.

A. Superscalar out-of-order microarchitecture

Figure 1 shows a 7-stage out-of-order microarchitecture. The stages are IF (Instruction Fetch), ID (Instruction Dispatch), IS (Issue), RR (Register Read), EX (Execution), WB (Writeback) and RE (Retire).

In the IF stage, the next instruction logic produces the next PC the processor will fetch. The BTB (Branch Target Buffer) identifies the type of the current instruction. For conditional branches, the branch predictor is involved for direction prediction. Unconditional branches are always predicted taken and the next PC is obtained from the BTB. Return instructions are a special case where a RAS (Return Address Stack) provides the return address. Otherwise, the default next PC is current PC plus one. After generating (predicting) the next instruction address, the fetch engine accesses the memory hierarchy to fetch the instructions into the pipeline.

In the ID stage, the fetched instruction is decoded first. Next, registers are renamed. The source registers are renamed by checking the renaming table. The output register gets a new physical register from the free list and the corresponding entry in the renaming table is updated. Then, this renamed instruction is dispatched into the issue queue to wait for issuing.

In the IS stage, ready instructions are selected to move to EX stage, if the corresponding functional units are available and issue logic has enough bandwidth.

In the RR stage, the issued instructions read values from the integer or floating point register files or the forwarding paths.

In the EX stage, the ALU or the Floating-Point Unit executes instructions. The execution latency depends on the type of instruction. Typically, division is the most expensive instruction. Load instructions have variable latencies that depend on whether they hit or miss in the L1 or the L2 cache.

In the WB stage, the outcome of EX stage is written back to the register file. Also, the mispredicted branches are recovered in this stage.

In the RE stage, the instruction retires if it is safe. If the exception bit is set for this instruction, recovery operations take place. All instructions after the exceptional instruction are flushed and refetched.

B. Potential performance bottlenecks

There are many factors that can lead to performance loss. Based on whether the performance constraints are hardware or software dependent, we classify those constraints into two categories - capacity constraints and inherent constraints, which are listed in Figure 2. The capacity constraints are

