

Characterizing Performance in Virtualized Execution

Hussam Mousa*, Kshitij Doshi**, and ElMoustapha Ould-Ahmed-Vall**

*Dept. of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

**Intel Corporation
5000 W Chandler Blvd
Chandler, AZ 85249

Abstract

Workload execution in virtualized machines is rapidly becoming commonplace while support for self-virtualization is already available in mass market processors. From a performance analysis and prediction standpoint, virtualization introduces new sources of uncertainty in reasoning about the factors that impact overall system performance and efficiency. This paper describes a general methodology for profiling the vertical software and hardware stack of a virtualized platform. A key contribution of this paper is an instrumentation approach that enforces alignment between measurements of potentially correlated factors, so that statistical analysis techniques such as regression modeling can be used effectively against the data. The paper shows, through the use of preliminary examples, how one can use such instrumentation to obtain decompositions of the CPI (cycles-per-instruction) metric for two workloads when each is executed first on native hardware and then in an identically configured virtual machine

1. Introduction

In recent years a trend toward virtualization of machines has accelerated in enterprise data centers and other high volume processing environments [21]. This trend is driven in large part by business needs to transcend such rigidities of static deployments as power, space, and thermal footprints, while minimizing wasteful energy use by redistributing execution among fewer machines than are needed during peak demand periods [12, 21]. Vendors of popular processors have delivered architectural support for processor self-virtualization by removing the need for software based execution shadowing or emulation, further accelerating this trend [1].

With virtualization entering the mainstream, it has become important to characterize its effects on workload performance, so that machines can be configured aptly and optimized for virtualized execution. It is also necessary to obtain profiles of platform behaviors under virtualized execution so that future modifications of software and hardware can take due account of factors that limit the performance of software when it is deployed inside virtual machines.

In particular, because virtualization gives rise to additional pressures on micro-architectural resources and introduces new types of behavior that affect performance, it is essential to identify the hardware and software factors whose impacts dominate performance and to calibrate those impacts. For instance, the ability to relate increases in virtual machine instances on a given physical platform to changes in the demands upon the resources of the physical platform helps identify efficient load balancing policies when an ensemble of virtual machines is scheduled across a collection of physical machines. Similarly, a detailed breakdown of the overheads that arise when a workload's execution is virtualized provides an opportunity to optimize the workload, perhaps along with hardware configuration, so that the overheads are removed or reduced.

For non-virtualized - or native - execution, well matured performance profiling instrumentations at multiple levels of the hardware and software hierarchy, supported by analysis and modeling techniques, has provided a rich source of understanding for workload execution. Simulations of instruction traces from native execution have been another source for obtaining critical modeling parameters such as per instruction overheads arising from dynamic conditions that generate execution stalls. By comparison, instrumentation and analysis techniques for virtualized execution are significantly more incipient.

Ironically, because virtualization introduces many dynamic behaviors - some through new usages of hardware such as co-execution of heterogeneous operating systems - its performance characterization demands high resolution yet minimally intrusive instrumentation. Likewise, its simulation requires a much broader range of instruction traces than what would be considered sufficiently representative for capturing native execution behavior.

This paper describes an approach for bridging the profiling instrumentation gap between native and virtualized execution, and shows by example a preliminary method for isolating the key contributors of performance overhead in both native and virtualized machine instances. As a first step, we apply workloads with single state behaviors in this study, but the methodology advanced in this paper is designed to apply equally to behaviors that are not steady over time. An aligned collection of hardware monitored as well as hypervisor (also known as the Virtual Machine Monitor) instrumented event counts is achieved with hypervisor embedded code, where the width of a single interval of event collection can be based on time, instruction count, or some other unit such as numbers of branches retired or intrinsic application phases. The triggers are programmatically generated either from the processor, or from within the workload that is of interest. Instrumentation efficiency is obtained by keeping control within the hypervisor to the maximum extent possible. This frugality and flexibility makes it possible to get fine grained intervals of hardware and software event data where the statistical relationships among events are captured within the interval boundaries. Each interval contains event statistics gathered from the hardware, the Virtual Machine Monitor, the Kernel and possibly the application itself. The paper illustrates how regression techniques can be applied to this data to gain key insights in the effects of virtualization on workload performance. We also demonstrate how the data can be used to quantify how the occurrence of various micro-architectural and virtualization events affects virtualization overhead.

2. Background

Virtualization has been used in various forms for several decades. Recently the renewed interest in virtualization produced two dominant models: *Para-Virtualization* and *Hardware Virtualization* [1]. Para-Virtualization runs on unmodified hardware and relies on the Virtual Machine Monitor (VMM) to manage the memory, allocate resource, and isolate the various guest kernels. It works by replacing critical calls in guest kernels with "para" calls to VMM equivalents. The VMM is then responsible for guaranteeing that calls comply with system policies and that guests are protected from failures by other guests.

Hardware Virtualization delegates many of the functions of the VMM to the processor. Guest kernels no longer need to be modified and can operate as if they are running on dedicated hardware. The guest kernel runs at slightly limited CPL0, the same hardware ring that native kernels execute within, while a new, root privilege is created for the hypervisor. The processor is designed to trap into hypervisor whenever one of a limited subset of operations is attempted from non-root privilege level, in order to prevent guest kernels from piercing the abstraction of the processor that is presented to them by the hypervisor, or to affect the isolation of other co-executing kernels. Such traps, generally referred to as VT exits, are a common source of overhead in many of the early implementations of hardware virtualization [18], and therefore are the target of considerable optimization through successive generations of processor implementations and hypervisor algorithms. Virtualization of hardware requires virtualization of activities on the I/O path as well.

