Enhancing Virtualized Application Performance Through Dynamic Adaptive Paging Mode Selection

Chang S. Bae
Dept. of EECS
Northwestern University
Evanston, IL 60208
cbae@u.northwestern.edu

John R. Lange
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
jacklange@cs.pitt.edu

Peter A. Dinda
Dept. of EECS
Northwestern University
Evanston, IL 60208
pdinda@northwestern.edu

ABSTRACT

Virtual address translation in a virtual machine monitor (VMM) for modern x86 processors can be implemented using a software approach known as shadow paging or a hardware approach known as nested paging. Most VMMs, including our Palacios VMM, support both. Using a range of benchmark measurements, we show that which approach is preferable for achieving high application performance under virtualization is workload dependent, and that the performance differences between the two approaches can be substantial. We have developed an algorithm, based on measuring the TLB miss rate and the VMM exit rate for paging-related exits, for measuring the performance of the current paging approach during normal execution and predicting when switching approaches is likely to be beneficial to the application. We combine this with new support in Palacios for switching paging approaches at any time to implement a dynamic adaptive paging approach called $\mathrm{DAV}^2\mathrm{M}$ for dynamically adaptive virtualized virtual memory. $\mathrm{DAV}^2\mathrm{M}$ attempts to deliver application performance that is the same or better than the best static approach for a given workload. Our evaluation shows that it does so, and, for a range of benchmarks, $\mathrm{DAV}^2\mathrm{M}$ delivers performance within 5–8% of native, reducing overheads by as much as a factor of four. Although both the shadow and nested paging approaches are evolving, the dynamic adaptive approach can combine their best qualities.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Experimentation, Measurement, Performance

Keywords

virtual machine monitors, operating systems, performance analysis

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-0709168, and the Department of Energy (DOE) via grant DE-SC0005343.

1. INTRODUCTION

A critical component of the overhead of a modern virtual machine monitor (VMM) for x86 or x64 hardware is the virtualization of address translation. Conceptually, a VMM introduces an additional layer of indirection that maps addresses that the guest OS believes are physical addresses, but called here guest physical addresses (GPAs), to the actual host physical addresses (HPAs). Address translation then is effectively from guest virtual addresses (GVAs) to GPAs and then to HPAs.

There are two general approaches to achieving this translation, shadow paging and nested paging. Both approaches have several variants. Shadow paging is a software-centric approach that flattens GVA→GPA→HPA translation to GVA→HPA translation and implements this translation in a single page table hierarchy that is under the exclusive control of the VMM and combines guest and VMM intent. In contrast, nested paging is a hardware-centric approach in which the hardware provides for a second page table hierarchy that the VMM can use to separately maintain the GPA→HPA mapping without having to involve itself in the guest's GVA→GPA decisions. Section 2 describes these approaches, and their variants on the x86/x64 in more detail.

Although both approaches can achieve very low overhead, we argue that there is not a single best approach for minimizing overheads and hence maximizing application performance. Rather, the best approach depends on the paging workload of the application running in the VM. Further, even for a single application, the best approach may not be static, but may rather vary over time. For example, an HPC application with multiple computation phases might have a preferred paging approach for each phase. As another example, a long-running VM might very well have different applications execute during its lifetime, each of which may prefer a different paging approach. In Sections 3 and 4 we report on a study, based largely on HPC workloads, that supports these claims. The virtualized application performance, relative to a native environment, can vary by as much as 105% between the two modes.

Most modern VMMs for x86/x64 hardware, including the Palacios VMM [19] in which this work is implemented, offer several variants of both shadow and nested paging. In these VMMs, the selection of paging approach is made at the time the VM is instantiated and holds for the VM's lifetime. As we explain in Section 5, we have extended Palacios to allow us to dynamically change the paging approach at *run-time*. As part of the handling of any exit to the VMM, we can switch to a different approach.

We have developed a prototype policy that uses this mechanism to enhance the performance of applications as they execute. The policy is based on two metrics of paging performance. When shadow paging is in use, the metric is the rate of VMM exits related to paging, which is extremely easy to measure in the context of exit handling. When nested paging is used, the metric is the TLB miss rate, which is measured at virtually zero cost using a hardware performance counter. A third metric, cycles per instruction (CPI), is used to measure application performance. These metrics are further described in Section 4. The metrics are valuable apart from our policy since they succinctly capture the performance of each paging approach and the effect on application performance.

Our prototype policy, dynamically adaptive virtualized virtual memory ($\mathrm{DAV^2M}$), uses our mechanism and metrics to probe application performance using the current and alternate paging modes. Probing is triggered when the paging performance under the current mode exceeds a threshold. Based on this probing, the thresholds associated with each paging approach are also adjusted. Performance testing takes into account transient effects due to the paging approach switch itself, and probes are temporally limited to avoid potential oscillations in this control algorithm. When probing identifies a clearly superior paging approach, it is made the current one. $\mathrm{DAV^2M}$ is described in detail in Section 6.