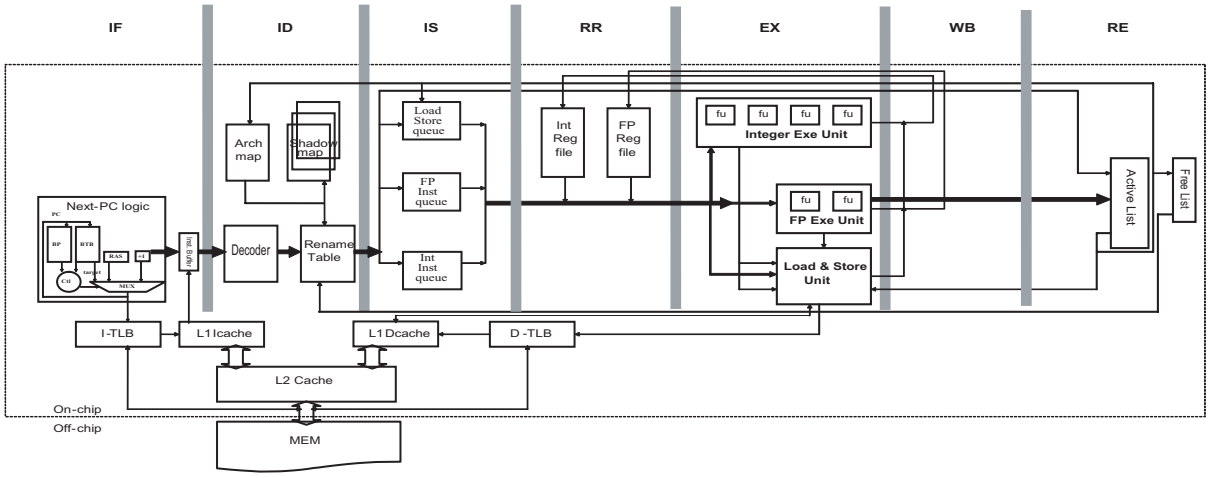

Fig. 1. 7 stage out-of-order superscalar microarchitecture

	Capacity Constraints (hardware)		Inherent Constraints (hardware and software)
IF	• Branch predictor size • BTB size • RAS size • I-cache size • I-TLB size	• Fetch width • Number of predictions/cycle • Prediction latency • I-Cache/memory latency • I-TLB miss latency • # cache read misses inflight	• Branch prediction accuracy • I-cache miss rate • I-TLB miss rate
ID	• Inst buffer size • Renaming resources # physical registers # shadow map	• Decoding width • Dispatch width	
IS	• Instruction Queue size • Load/Store Queue size • ROB size	• Issue width	• # inst issued/cycle
RR	• # read port of reg file		
EX	• # Int Unit • # FP Unit • # LD/ST Unit • D-cache size • D-TLB size	• Exe latency • Branch misprediction penalty • D-cache/memory latency • D-TLB miss latency • # cache read misses inflight	• D-cache miss rate • D-TLB miss rate
WB	• # write port of reg file		
RT	• Commit width • # cache write misses inflight		

Fig. 2. Major performance constraints

Fetch/Retire width	4 instructions
Branch predictor	gshare 16K entries
BTB	1K entries
RUU size	128
Load/store queue	64
Functional units	4 intALU, 2 int mul/div
I-TLB	64 entries, 30 cycle miss latency
D-TLB	128 entries, 30 cycle miss latency
I-cache L1	32KB, 2-way set associative, 32 byte line, 1 Cycle hit latency
D-cache L1	32KB, 2-way set associative, 32 byte line, 1 Cycle hit latency
L2 cache	1MB, 4-way set associative, 64 byte line, 12 Cycle hit latency, 120 Cycle miss latency

TABLE I

ARCHITECTURAL CONFIGURATIONS

caused by limited hardware resources, such as cache size, or read/write ports. The inherent constraints are both hardware and software dependent. For example, branch misprediction rate is impacted by branch predictor configuration, as well as the branch characteristics of the program.

IV. METHODOLOGY

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha AXP ISA. Table I presents the configuration parameters for the baseline microarchitecture. This work studies 12 SPEC 2000 integer benchmarks: `mcf`, `parser`, `vpr`, `gzip`, `crafty`, `gcc`, `gap`, `parser`, `perl`, `eon` and `twolf`. We run each program for 10 Billion committed instructions from the beginning.

V. COUNTER-BASED PERFORMANCE MODELING

In this section, we present an event counter based performance model. For modern microprocessors, event counters are becoming a standard on-chip resource. Utilizing event counters is a low overhead mechanism to monitor the behavior of microprocessor execution. However, for modern complex microarchitectures, event counters are not accurate enough for fine-grain instruction-level performance analysis because of the overlapping of events and the interaction between instructions. Yet, from a long-term program performance point of view, the overlapping effect can be approximately modeled with the event counter statistics.

The goal of our counter-based performance model is to provide runtime performance information to identify the performance bottlenecks that a microprocessor suffers. As we described in Section III, there are many constraints that impact performance. With the consideration of hardware cost/effectiveness, investing the limited event counters to those critical events is a reasonable choice. For our generic superscalar microarchitecture, the critical events we choose are: I-TLB miss, IL1 miss, IL2 miss, direct branch mispredictions, issue queue being full, resource contention, expensive instructions, D-TLB miss, DL1 miss, DL2 miss and indirect branch mispredictions. While other events may be critical for

	Events	Cost Model
IF	I-TLB miss	N * $P_{itlb\text{-}miss}$
	IL1 miss	N * $P_{il1\text{-}miss}$
	IL2 miss	N * $P_{il2\text{-}miss}$
ID	Direct Branch misprediction	N * ($P_{misbp\text{-}dir}$)
IS	Queue Full	N – Cost(exe)
EX	Resource Contention	N / f
	Expensive Instructions	N * (latency - 1) / f
	D-TLB miss	N * $P_{dtlb\text{-}miss}$ / f
	DL1 miss	N * $P_{dl1\text{-}miss}$ / f
	DL2 miss	N * $P_{dl2\text{-}miss}$ / f
WB	Indirect Branch Misprediction	N * ($P_{misbp\text{-}idir}$)

Fig. 3. Performance model

a specific architecture or execution situation, these critical events we define are enough to reveal the general performance behavior. In addition, our analysis algorithm is generic. It can be extended and applied to new cases.

A. Long-term event cost model

The counters give the numbers of critical events for a sampling interval. However, these raw numbers can not tell us how important each of them is from the whole microprocessor performance viewpoint. For example, we should treat L1 cache misses and L2 cache misses differently, because the miss latency difference between them is almost 10 fold. It is clear that different critical events have a different impact on the overall microprocessor performance. Our model tries to translate those raw numbers of different events into a quantitative representation of performance.