The primary difficulty that arises in fine-grained performance profiling in a virtualized context is in achieving completeness of performance state capture. A typical system can have multiple autonomous privilege domains in concurrent execution, where guest kernels do not exert direct control over each other (for profiling purposes) or over the processor. As the VMM is the only software component with control over the entire system hardware and memory space, it is the logical choice for basing a whole system profiler; yet, the VMM typically does not have sufficient visibility into guest encapsulated software performance state. Thus, an effective profiler must operate from the VMM where it has complete and direct access to the processor's profiling features; and, its operation depends upon accurately allocating and aligning events to their appropriate domains. Guest kernel cooperation -even if to a very limited degree, also needs to be built-in so that guests can participate efficiently in triggering and pacing profile data collection and align activities at the workload level with performance data capture in the VMM at the desired granularity.

Finally, since the VMM is a major system level component, its actions will affect the overall workload behavior and the effects of that behavior on the processor. It is therefore important to include the VMM's events as part of the whole system profile. In other words, to capture a comprehensive snapshot of the system's behavior for any given execution interval, it is important to gather data from the processor, VMM, guest kernels, and executing workload. In the next section we present our proposed system design for a whole virtualized system profiler that addresses these challenges.

3. Virtualization Enabled Profiling

In this section we describe a profiling system we built to capture detailed behavioral snapshots of the entire hardware and software stack of a system -whether in native or virtualized execution. Subsection 3.1 presents the data model for our system, while subsection 3.2 describes the profiling infrastructure. We describe our experimental methodology and setup in section 4.

3.1 Data Model

		PMC data			Hypervisor data			Guest Kernel data		
Interval _m	CPI _m	PMC _{1m}	...	PMC _{nm}	VT _{1m}	...	VT _{nm}	OS _{1m}	...	OS _{nm}

Table 1. Data collection model for vertical virtualization profiling. For Natively executed programs, the Hypervisor data fields are set to 0.

The basic unit of data in our system is an execution interval; a sequence of instruction delimited by an arbitrarily defined event. Intervals can be defined based on time slices, number of retired instructions, or other micro-architectural or workload events. In this paper, we use intervals of equal number of retired processor instructions.

Each interval includes counters for events derived from three primary sources: hardware performance monitoring counters (PMC), guest kernels, and the hypervisor. The data model is depicted in Table 1. The system insures that events are synchronized and aligned to the intervals during which they were collected. Events are normalized to the number of retired instructions.

The collection of data at the granularity of an execution interval is motivated by the studies that demonstrate potential presence of varying performance behaviors during different parts of a workload’s execution [23]. By collecting the data at the granularity of small intervals (millions of retired instructions), we are able to capture and account for these different behaviors. For a longer discussion on the rationale behind this method of data collection, the reader is referred to [20, 19].

3.2 Virtualized Profiling System Design

To support the data model in Table 1, we developed a utility that achieves the following minimum objectives: aligned data collection from software and hardware sources, low overhead, and low application perturbation. The utility design is shown in Figure 1.

The system’s primary controls are located within the hypervisor. They are responsible for configuring, starting, stopping, and reading all the various profiling agents. This controller receives a “recipe” from the user - a plan for an entire profiling session. This eliminates the need for any communication between the user’s control system and the lower level profiling components during the profiling session. After configuring the various profiling agents, the controller synchronously starts profiling at all levels and waits for the trigger to end an interval. Upon receipt of the trigger, the controller again stops all the profiling agents in synchrony and reads the current event counts. These counts are stored in a buffer that is statically allocated within the hypervisor’s address space to avoid data transfer during a workload’s execution. It is important to note that the stopping and reading of the profiling agents usually occurs during a VT exit event so the guests are suspended anyway. We try to group these profiler’s maintenance activities with other normally occurring VT exit events. This minimizes workload perturbation, since those VT exits are a “normal” part of virtualized system execution.

If the memory buffer for storing event frequencies fills up, a virtual interrupt is raised which is handled by the user level control utility. This causes a transfer of data from the hypervisor space to the user space, but it generally only occurs a few times -and only during the execution of really long workloads.

The PMC drivers are written to achieve maximum flexibility, efficiency and low overhead. Virtualization events (such as VT exits, fault injections, etc.) are collected through an interface with Xentrace [23]. Presently we capture counts of events,

and we plan to capture more detailed information about specific events such as interrupt types and VT exit reasons in the future. Profiling agents that operate from guest kernels and capture guest performance state in synchrony will also be added to this framework in the near future.

Since guest kernels are in fact executing on virtual CPUs, a temporal mapping of virtual to physical CPUs is maintained for the purpose of proper designation of architectural performance events. Virtualization events collected are valid for the entire system, since they are effects of the hypervisor, a system wide component.

The native profiling version of our system is very similar to the above design except that the controller is located in the kernel instead of the hypervisor.