We implemented $\mathrm{DAV^2M}$ in Palacios and evaluated it using the application benchmarks described in Section 3 that are most sensitive to particular paging approaches, including benchmarks that are insensitive for comparison. Detailed results are shown in Section 7. The most salient points are as follows. First, for workloads which have a single best paging approach, $\mathrm{DAV^2M}$ is able to quickly converge on that approach. Because of this quick convergence and the reasonably low overhead of the mechanism for switching the paging approach, the performance of these workloads under $\mathrm{DAV^2M}$ is nearly identical to that we would have seen if we chose the paging approach correctly when the VM was configured ($\leq 1\%$). A second important result is that, for a benchmark whose best paging approach varies over its execution, $\mathrm{DAV^2M}$ is able to dynamically change the paging approach to match changing circumstances, without oscillatory behavior.

The contributions of our paper are as follows.

- We demonstrate, based on a study of a range of application benchmarks, that there is no single best paging approach in VMM that will maximize application performance. The choice of approach can have a significant impact.
- We describe VMM-based metrics that cheaply and succinctly measure paging and application performance.
- We describe the design, implementation, and evaluation of a new mechanism in our Palacios VMM that allows for dynamically changing the paging approach during run-time.
- We describe the design, and implementation of a policy, $\mathrm{DAV^2M}$, that drives our mechanism to dynamically select the best paging approach as a VM executes.
- We provide a detailed evaluation of $\mathrm{DAV^2M}$ in Palacios, showing that it is highly effective, using the same application benchmarks.

Our work is publicly available in the context of the Palacios VMM codebase, which can be found at v3vee.org.

2. PAGING APPROACHES

In a virtualized environment, paging is complicated because there are essentially two levels of address translation. Conceptually, the guest OS controls the translation from guest virtual addresses (GVAs) to guest physical addresses (GPAs) by manipulating page tables in its address space. The VMM controls the translation from GPAs to host physical address (HPAs) by manipulating some other structure that implements the mapping. The two structural forms we consider here are shadow paging and nested paging.

Shadow paging. Shadow paging is a form of virtualized paging that is implemented in software. In order to understand shadow paging, it is helpful to differentiate the privilege level of the guest page tables and the VMM page tables. Because the VMM runs at a higher privilege level, it has the ultimate control over the control registers used to control the normal paging hardware on the machine. Because of this, it can always ensure that the page tables in use contain the correct mapping of guest addresses to host addresses. These page tables, the shadow page tables, contain mappings that integrate the requirements of the guest and the VMM. The shadow page tables implement a mapping from GVA to HPA and are in use whenever the guest is running.

The VMM must maintain the shadow page tables' coherence with the guest's page tables. A common approach to do so is known as the virtual TLB model [16, Chapter 28]. The x86's architected support for native paging requires that the OS (guest OS) explicitly invalidate virtual address (GVAs) from the TLB and other page structure caches when corresponding entries change in the in-memory page tables. These operations (including INVLPG and INVLPGWB instructions, CR3 (the pointer to the current page table) writes, CR4.PGE writes, and others) are intercepted by the VMM and used to update the shadow page tables. The interception of guest paging operations can be expensive as each one requires at least one exit from the guest, an appropriate manipulation of the shadow page table, and one reentry into the guest. These operations are especially expensive when context switches are frequent. A typical exit/entry pair, using hardware virtualization support, requires in excess of 1000 cycles on typical AMD or Intel hardware.

In this paper, we use an implementation of shadow paging with caching, as is common in many VMMs. Shadow paging with caching attempts to reuse shadow page tables. Ideally, the reuse is sufficiently high enough that a context switch can be achieved essentially with only one exit, to change the CR3 value. The VMM maintains in memory both shadow page tables corresponding to the current context, and shadow page tables corresponding to other contexts. The distinction is not perfect—it is perfectly legitimate for the guest to share guest page tables among multiple contexts, or even at different levels of the same context. Furthermore, the guest kernel has complete access to all of the guest page tables, for all contexts, at any time.

Nested paging. Nested paging is a hardware mechanism that attempts to avoid the overhead of the exit/entry pairs needed to implement shadow paging by making the GVA$\rightarrow$GPA and GPA$\rightarrow$HPA mappings explicit and separating the concerns of their control, making it possible to avoid VMM intervention except when a GPA$\rightarrow$HPA change is desired. Both AMD and Intel support nested paging.

In nested paging, the guest page tables are used in the translation process to reflect the GVA$\rightarrow$GPA mapping, while a second set of page tables, visible only to the VMM, are used to reflect the GPA$\rightarrow$HPA mapping. Both the guest and the VMM have their own copy of the control registers, such as CR3. When a guest tries to reference memory using a GVA and there is a miss in the TLB, the hardware page-walker mechanism performs a two dimensional traversal using the guest and nested page tables to translate the GVA to HPA. When the page walk completes, the result is that the translation is cached in the TLB.