The modeling of each event is shown in Figure 3.

- I-TLB miss: An I-TLB miss causes a fetch engine stall. The cost of I-TLB miss events is the product of the number of events and the I-TLB miss penalty, because when instruction fetch is stalled, a pipeline bubble will be created during the waiting period.
- I-L1 miss: The cost of I-L1 misses is the product of the number of events and the I-L1 miss penalty.
- I-L2 miss: The cost of I-L2 misses is the product of the number of events and the I-L2 miss penalty.
- Direct branch mispredictions: A direct branch misprediction causes a pipeline flush. The cost of direct branch mispredictions is the product of the number of events and branch misprediction penalty. The reason is the ID stage is still in in-order and a pipeline bubble will be created during the period of waiting for pipeline recovery.
- Resource contention: Some instruction types will be stalled if there are not enough idle functional units. The cost of resource contention is the event number divided by a parallelism coefficient `f`, because resource contention doesn't cause a whole pipeline stall. Other type ready instructions can execute without waiting. So the performance impact of resource contention is proportional to the number of contention events and inversely proportional to the ILP. Since IPC is an approximation of available ILP, the coefficient `f` we choose is IPC plus one. The addition is to avoid amplifying the event cost when IPC is less than 1.
- Expensive instructions: Some expensive instructions, such as division, take a longer time to finish. The cost of expensive instructions is the product of the number of events and the extra execution latency divided by the parallelism coefficient `f`.
- D-TLB miss: The cost of D-TLB misses is the product of the number of events and D-TLB miss penalty divided by parallelism coefficient `f`.
- D-L1 miss: The cost of D-L1 misses is the product of the number of events and D-L1 miss penalty divided by parallelism coefficient `f`.
- D-L2 miss: The cost of D-L2 misses is the product of the number of events and D-L2 miss penalty divided by parallelism coefficient `f`.
- Indirect branch mispredictions: Mispredicted indirect branches cause a longer instruction fetch engine stall because they are resolved later. The cost is the product of the number of events and the misprediction penalty.
- Issue queue being full: The issue queue being full causes a pipeline stall. But stalls in the EX stage is the major reason for the issue queue being full. In order to avoid accounting for the same cost twice, the cost of issue queue being full is the number of events minus the cost of events in EX stage.

B. Model verification

The goal of our cost model is to quantify the performance effects of events based on the bubbles that are inserted into the pipeline. The straightforward way to verify our cost model is to compare the actual execution cycles to the sum of the execution cycles in an ideal pipeline (i.e. pipeline throughput without bubbles) and the event costs from our model. The formulas used for model verification are as follows.

$$T_{model} = T_{ideal} + \sum_{i} Cost_i \tag{1}$$

$$R_{error} = |T_{real} - T_{model}|/T_{real} \tag{2}$$

In Formula 1, the predicted execution time (`Tmodel`) from our model is computed by adding the ideal execution time (`Tideal`) and the total event cost (`Costi`). The ideal execution time is the number of executed instructions divided by the issue width, assuming each pipeline stage takes one cycle. Formula 2 represents the relative model error rate (`Rerror`) calculated by dividing the absolute value of the difference between real execution time and predicted execution time with the real execution time. Figure 4 shows the error rate for different execution periods - 1M, 10M, 100M and 1B instructions. On average, our model can achieve a 5% error rate, when l0M or more instructions are executed.