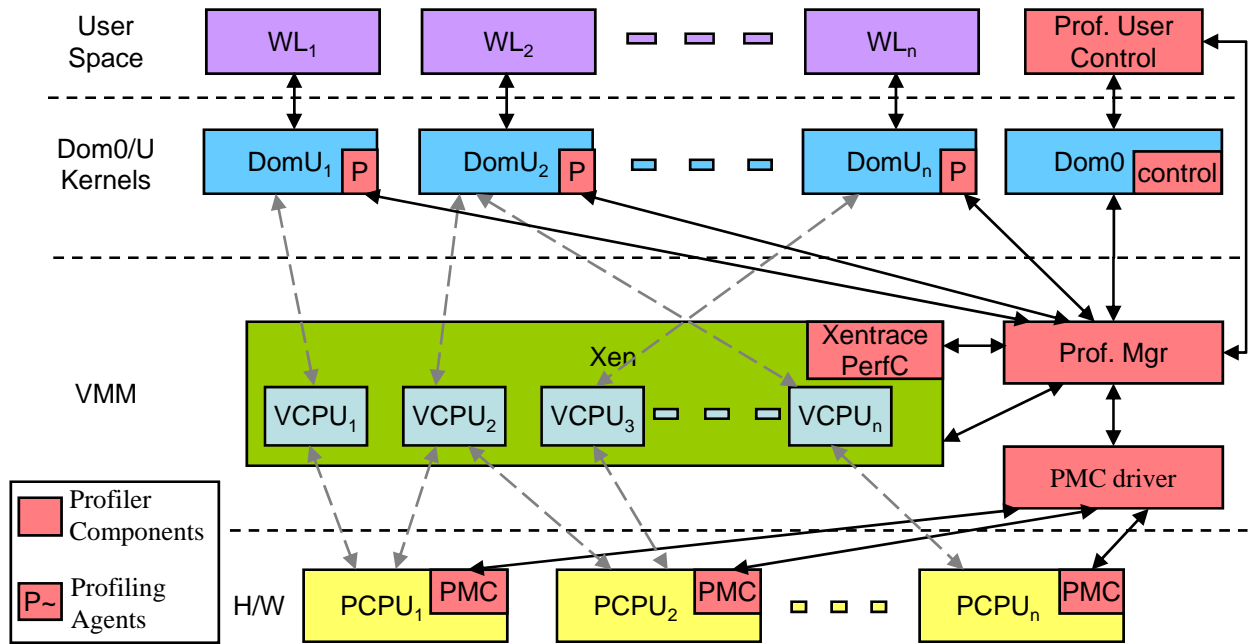


Fig. 1. Virtual Profiler overall system design. The profiling manager synchronizes the data collected from all the layers according to an arbitrary interval trigger.

4. Experimental Methodology

For this study, we collect 12 architectural events and 5 virtualization events. The events collected are listed in Table 2. Full descriptions of the architectural events are available in [13].

We run 10 benchmarks from the Dacapo java suite [4] release 2006-10-MR2 on the Sun Java Hotspot Client Virtual Machine (build 1.4.2.16-b05), and a custom benchmark consisting of building the Linux kernel image following random source file touches. Descriptions of the benchmarks are provided in Table 3.

Each benchmark is executed on a native platform based on SUSE SLES10 SP1 64-bit operating system and a virtualized platform based on the Xen hypervisor [3] and running an identical Linux guest kernel to the one in the native setup. Both the guest and the native kernel are configured with identical available memory, swap space and number of CPUs. The physical configurations are listed in Table 4.

Due to the fact that there are only two reconfigurable performance counters on the Intel Core 2 Duo, we use multiplexing as a technique for collecting more events than there are counters available. We group our events into sets that could be collected

<u>Architectural Events</u>	
<i>L1 Inst Cache Misses</i>	<i>Page Walks</i>
<i>L1 Data Cache Misses</i>	<i>Conditional Branch Mispredictions</i>
<i>L2 Cache Misses</i>	
<i>% Load Instructions</i>	<u>Virtualization Events</u>
<i>% Store Instructions</i>	<i>VT Domain Transitions</i>
<i>% Branch Instructions</i>	<i>VT Interrupt Events</i>
<i>% Multiply Instructions</i>	<i>VT Scheduling Events</i>
<i>% Divide Instructions</i>	<i>VT Page Fault</i>
<i>DTLB Misses</i>	<i>VT I/O Even</i>
<i>ITLB Misses</i>	

Table 2. Profiled events collected by VPROF for the results presented in section 5. Virtualization events might be an aggregation of several sub events (e.g. I/O events is the sum of I/O reads and writes).

<i>Dacapo</i>	
Antlr	Parse grammar and generate parser and lexical analyzer
bloat	Optimize and analyze Java byte code
eclipse	Execute JDT performance tests for the Eclipse IDE
fop	Parse, format and generate a PDF file
Hsqldb	Execute queries against an in memory database
ython	Interpret a Python benchmark
luindex	Use lucene to index some books
lusearch	Use lucene to text search keywords against a data set
pmd	Analyze Java classes for a range of code problems
Xalan	Transform XML documents into HTML
<i>Linux Build</i>	Touch several random source files and build a bzImage

Table 3. Benchmarks used in the experiments and their descriptions

	Native platform	VT Platform
Kernel	Linux 2.6.16.46-0.12	
Hypervisor	N/A	Xen 3.0.4_13138-0.40
Available Processors	2 Physical CPU	2 Virtual CPU
Sys Memory	1024MB (System limited)	
Swap file	1024MB	
Processor	Intel® Core™2 Duo (dual-core) 2.66 GHz 2 GB Main Memory, 4 MB L2 Cache, 32 KB L1 and 32 KB L2 cache	

Table 4. Experimental Setup.

simultaneously. We then divide our desired interval width by the number of such sets, and use this new sub-interval width for the actual collection. We then switch the PMC collected events among the different sets at the end of every sub-interval. Finally we scale the collected PMC accounts back to the full desired interval width. Software events are not multiplexed but collected across the total number of sub-intervals that span an interval. For this study, we set the full interval width to 60 million retired instructions, and with six counter sets, the sub-intervals were set to 10 million retired instructions.