It is important to realize that with nested paging every step of the page walk in the guest's page tables requires a traversal of the nested page tables—the guest and nested page walk lengths do not add, they *multiply*. The consequence is that a TLB miss can be very expensive to handle. However, hardware page walk caching has been extended to ameliorate this, and the VMM can further

CPU	Opteron 2350 2 GHz
Cache	L1 DCache (2-way): 64KB
	L1 ICache (2-way): 64KB
	L2 (16-way): 512KB
	L3 (32-way): 2MB
TLB entries	L1 DTLB (full): 48 (4K, 2M, 1G)
(page size)	L2 DTLB: 512 (4K) 4-way,
	128 (2M) 2-way, 16 (1G) 8-way
	L1 ITLB (full): 32 (4K), 16 (2M)
	L2 ITLB (4-way): 512 (4K)
BUS	1 GHz
Memory	2GB 667 MHz (DDR2)

Figure 1: Features of primary test machine.

ameliorate it by using large pages (short walks) in the nested page tables. In this paper, we use the AMD nested paging implementation on the Opteron 2350. Palacios can use large pages for nested page tables.

Comparison. A TLB miss under nested paging potentially incurs a very high cost compared to shadow paging because of the two dimensional page walk that is needed. In the worst case, using 4-level page tables in the guest and VMM, the cost for a TLB miss is 24 memory references. In contrast, the cost of a TLB miss for shadow paging is the same as that of the native case. In the extreme where there is little locality of reference, it is likely that nested paging will underperform shadow paging.

On the other hand, a shadow page fault is very expensive due to the necessary involvement of the VMM. This is primarily because the cost of VM exit is quite high, as previously noted. Furthermore, a guest page fault will often produce a pair of exits: the first to inject the page fault into the guest, and the second to fill the shadow page table entry appropriately. This leads to a situation in which a guest that frequently modifies its page tables will perform better using a nested rather than shadow paging approach.

3. WORKLOADS

We have previously conducted studies on virtualized paging using a range of benchmarks [8, 15]. It is important to note that for many workloads, the paging approach makes little difference to performance. In this paper, we focus on specific benchmarks that highlight the differences in performance between the two approaches. These are typically benchmarks that stress the TLB.

We surveyed widely-used benchmarks such as SPEC CPU (2000 and 2006) [3] and PARSEC [13, 12] under a native environment, and found TLB-intensive workloads as shown along the x-axis of Figure 2. Our measurements were done on a Dell PowerEdge SC1435 described in Figure 1. We ran the benchmarks under Linux 2.6.27 (64-bit). The PARSEC benchmarks used the distributed precompiled binaries. GCC 4.3.2 was used to compile SPEC CPU. We used Oprofile 0.9.4 for measuring TLB misses and clock cycle counts, PinPoints (Pin 2.8/SimPoint 3.2) [1] to profile memory access patterns, and libhugetlbfs 2.4 [14] to examine the effect of large pages.

Large page effects. Given this set of TLB-sensitive benchmarks, we also considered the effect of the use of large pages. One expectation is that large pages will reduce the TLB miss rate simply by reducing contention. A second expectation is that because large pages imply fewer levels on the page hierarchy, we will decrease

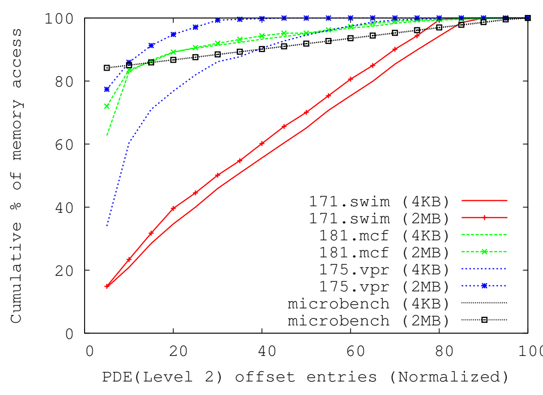


Figure 3: Locality of reference analysis of benchmarks. The memory access pattern of $171.swim$ is so random that TLB misses are frequent, even when large pages are used. However, ∼20 % of the page table covers >80% of memory accesses without large pages ($181.mcf$) or with large pages ($175.vpr$).

the number of page table entries, making it more likely that entries will appear in the data cache. Many of the benchmarks showed increased performance (by 10–20%) using large pages. However, there are also cases, $181.mcf$ and $436.cactusADM$, where just the opposite occurs.

Locality in 2-level page entries. To better understand why there are benchmarks where there is a high TLB miss rate even when using large pages, we considered $171.swim$, $175.vpr$, $181.mcf$, and a "worst case" microbenchmark that scans pages in a manner designed to maximize the TLB miss rate by forcing misses at every level of the page hierarchy. In Figure 3, we present the degree of locality that holds in the first level ("PDE") of the page hierarchies for these benchmarks. Two benchmarks have more locality than the worst case. However, it is important to note that $171.swim$ shows a very random memory access pattern with a large working set, which helps to explain why large pages do not enhance its performance significantly.

4. BEHAVIOR AND METRICS

We now consider the selection of metrics for quickly measuring the performance of shadow and nested paging.

Focused benchmark set. Workloads that are TLB-intensive will produce the largest differences between the performance of shadow and nested paging. We therefore have selected the following benchmarks from Figure 2 for further study: $171.swim$, $301.apsi$, $434.zeusmp$, $403.gcc$, and $164.gzip$. For comparison, we add to this set $191.fma3d$ and $186.crafty$, which are much less TLB-intensive. $403.gcc$ and $164.gzip$ use multiple sequential inputs, and thus are likely to show phase behavior in page translation. We might expect that different inputs will result in different page mappings. Hence, we expect these benchmarks to stress shadow paging, requiring many exits to the VMM to repair the shadow page tables when phase changes occur.