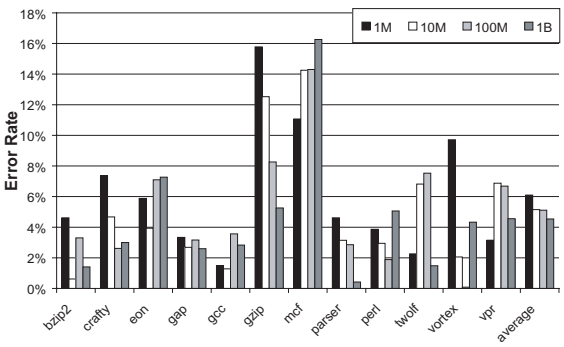

Fig. 4. Performance model verification

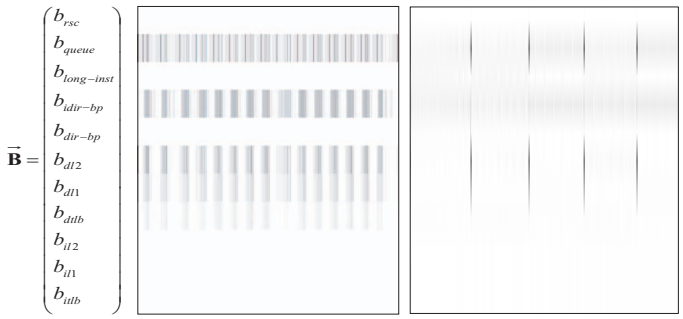

Fig. 5. Performance bottleneck phase behavior for `bzip2` and `gap`

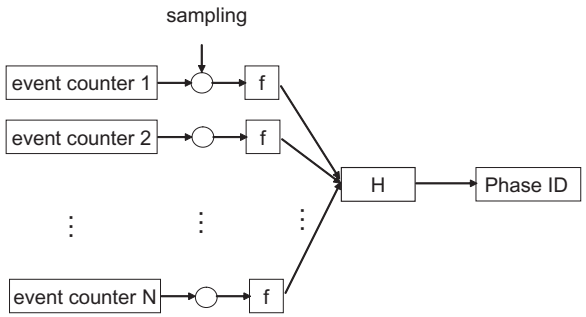

Fig. 6. Bottleneck phase tracking architecture

VI. Performance bottleneck phase tracking

In the previous section, we presented an approach to collect system performance information. Now we discuss how to analyze performance bottlenecks based on that information.

A. Bottleneck vector

Choosing an appropriate quantitative representation is very important for data analysis. A good representation can make analysis much easier and more accurate. For our model, we choose a vector to represent the system performance information. There are two reasons why we use a vector. The first is that the data we collect are disjoint values in event counters. These counters convey different information about different components in the microprocessor and will be handled separately. Therefore, keeping data in a vector will be an appropriate way without information loss. The other reason is that vectors are a powerful and fundamental mathematical representation. Computations based on vectors are relatively easy in both computation and hardware cost.

Based on the event counters we described in the last section, the bottleneck vector has eleven dimensions which corresponds to each counter. The vector is shown in Figure 5.

B. Bottleneck phase

Researchers proposed the program phase concept to describe the phenomena that a program exhibits repeating long-term behavior. The principle behind program phases is executing instructions visiting the static program with regular patterns. For example, in a loop, the static loop body will be visited repeatedly during execution. It is not a surprise that program phase phenomena has a similar effect on performance bottlenecks because performance bottlenecks are determined by both hardware and software. With a fixed hardware configuration, regular software patterns result in regular bottleneck behavior, called bottleneck phases.

To illustrate the bottleneck phase behavior, we track the bottleneck vectors for SPEC benchmarks `bzip2` and `gap` for a 10B instruction execution, which is sampled every 1M instructions. Each element of the vector is shown in parallel in the behavior graph. The bottleneck intensity is represented by darkness. The darker the points are, the more intense the performance bottleneck is. In Figure 5, we can clearly see the regular pattern of bottleneck changes.

C. Bottleneck phase tracking

Identifying bottlenecks is not a difficult task from a mathematical perspective. We can use the vector itself as an ID. However, hardware cost and computation complexity will make this design very expensive. One efficient way to keep component cost information without involving too much hardware budget is hashing the elements of the bottleneck vector into a phase ID. The tracking scheme is shown in Figure 6. For a given sampling interval, event counters are sampled and cleared. Then those raw event numbers are processed by an array of function blocks. The function blocks perform two computations. One is average cost computation based on the cost model. In our study, we use cost per 1K instructions as the average cost for a sampling interval. The other is cost quantization, which is representing the cost from the first step in terms of the number of cost units. The cost unit value is a fixed parameter defined by users, depending on the accuracy requirement. Dividing by the cost unit facilitates classifying similar cost values into the same cost group. cost value. Finally, the elements of the processed bottleneck vector are hashed into a bottleneck phase ID.