5. Results and Analysis

Based on the understanding of the system level operation of a virtualized platform, we can think of the performance as being impacted by two distinct - yet cross interfering - factors: additional architectural events due to virtualization, along with a potential increase in their costs, and additional virtualization (hypervisor) related events. Both factors will increase the Cycles per Instruction (CPI) metric, while the second factor will also increase the path length (the number of instructions per unit of work). In this paper we focus on the CPI and plan to study the impact on increased path length in future work.

In the remainder of this section, we describe several analytical approaches that can be used to gain insights into which factors contribute to virtualization overhead and the extent of their impact. Note that the actual values and relative weights presented in the next subsections are very specific to the processor micro-architecture as well as hypervisor and operating system versions, and will very likely decline rapidly as processor vendors deliver architectural improvements to reduce the cost and frequency of hypervisor transitions and interruptions. We believe, however, that our methodology and analytical basis are broadly applicable to the study of virtualization under any processor architecture, hypervisor, and guest kernel.

5.1 CPI Decomposition with Multi Linear Regression

In this section, we describe a simple analytical approach to study virtualization overhead. It is based on the idea of CPI decomposition; that is to use regression analysis to statistically divide the CPI in its various components: e.g., core CPI, cache misses, branch misprediction, TLB misses and so on. This CPI decomposition is performed for both the native and virtualized execution. CPI decomposition also gives an estimate of how various VT events such as exits into the hypervisor affect the program execution time by contributing to an increased CPI. The comparison between these two decompositions yields valuable insights into how virtualization changes the cost (in terms of CPI overhead) of micro-architectural resources.

We decompose the CPI by applying multi-linear regression with the CPI as the dependent variable, and the various architectural and virtualization events as the independent variables.

$$Y = CPI = CPI_0 + a_1X_1 + a_2X_2 + \dots + a_nX_n + b_1V_1 + b_2V_2 + \dots + b_mV_m$$

where (X_1, X_2, \dots, X_n) and (V_1, V_2, \dots, V_m) are the different micro-architectural and VT events, respectively. As expected, no virtualization event is used to fit the regression equation for native execution. The events used in this study are listed in Table 1. To reduce the effects of multi-colinearity, we bundle the independent variables in groups of related events (e.g., TLB misses and page walks). The groupings and CPI contributions of individual events are shown in Table 5.

Figure 2 presents a visualization of the CPI decomposition for both the native and virtualized cases of the Linux build benchmark. By decomposing the CPI into the constituent components, we can observe the relative change due to virtualization and estimate how much of the overhead is due to the different events.

It is evident from the figure that for the Linux kernel build, virtualized execution generates a significant overhead. CPI for virtualized execution has an average value of 1.72, while the native case has an average CPI of only 1.03. Such large increase is understandable for this workload as the build activity is process creation intensive, and gives rise to large numbers of address space setup and teardown activities. This leads to many transitions to the hypervisor as a result of copy-on-write fault handling. Hardware assisted memory virtualization should remove the majority of such transitions in future processors.

The figure also indicates some of the key contributors to the architectural increase in CPI. It is clear that the load/store operations are having a much higher impact in the case of virtualized execution than in the native case. This is not surprising since virtualization can increase resource pressures and result in these operations experiencing increased cache and TLB misses.

The contribution of TLB misses and resulting page walks to CPI reduced considerably in VT execution of the Linux build benchmark. This detail should be interpreted with care: as cache misses increase from virtualization transitions, more

Event Categories	Events	Linux Native	Linux VT	Java Native	Java VT
Instruction Cache	IL1_MISSES	1.36%	5.65%	4.41%	7.17%
Data Cache	DL1_MISSES	1.62%	5.15%	1.23%	2.88%
	L2_MISSES				
Load/Store Unit	LOADS	19.9%	24.7%	33.6%	25.3%
	STORES				
TLB group	ITLB_MISSES	16.5%	4.56%	7.71%	2.15%
	DTLB_MISSES				
	PAGE_WALK				
Other	BR_MISPRED	60.6%	30.1%	54.1%	56.3%
	BR_RETIRED				
	DIV_OPS				
	MUL_OPS				
	CPI0 (constant)				
VT scheduling	VM Wake	0%	5.49%	0%	1.39%
VT transitions	VM Enter	0%	20.7%	0%	5.04%
	VM Exit				
VT interrupts	VT Page Fault	0%	6.12%	0%	0%
	VT Page Fault Inj				
	VT Exception Inj				
	VT Interrupt				

Table 5. Grouping of Events and their relative contributions to the overall CPU according to the CPU decomposition model