Palacios VMM. The shadow and nested paging implementations we use are implemented in the Palacios VMM. Palacios is is an OS-independent, open source, BSD-licensed, publicly available type-I VMM designed as part of the V3VEE project (`http:`

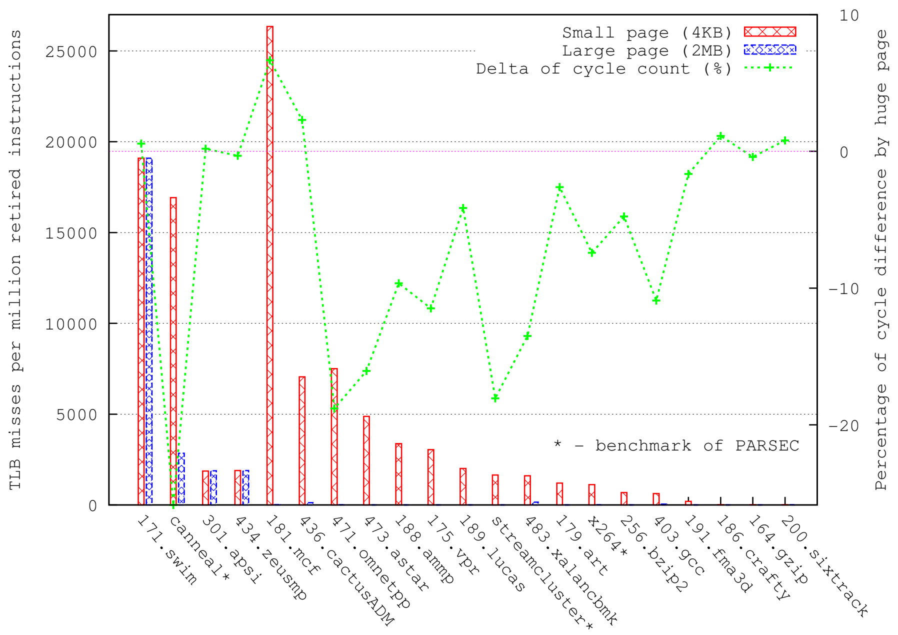


Figure 2: TLB-intensive workloads surveyed for this paper and their native performance with large and small pages. The bars indicate TLB misses (smaller is better), while the line shows the percentage performance difference between large and small pages for each benchmark (smaller is better).

`//v3vee.org`). For this work, we use Palacios embedded in the Kitten lightweight kernel. Detailed information about Palacios and Kitten can be found elsewhere [19] with code available from our site. We specifically use these commits: Palacios commit 1cd2958b5eb63b2ac63ced17447ba3b45c43f51a and Kitten commit 738:02e673de9a2e. The machine used is as described in the previous section.

Guest OS. We run the benchmarks on a guest (and native) OS based on Puppy Linux 3.01 [2], with a 2.6.18 Linux kernel, running on a single core 32 bit guest environment.

Conservative shadow paging performance. Although we employ shadow paging with caching, the implementation we evaluate in this paper is likely to produce conservative performance compared to nested paging for several reasons:

- 32 bit guest addressing. 32 bit guest addressing results in the guest page tables being at most 2 levels deep. This means that a nested page walk is much shorter than if 64 bit addressing (4 levels) were used in the guest.
- Small pages. Our shadow paging implementation uses small pages, while our nested paging implementation can use large pages at the nested level. As described in Section 3, using large pages provides an opportunity to reduce TLB contention, thus favoring nested paging. However, it is important to note that benchmarks such as 171.$swim$, 301.$apsi$, and 434.$zeusmp$ are relatively insensitive to the choice of large or small pages.
- TLB tags [6, Chapter 12] and VMCB caching [5, Chapter 15.15] are not used. These features benefit shadow paging more than nested paging as they reduce exit costs.

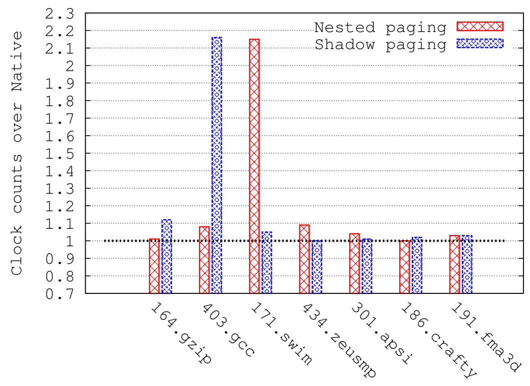


Figure 4: Performance, compared to native, of selected benchmarks using nested and shadow paging. Lower numbers are better. Neither approach is consistently superior.

Performance comparison. Figure 4 shows the performance, compared to native, of nested and shadow paging, for the selected benchmarks. As we can see, neither approach is consistently superior. We seek to automatically and dynamically choose the approach that gives the best performance.