VII. Performance bottleneck phase prediction

In the previous section, we described how to track and identify a bottleneck phase at runtime. There may be a need to

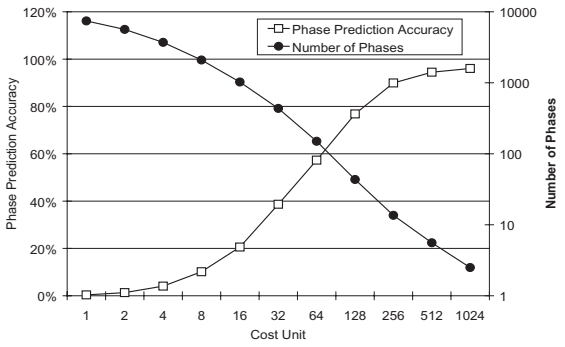

Fig. 7. Prediction accuracy vs. phase ID numbers

predict bottlenecks in advance to guide system optimization. In this section we will present the predictability of performance bottlenecks with several prediction schemes.

As we described in Section VI, bottleneck phase IDs depend on both the cost unit and the hash function. A smaller cost unit value can identify even small differences between vector components and classify them into separate phases, thus creating more unique phase IDs. With an increasing number of phases, prediction becomes more difficult. A last value predictor is the simplest prediction scheme, which predicts the current phase as the next phase. The last value predictor relies on phase stability. In Figure 7, we plot its prediction accuracy and the number of phases for several cost units. We can see that as the cost unit increases, the number of phases decreases and the corresponding prediction accuracy increases.

In addition to a last value predictor, we studied two other prediction schemes - history predictor and Markov predictor. The history predictor keeps the previous phase information and predicts the phase with the most number of appearances in the history as the next phase prediction. The history predictor can filter the sudden phase change noises, providing more stable prediction than last value predictor if there are frequent phase transitions. The third prediction scheme we studied is a Markov predictor, which can track the phase transitions. We use history information to identify the transition state. The Markov table is indexed by the lower bits of the hashed history information. Each entry consists of the higher bits of the hashed history information as the tag and the predicted phase ID. For every prediction, Markov prediction table is checked first. On a hit, the phase ID stored in this entry will be the prediction. On a miss, we use the current ID as the prediction. In Figure 8, we compare these predictors. The cost unit is 128 for this figure. The sampling period is 1M instructions and 10B instructions are executed in total. The history length for the history and Markov predictors is 3. The Markov table has 256 entries. We can see that for most of the benchmarks, the three predictors get similar accuracies. This is because bottleneck phases are fairly stable. For `mcf`, since the numbers of phases and transitions between phases are high, the Last predictor gives extremely low prediction accuracy and the Markov predictor provides much more accurate predictions.

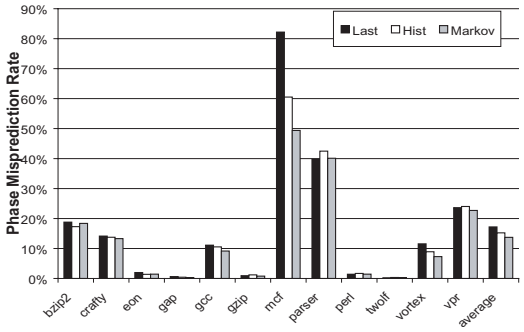

Fig. 8. Bottleneck phase predictors

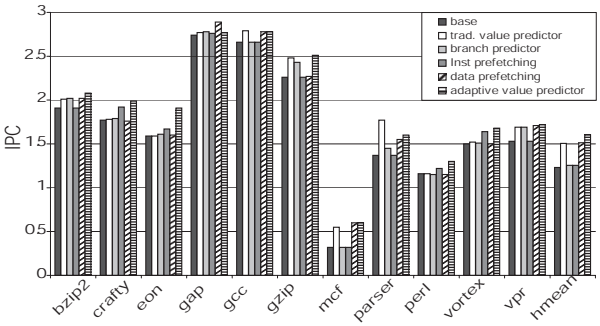

Fig. 9. Performance results of adaptive value predictor

VIII. Potential applications

The most direct application of our technique is resource management for adaptive microarchitectures. The advantage of an adaptive microarchitecture is the ability to adjust system resources at runtime to meet application needs. To our knowledge, current adaptive microarchitecture designs focus on a specific microprocessor component, such as cache ways, issue queue size and so forth. There is no work that exploits system-wide adaptation - to do adaptation for multiple components simultaneously. The main reason is the lack of system-wide information and analysis. Our model provides a way to do system level performance analysis and prediction that could be used as a system-wide adaptation guide. Next, we will provide a system-wide adaptation design - an adaptive value predictor.