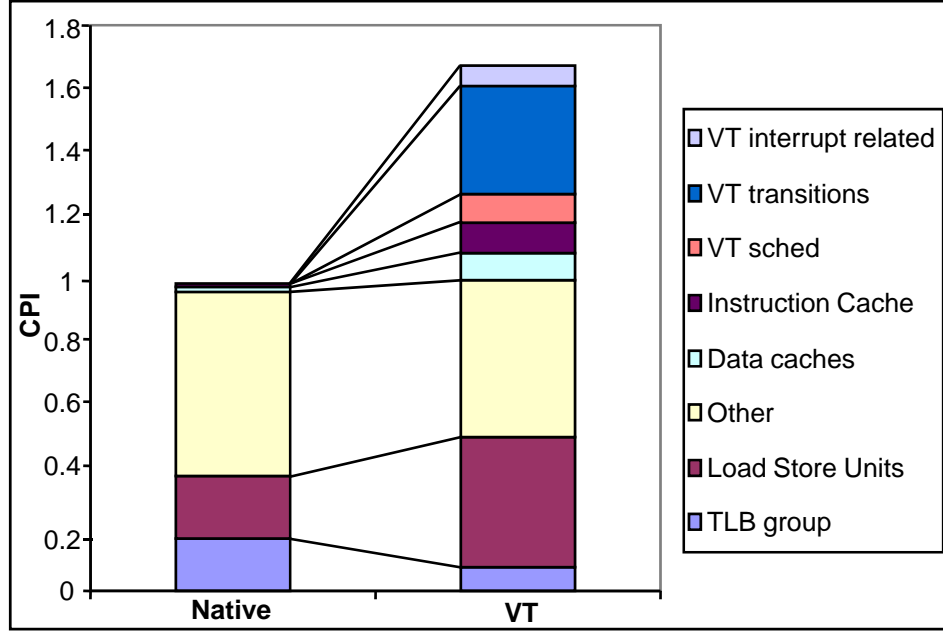


Fig. 2. CPI inflation and decomposition for Native and Virtualized execution of the Linux Build benchmark.

opportunities arise for cache miss handling to mask some of the cost of TLB misses that arise in parallel. Also, whereas a majority of TLB misses are handled in hardware during native execution, in VT execution, the virtualization of host physical memory by hypervisor logic changes the share of the burden that falls directly on the hardware –simultaneously raising the burden on software. Hardware assisted TLB virtualization (in future processors) can be expected reestablish the more prominent role of hardware in translation cache management.

Among the VT events, it is clear that VT transitions have a significant impact on the performance of this workload. In fact, more than 20% of the overall workload execution time (and close to half of the VT overhead) is attributable to this event.

In a similar way, Figure 3 provides the native and VT CPI decomposition for the Dacapo suite benchmark. For this workload, virtualization has a relatively low overhead: only 9%. The figure indicates that this overhead is mainly due to VT transitions and scheduling. We can also observe a change in the relative distribution of architectural contribution to the CPI, with an increase in pressure on the data caches, and an increase in the base CPI (the "Other" group). Again, the relative contribution of the TLB unit is diminished by the increase in cache misses and their respective impact on other architectural components.

The main drawback of multi-linear regression is that it can suffer from low accuracy since it cannot account for the non linear interaction between the independent variables. To assess the validity of the conclusions, we evaluate the different performance models using two main accuracy metrics:

1. The correlation coefficient between the predicted and actual CPI value.
2. The average absolute error: difference between actual and predicted CPI.

As shown in Table 6, the native and virtualizations models for both benchmarks produce encouraging accuracy numbers. In the next sub-section, we introduce the use of Model trees for as mechanism to gain insight into the difference between performance under native and virtualized executions. We are currently investigating the usefulness of Model trees and other regression techniques to perform a detailed CPI decomposition.

This section demonstrated how simple multi-linear regression can be used to assess VT overhead to determine the most important micro-architectural and virtualization events causing this overhead for a particular workload. In the next subsection we apply a non-linear analysis technique, Model Trees, to evaluate the relative importance of various virtualization and architectural factors in predicting the CPI metric.

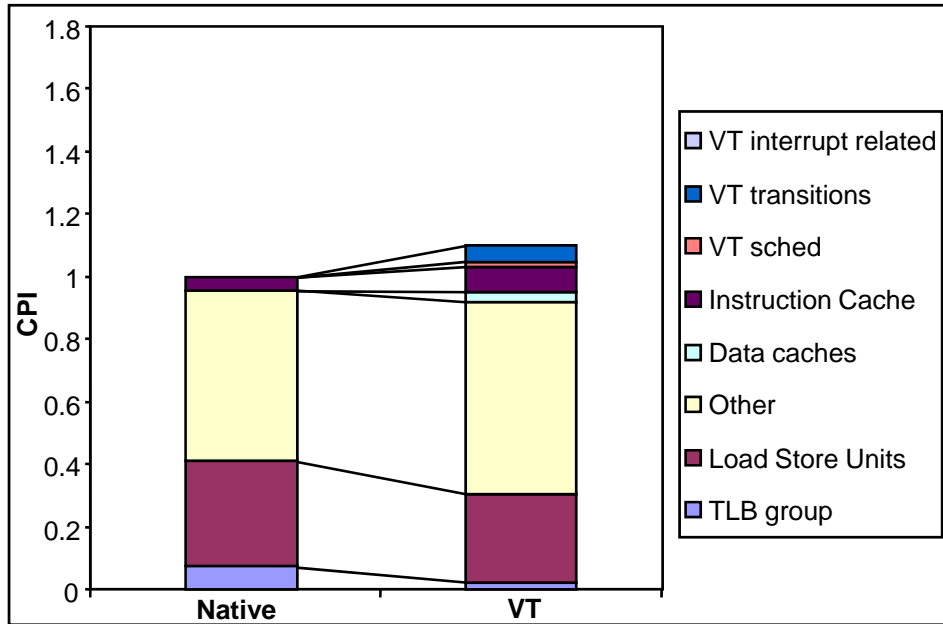


Fig. 3. CPI Inflation and Decomposition for Native and Virtualized Execution of Dacapo Suite.