Metrics. We now consider the selection of metrics for measuring application performance and the performance of the current paging approach. Note that application performance may decline for reasons unrelated to paging, so it is essential to measure it independently. The metrics must be very inexpensive to measure. We have selected the following metrics:

- Application performance: Cycles per instruction (CPI), measured as the number of CPU cycles needed to execute a

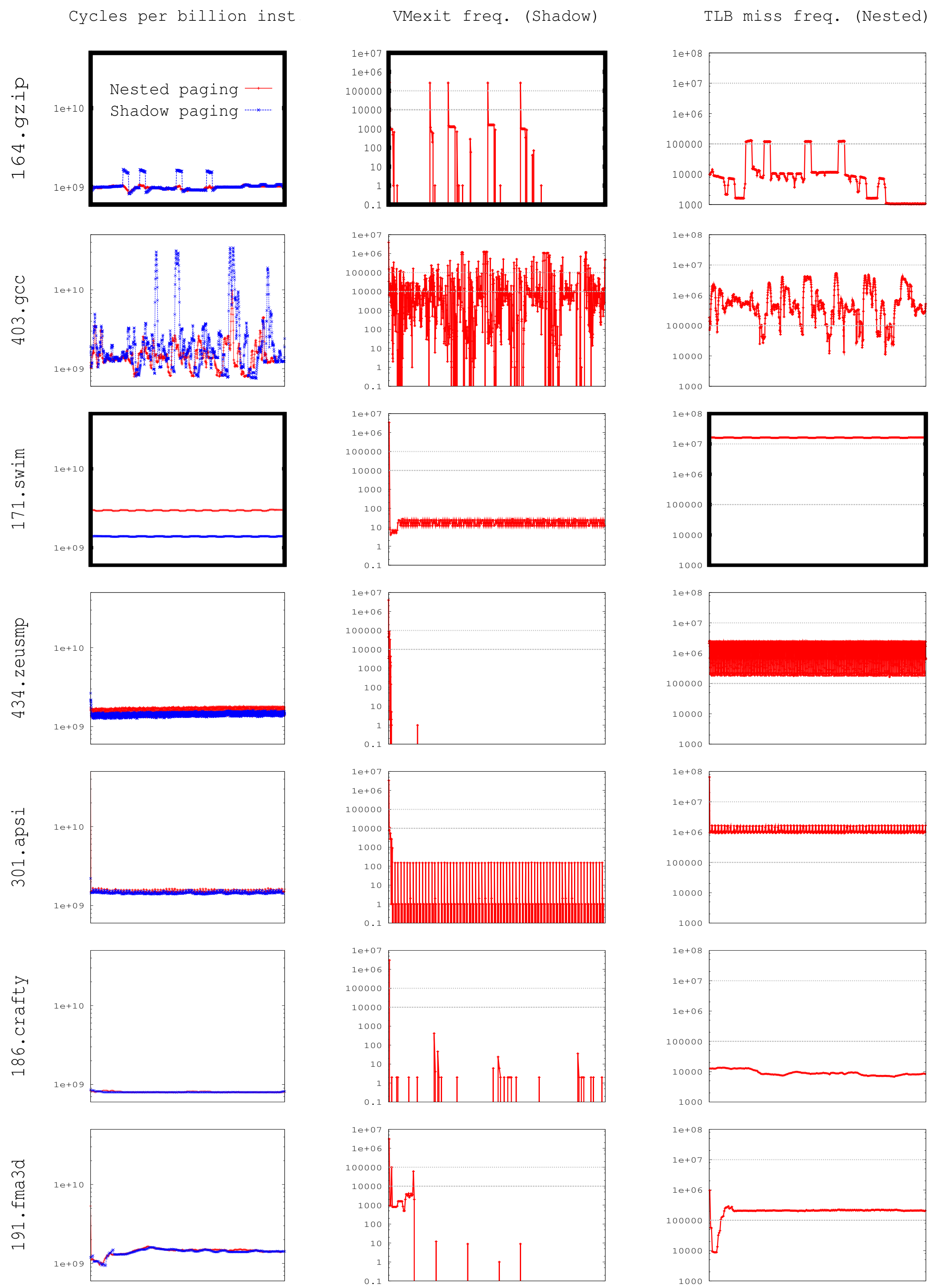


Figure 5: Deeper analysis of nested and shadow paging performance, and metrics. Each row is a single benchmark. The first column shows the performance, using CPI, of the benchmark. The second column shows the paging-related VM exit rate when using shadow paging. The third column shows the TLB miss rate when using nested paging. Highlighted graphs are described in Section 4.

window of instructions. We smooth the CPI with a 10 step moving average.

- Nested paging performance: TLB miss rate, measured over a window of instructions. We smooth the TLB miss rate with a 10 step moving average.
- Shadow paging performance: VM exit rate due to paging, measured over a window of instructions.

To capture the CPI and TLB miss rate, we make use of the hardware performance counter unit (PMU) [7, Chapter10]. The counters used are the cycles outside of halt states, the count of retired instructions, the count of L1 DTLB misses, and the count of L2 DTLB misses. The first two are combined to create the application performance metric (CPI), and the last is used for the nested paging performance metric. VM exits related to paging are counted in Palacios's exit handler. Instructions and TLB misses are counted in the context of the guest, but the cycle count is measured under both VMM and guest context. Because of this, we are measuring these metrics, as they affect the guest, over "wall clock" time (not virtual time). Measurements are made in the context of VM exits that are already occurring, with the window and sampling interval chosen so that there is $< 1\%$ overhead.