Value prediction is an effective technique to eliminate data flow restrictions by predicting register values before they are resolved [13]. Programs exhibit value locality, i.e. recurrence of register values. Value prediction exploits value locality to predict register values with a predictor, in order to break data dependences among instructions and provide more available instructions to improve ILP. Meanwhile, Gonzalez et al. [10] proposed control-flow speculation through value prediction for superscalar processors. In their work, they predict the outcomes of branches by predicting the value of inputs with a value predictor and performing an early computation of results according to the predicted values. In addition, researchers also use value prediction to assist prefetching, in order to tolerate long memory access latencies [11], [4].

Since fairly large value prediction tables are needed for

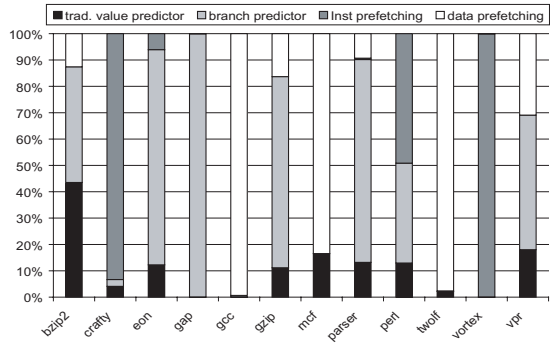

Fig. 10. Execution breakdown of 4 adaptation choices

high accuracy, it is infeasible to include separate value predictors in hardware to take advantage of all these techniques. There is an alternative adaptive design however: an adaptive value predictor, which can work for branch prediction, value prediction or prefetching based on which bottleneck is most costly - control dependence, data dependence or memory access. As we observed that the performance bottleneck varies for different applications and even different execution potions of same application, an adaptive value predictor can maximize the effectiveness of the value predictor resource by its adaptivity. In our experiments, we use a 16K entry two-delta stride value predictor to perform value or address prediction. Four functionalities are provided by the adaptive value predictor - traditional value predictor, branch prediction assistant, instruction prefetching and data prefetching. Which choice depends on the corresponding bottleneck cost - issue queue being full, branch misprediction, I-cache access, and D-cache access. The bottleneck vector is sampled every 1M instructions. Bottleneck phases are predicted with a last value predictor. The adaptive value predictor tunes its functionality to the maximum bottleneck cost after each sampling interval.

The results are shown in Figure 9. The adaptive value predictor can improve performance by 30% compared with the baseline and 7% compared with the best alternative, data prefetching. Figure 10 shows the breakdown of program execution with these 4 adaptation choices. We can clearly see the adaptive value predictor tuning its functionality to the characteristics of each benchmark.

IX. Conclusions

Attacking performance bottlenecks is one of the main goals for microarchitects and application developers to achieve high performance and efficiency. Identifying and analyzing performance bottlenecks is the basis of system and software optimizations. Modern microprocessors provide dedicated hardware support for performance analysis - event counters, which can provide an inside view of program runtime behavior. Programs themselves exhibit regular execution patterns in the long-run. Many techniques are proposed to identify and predict the regular behavior by tracking the footprint of instructions or working set.

In this paper, we exploit the program performance bottleneck behavior from a long-term program behavior point of view. We proposed an event counter based performance model and a mathematical representation of runtime performance bottlenecks - the bottleneck vector. Based on these, we can accurately capture the runtime performance bottlenecks and provide accurate bottleneck prediction. We also present an adaptive value predictor to demonstrate a system optimization application of our model. In the future, we will apply our model to power and thermal management to provide efficient management schemes to improve power efficiency with as little performance loss as possible.

References

[1] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *Proc. of the 33rd Annual Intl. Sym. on Microarchitecture*, Dec 2000.

[2] E. A. Brewer, C. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Measurement and Modeling of Computer Systems*, pages 247–248, 1992.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[4] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[5] J. D., J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[6] A. S. Dhodapkar and J. E. Smith. Managing multiconfiguration hardware via dynamic working set analysis. In *Proc. of the 29th Annual Intl. Sym. on Computer Architecture*, May 2002.

[7] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[8] B. A. Fields, R. Bodk, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Micro-36*, 2003.

[9] A. Goldberg and J. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. In *IEEE Transactions on Parallel and Distributed Systems*, pages 28–40, 1993.

[10] J. Gonzalez and A. Gonzalez. Control-flow speculation through value prediction for superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[11] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[12] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[13] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[14] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[15] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of the 30th Annual Intl. Sym. on Computer Architecture*, Jun 2003.

[16] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro*, Aug 2002.

[17] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[18] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Supercomputing*, November 1996.