Model	Correlation Coefficient	Average Absolute Error
Linux Native	83%	6.0%
Linux Virtualization	88%	8.72%
Dacapo Native	75.6%	10.72%
Dacapo Virtualization	80.91	10.09%

Table 6. Accuracy of the CPI inflation model presented in section 5.1

5.2 Model Tree analysis for VT CPI Estimation

In this section, we apply Model Tree analysis [20] to assess the effects of virtualization on workload performance. This approach allows for handling non-linear interactions between the different independent variables. Model Trees are a type of regression tree [5] that associate a Linear or Multi-linear model with the leaves of the tree.

To apply Model trees, we merge the two data sets (native and VT execution). The VT events that are available for virtualized execution are replaced with a single "VT" variable, which is either 0 or 1, respectively to indicate whether a given data point comes from the native or virtualized execution. This modification was able to present a better convergence for the models. In future work we will investigate the optimal set of virtualization events that can produce a converging Model Tree with high accuracy.

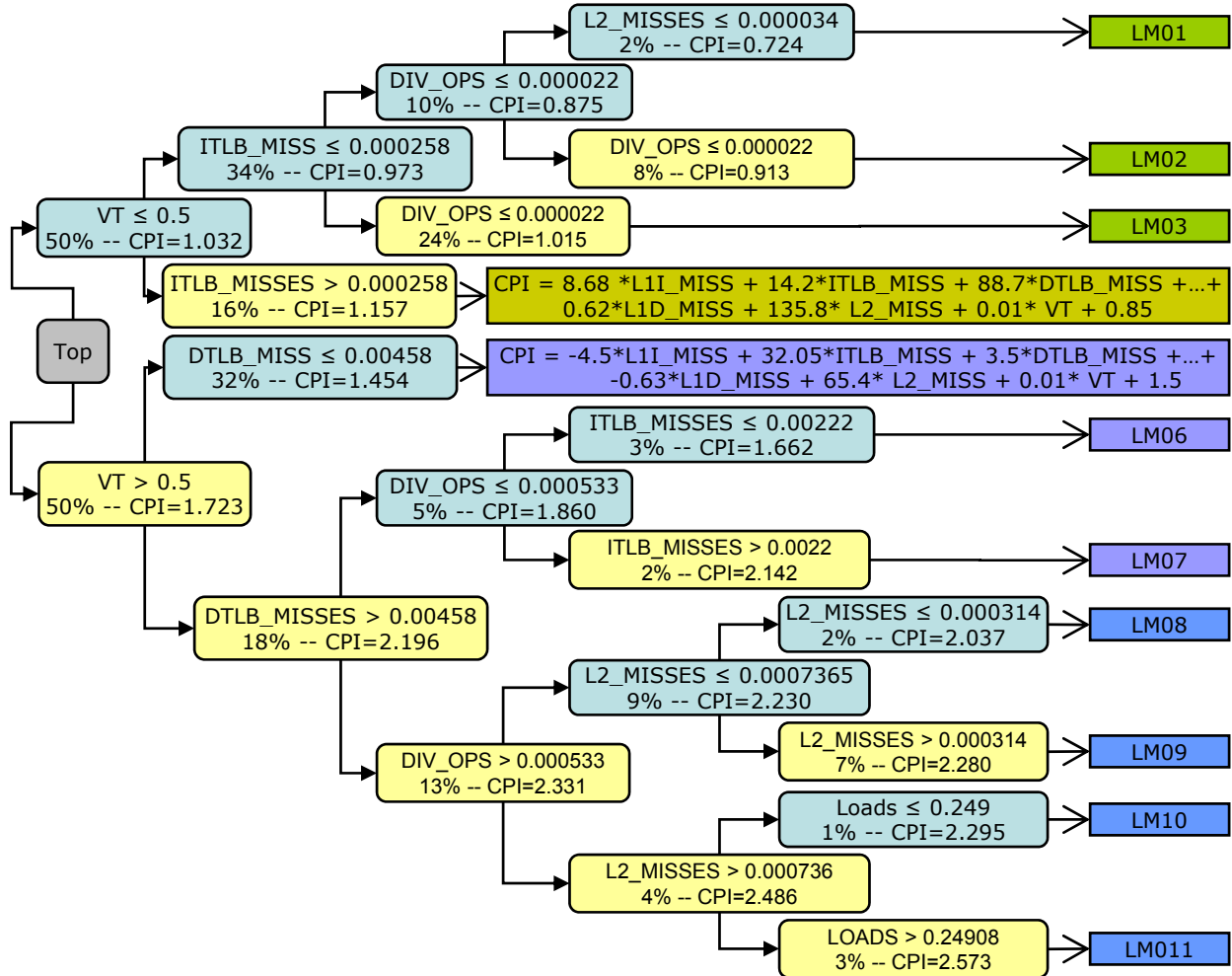


Fig. 4. Model Tree for the Linux Build. Two key Linear Models are shown. The remaining Linear Models are omitted for brevity

There are two main components to the generated Model Trees: the Model Tree itself and a set of linear equations at the leaf nodes of the tree. Figure 4 presents the tree for the Linux build benchmark. In this tree, the first split event in the tree is indeed the VT variable (VT \leq 0.5 simply indicates that virtualization was enabled). This means that the factor that has the most effects on the CPI for this workload is whether this was a virtualized execution. The fact that the model clearly splits the data into distinct groups also indicates that performance models for native and virtualized executions are inherently different