Figure 5 provides a detailed view of how these metrics vary over the execution of the selected benchmarks. The first column shows the CPI over time for both shadow and nested paging for easy comparison. The second shows the paging-related VM exit rate for shadow paging, while the third shows the TLB miss rate for nested paging. For all graphs, lower is better. The thresholds shown in the graphs in the middle and right-hand columns will be explained in Section 6. The highlighted graphs are explained next.

Comparing the left column of Figure 5 and Figure 4 we can see that our CPI measurement does indeed capture the application-level difference in performance between shadow and nested paging. Furthermore, we can see a clear connection between low application performance (high CPI) and shadow paging performance. For example, for $164.gzip$, the four bursts of low CPI (left graph) are explained by the four bursts of high paging-related VM exit rate (middle graph). Similarly, the better performance of $171.swim$ under shadow paging is captured by its correspondingly lower CPI (left graph), and the high TLB miss rate (right graph) explains why nested paging has difficulty performing well.

5. MECHANISM

We have implemented a facility in Palacios that allows us to switch between shadow paging and nested paging in the middle of handling any exit. There are essentially two elements to this mechanism, (a) management of the paging and TLB-related aspects of the hardware-specific virtualization extensions, particularly the VMCB [5] or VMCS [16], and (b) management of the paging state for each mechanism in a manner such the guest and VMM intent represented in one can readily be translated to the other.

Palacios maintains a relatively stable GPA→HPA mapping. Because of this, maintaining nested paging state is quite straightforward. In essence, unless the guest physical memory map changes, the nested page tables can simply be kept cached. Thus, when we switch from shadow to nested paging, we can simply reuse them. In contrast, the shadow page tables include guest intent, which is not under our control. If we tracked guest page table updates while using nested paging, we would obviate the performance benefits of nested paging. Instead, we simply flush the shadow page tables when we switch from nested paging to shadow paging. Notice that shadow page caching is still used while we are running with shadow paging. It is only when we transition from shadow paging

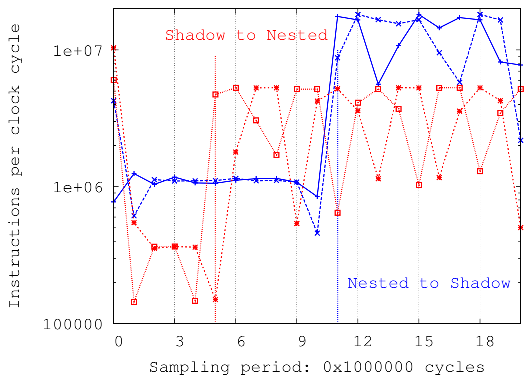


Figure 6: Overhead for shadow→nested and nested→shadow transitions for $403.gcc$.

to nested paging, and then return to shadow paging that we lose the shadow paging cache contents.

Because of the asymmetry in tracking and reconstructing paging state, it is considerably cheaper to switch from shadow paging to nested paging than the reverse, all other things being the same. Of course, the actual switching cost also depends on the workload. Figure 6 shows both costs in the context of the $403.gcc$ benchmark, where they are among the highest. Shadow to nested requires about half of the time of the reverse in this benchmark. The real time for the most costly switch, including the time for the transient effects to quiesce, on this most costly benchmark is $\leq$ 100ms. On other workloads, the transitions occur in much less time.

6. POLICY

DAV^{2}M is a threshold-based policy. The performance of the current paging mode is compared with a threshold. If the threshold is exceeded, we switch to the alternative mode. A naive approach to setting these thresholds might be to make them fixed. However, as we can see from Figure 5, our metrics vary widely across workloads, and also across time within an individual workload. Furthermore, even if a threshold were correct for a workload, fixing it would make possible oscillatory behavior, bouncing between the alternative modes.

In Figure 5, for the graphs in the middle and right-hand columns, the horizontal lines represent potential static thresholds for illustration. Clearly, a workload such as $171.swim$ would be well served by these thresholds. However, consider $403.gcc$. For this workload, the illustrative thresholds would result in continuous switching that would kill performance. Consider $434.zeusmp$. For this workload, the thresholds would lead to a consistent selection of nested paging, which would in a 8% increase in execution time compared to using shadow paging.

DAV^{2}M uses dynamic thresholds that are adjusted whenever we switch modes. The adjustments to the thresholds are based on the performance difference that is seen, as measured using CPI. If performance increases due to the switch, the threshold is adjusted so that switch will occur at a lower threshold. Otherwise, the threshold makes the switch less probable in the future.

States. DAV^{2}M uses five states, as shown in Figure 7, and described in the following.