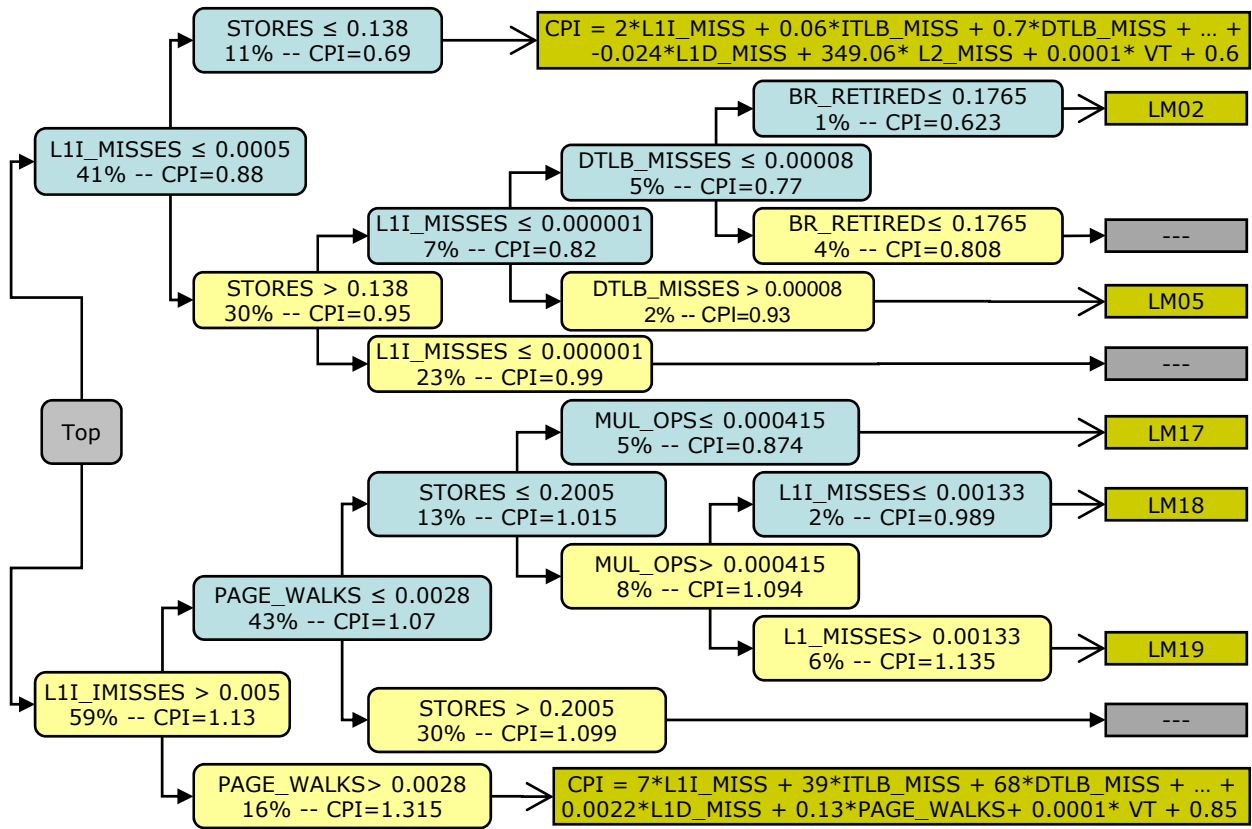


Fig. 5. Model Tree for the Dacapo Suite

even when evaluated primarily using micro-architectural events. This empirically confirms our research motivation; namely of the need for building new performance and projection models for VT. In the Model Tree, the linear equations can be found at the leaf level nodes labeled as LM01, LM02 ... etc. For reasons of space, the linear model equations have been left out of this omitted and only the average CPI and the representation within the data sample are included for each linear grouping.

Another use of this Model tree analysis is to provide an alternative way of estimating the average VT overhead: this can partly be done by determining the difference in the average CPI between the sub-tree on the left and the one on the right of the root node. Since the model only uses micro-architectural events, it can also be used to estimate how much of the overhead is attributable to the way virtualized execution exercises micro-architectural resources (explained using the variables in the sub-tree for virtualization) and how much can only be explained by including VT events. The later part could be viewed as dependent on VT architecture.

An additional insight gained from this technique is the ability of establishing the relative importance of an event to the overall performance of the workload. This can be done by observing the distance of an event from the tree's root. For example, in figure 4 ITLB_MISS was at the very top of the model tree suggesting that this value was a primary CPI predictor under Native condition. Under Virtualization this event occurs at level 3, and only in the decision path for about 10% of the VT subtree (5% of the entire data set). This suggests a decrease in the importance of ITLB misses in predicting the CPI for virtualized execution as opposed to Native execution. Another example is the increased significance of DTLB misses for Virtualized execution over native execution as evidenced by the appearance of the DTLB_MISS event at the very top of the VT subtree whereas it was not present in the Native subtree.

We also applied Model Trees to analyze to the Dacapo suite benchmark and resultant model is shown in Figure 5. Unlike the Linux benchmark, the VT dummy variable does not appear at the root of the tree. In fact, it does not show up as a split variable. Instead, it appears in all the linear equations with different coefficients in the different models. This suggests that virtualization is not a significant predictor of performance for this workload suite. However, like the Linux benchmark model, the Dacapo Model Tree indicates that virtualization is still associated with a statistical degradation - albeit of a smaller magnitude - in performance: that is, the coefficient in front of the VT dummy variable is strictly positive in all the linear models.

6. Related Work

Several previous studies have examined specific impacts of virtualization on system performance. [2] studies the network processing slowdown due to virtualization with Xen, while [8] examines the overheads from I/O processing in the hypervisor due to requests from specific virtual machines. [17] analyzes the cross interference of concurrently executing guests and [24] evaluates the impact of Xen based paravirtualization on the message passing interface and process execution in communication intensive High-Performance-Computing (HPC) systems.