- **Shadow** is the state when we are operating under shadow paging. The VM exits related to paging are being counted in $count_{vexit}$. When we retire $window_{vexit}$ instructions, the

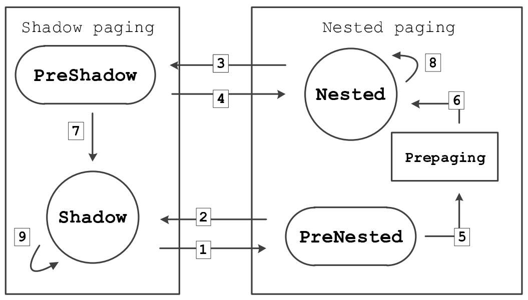


Figure 7: State transition diagram.

counter is reset. When it is necessary to leave the state due to $count_{vexit}$ exceeding its threshold, $threshold_{vexit}$, the counter and cpi_{shadow}, computed from the number of retired instructions and the number of cycles during $window_{vexit}$ are stored.

- **PreNested** is the state in which we are probing the performance of nested paging after deciding that shadow paging performance is insufficient. Specifically, we compare the previous cpi_{shadow} with the current cpi_{nested}. Nested paging is being used. PreNested has a limited duration that expires when at least $window_{vexit}$ instructions have been retired.
- **Prepaging** is the state that is in force after the PreNested and before Nested. The purpose of this state, during which nested paging is also in force, is to allow the TLB miss rate behavior to quiesce before switching to Nested. Without this state, we could easily misinterpret the high TLB miss rate after a shadow-to-nested switch and switch back to shadow. The workloads $164.gzip$ and $403.gcc$ are examples where such oscillation would occur.
- **Nested** is the state in which we are operating under nested paging. Here we are counting TLB misses in $count_{tmiss}$, reseting every $window_{tmiss}$ instructions. When this count exceeds $threshold_{tmiss}$, we will leave the state, but update cpi_{nested} before doing so.
- **PreShadow** is the state in which we are comparing the previous cpi_{nested} with the current cpi_{shadow}, while running with shadow paging active. Like PreNested, PreShadow holds for a limited duration. As in Shadow, VM exits are being monitored.

Because the measurement granularity is courser when operating with shadow paging (due to operating over exits), the transient effects of switching from Nested to PreShadow are generally over by the time the PreShadow period is over. Thus no state analogous to PrePaging is needed.

Figure 8 illustrates an example timing of state transitions. Transitions are labeled by number as in the state diagram.The figure begins in the Shadow state purely for convenience. The time is in retired instructions.

To avoid repeated switching between nested and shadow paging, we modify the thresholds as we transition from state to state. Furthermore, in testing thresholds we multiply by a factor $pFactor$. Finally, we consider the intervals between transitions from nested to shadow ($count_{n2s}$) and shadow to nested ($count_{s2n}$) and compare to a window threshold $window_{trans}$. When a transition occurs within this window, both $threshold_{vexit}$ and $threshold_{tmiss}$ are increased, which will damp the system.

Algorithm specifics. DAV^{2}M advances through handling the following two events:

```
VM exit for a shadow page fault:
  $count_{vexit} \leftarrow count_{vexit} + 1$
  $determineState_{next}(state_{cur}, state_{next})$
  $transState(state_{cur}, state_{next})$
VM exit for a PMU overflow:
  $count_{inst} \leftarrow 0$
  $count_{punit} \leftarrow count_{punit} + 1$
  $count_{s2n} \leftarrow count_{s2n} + 1$
  $count_{n2s} \leftarrow count_{n2s} + 1$
  $determineState_{next}(state_{cur}, state_{next})$
  $transState(state_{cur}, state_{next})$
```

The PMU is set for $window_{inst}$ instructions as the as sampling period, as Section 5. For every retired instruction beyond this, an overflow occurs, causing a VM exit. These exits are counted in $count_{punit}$.
$transState(state_{cur}, state_{next})$ transitions to the next state and updates the thresholds. It is implemented as:

```
if $state_{cur} \neq state_{next}$ then
  if transit from PreNested to Shadow or
    from Nested to PreShadow then
    switch paging mode
    if $count_{n2s} < window_{trans}$ then
      increase $threshold_{tmiss}$ and $threshold_{vexit}$
      $count_{n2s} \leftarrow 0$
    end if
  else if transit from Shadow to PreNested or
    from PreShadow to Nested then
    switch paging mode
    if $count_{s2n} < window_{trans}$ then
      increase $threshold_{tmiss}$ and $threshold_{vexit}$
      $count_{s2n} \leftarrow 0$
    end if
  end if
end if
```

$determineState_{next}(state_{cur}, state_{next})$ determines the next state. It is implemented as:

```
$state_{next} \leftarrow state_{cur}$
if $state_{cur}$ is PreShadow or Shadow then
  if $count_{vexit} > threshold_{vexit}$ then
    update $cpi_{shadow}$
    $state_{next} \leftarrow$ PreNested
    $count_{punit} \leftarrow 0$
  else if $state_{cur}$ is PreShadow then
    if $count_{punit} = window_{tmiss}$ then
      update $cpi_{shadow}$
      if $cpi_{shadow} > cpi_{nested} * pFactor$ then
        $state_{next} \leftarrow$ Nested
        $count_{tmiss} \leftarrow 0$
        $count_{punit} \leftarrow 0$
        increment $threshold_{tmiss}$
      end if
    else if not ($count_{punit}$ mod $count_{vexit}$) then
      $count_{vexit} \leftarrow 0$
    end if
  else if $state_{cur}$ is Shadow and
    $count_{punit} = window_{vexit}$ then
    $count_{vexit} \leftarrow 0$
    $count_{punit} \leftarrow 0$
```