Current studies in virtualization often rely on execution sampling utilities or extensions to native systems. Xenoprof is a Xen profiling utility based on the popular Oprofile system used for native UNIX based systems [18]. Xenoprof extends the profiling capabilities of Oprofile by distributing measured performance counter data to the corresponding guest systems. Xenoprof, through extended Oprofile utilities, then assigns performance events to specific code regions from the kernel and the application. Perfctr and PAPI [6] are two additional utilities commonly used for performance counter measurements and correlations. Neither currently supports virtualized platforms. Xentrace [23], Xentop [9], and XenMon [10] provide instrumentation implemented with the Xen system for counting hypervisor events and for tracing of hypervisor. We utilize Xentrace to collect hypervisor events in addition to the events we collect from the hardware PMC and the guest kernels. In addition to the facilities provided by these utilities, we also collect a synchronized vertical profile of the execution stream including the entire software and hardware stack. In this respect our approach is similar to the Vertical Profiling system described in [11] for the Jikes Java Virtual Machine.

As hardware and software systems become increasingly complex, statistical techniques for performance modeling [14, 15, 20, 22] have become invaluable design and optimization tools for gaining insights into performance critical factors. Several studies also propose performance models to reduce the number of simulations required to explore the design space of processors and other systems on chip. [22] presents a model for representing performance metrics as a linear function of micro-architectural events and uses it to improve the process of design space exploration for embedded processors. Recently more complex analytical models have been applied to benchmark performance analysis. [15] proposes a model based

on Artificial Neural Networks. [16] uses a non-linear model based on radial basis function networks. [19] compares five machine-learning regression algorithms applied to PMU data for a subset of SPEC CPU2006. Model trees were found to perform as well as artificial neural networks (ANNs) and support vector machines (SVMs) and had the advantage of interpretability. [20] describes the application of Model trees for characterizing benchmark performance.

Finally, [7] proposed a benchmarking suite for evaluating and comparing Virtual Machine Monitors. This study is an important step towards standardizing the measurement criteria for comparing hypervisor performance. The methodology and instrumentation approach we have advanced in this paper can aid exploring in detail the performance of virtual machine monitors and to assess the architectural and system level impacts of virtualization under such a benchmarking suite.

7. Conclusions and Future Work

We described an analysis approach that allows one to break down the cycles per instruction (CPI) performance metric for a workload using statistical regression. The specific challenge our approach addresses is that of obtaining the performance measurements at a fine granularity of sampling, such that factors of interest in software and hardware are counted in unison and at very low perturbation to the system. We demonstrated by example how such instrumentation allowed us to deconstruct the CPI metric for two different workloads, using two different regression algorithms: first, a multi-linear fit against performance data at the whole workload level, and next, a Model-tree characterization of the workloads using fine grained data gathered at the granularity of several million instructions. Our work is particularly useful in the context of execution in virtual machines because it removes the need to assume single state behaviors over long intervals of time in order to relate CPI to its factors.

This work is a first step on a path to building more complex and detailed predictive models for workload performance. Our next steps are to extend the instrumentation and the methodology so that resource utilization under homogenous and heterogeneous consolidation of workloads can be measured accurately and used in statistical prediction of overall workload performance.

References

1. K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.
2. P. Apparao, S. Makineni, and D. Newell. Characterization of network processing overheads in xen. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 2, Washington, DC, USA, 2006. IEEE Computer Society.
3. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.
4. S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.
5. L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.
6. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal for High Performance Computer Applications*, 14(3):189–204, 2000.
7. J. Casazza, M. Greenfield, and K. Shi. Redefining server performance characterization for virtualization benchmarking. *Intel Technology Journal*, AUG 2006.
8. L. Cherkasova and R. Gardner. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

9. J. Fischbach, D. Hendricks, and J. Triplett. Xenop. Xen builtin Utility, 2005.
10. D. Gupta, R. Gardner, and L. Cherkasova. Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, HP, OCT 2005.
11. M. Hauswirth, A. Diwan, P. Sweeney, and M. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 281–296, New York, NY, USA, 2005. ACM.
12. Novell Inc. Virtualization in the data center. Technical Report 462-002015-001, MAY 2006.
13. Intel Corporation. *Intel64 and IA-32 Architectures Software Developers Manual*, 2006.
14. E. İpek, S. McKee, R. Caruana, B. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, New York, NY, USA, 2006. ACM.
15. P. Joseph, K. Vaswani, and M. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.
16. B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 185–194, New York, NY, USA, 2006. ACM.
17. J. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, and M. McCabe. Quantifying the performance isolation properties of virtualization systems. In *ecs'07: Experimental computer science on Experimental computer science*, pages 5–5, Berkeley, CA, USA, 2007. USENIX Association.
18. G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, AUG 2006.
19. E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, and K. Doshi. On the comparison of regression algorithms for computer architecture performance analysis of software applications. *First Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilation*, pages 116–125, JAN 2007.
20. E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 116–125, 25-27 April 2007.
21. M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, MAY 2005.
22. L. Simonson and L. He. Micro-architecture performance estimation by formula. In Timo D. Hmlinen, Andy D. Pimentel, Jarmo Takala, and Stamatis Vassiliadis, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Lecture Notes in Computer Science, pages 192–201. Springer, 2005.
23. M. Williamson. Xentrace.
24. L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Evaluating the performance impact of xen on mpi and process execution for hpc systems. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 1, Washington, DC, USA, 2006. IEEE Computer Society.