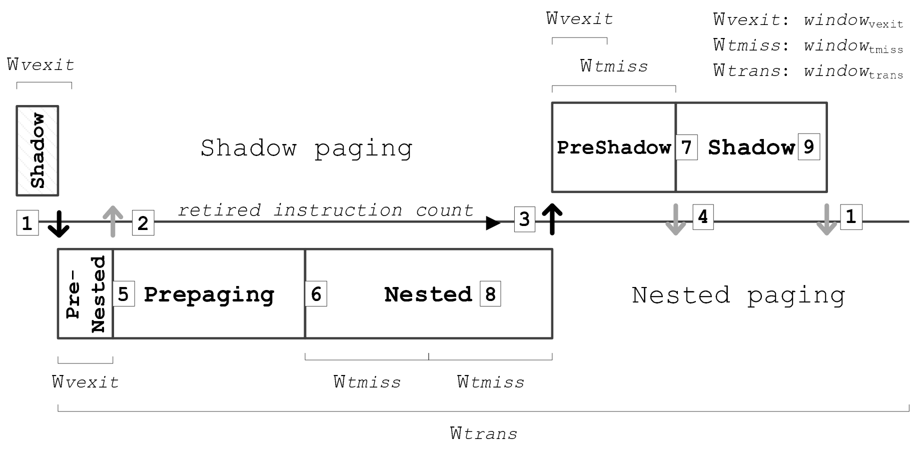


Figure 8: State transition timeline.

```
        end if
    else if $state_{cur}$ is either PreNested, Prepaging or Nested then
        if $state_{cur}$ is PreNested and
            $count_{punit} = window_{vexit}$ then
            update $cpi_{nested}$
            if $cpi_{nested} > cpi_{shadow} * pFactor$ then
                increment $threshold_{vexit}$
                $state_{next} \leftarrow$ Shadow
            else
                $state_{next} \leftarrow$ Prepaging
            end if
        else if $state_{cur}$ is Nested and
            $count_{punit} = window_{tmiss}$ then
            update $count_{tmiss}$
            if $count_{tmiss} > threshold_{tmiss}$ then
                update $cpi_{nested}$
                $state_{next} \leftarrow$ PreShadow
            else
                $count_{punit} \leftarrow 0$
            end if
        else if $state_{cur}$ is Prepaging and
            $count_{punit} = window_{prepaging}$ then
            $count_{punit} \leftarrow 0$
            $state_{next} \leftarrow$ Nested
        end if
    end if
```

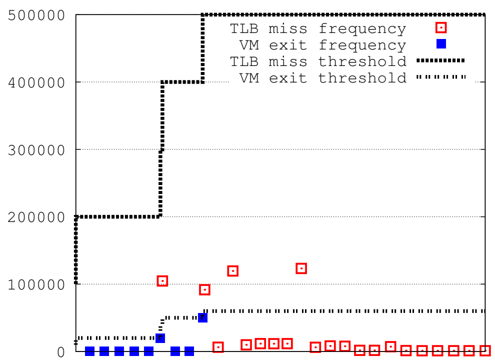
Several different windows have been introduced. Their relationships must be

$$window_{inst} \le window_{vexit} < window_{tmiss} < window_{trans}.$$

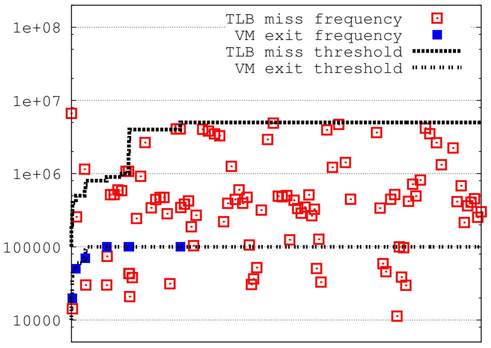
Threshold adjustment in action. Figure 9 shows how the thresholds are dynamically adjusted during the execution of two different benchmarks that would otherwise likely cause oscillation. The graphs show the execution of the 168.*gzip* and 403.*gcc* benchmarks, using the parameters described in Section 7.

7. RESULTS

We now present an evaluation of DAV^{2}M as implemented in Palacios, using the selected benchmarks described in Section 4.



(a) 168.*gzip*



(b) 403.*gcc*

Figure 9: Examples of DAV^{2}M's threshold control in action.

Parameters and initial settings. The parameters and starting values for DAV^{2}M, described in Section 6, were set as follows:

- $pFactor = 1.1$
- $window_{inst} = 10^9$ instructions
- $window_{vexit} = 10^9$ instructions
- $window_{tmiss} = 10 \times window_{inst}$
- $window_{trans} = 100 \times window_{inst}$
- $threshold_{vexit} = 10^4$
- $threshold_{tmiss} = 10^5$

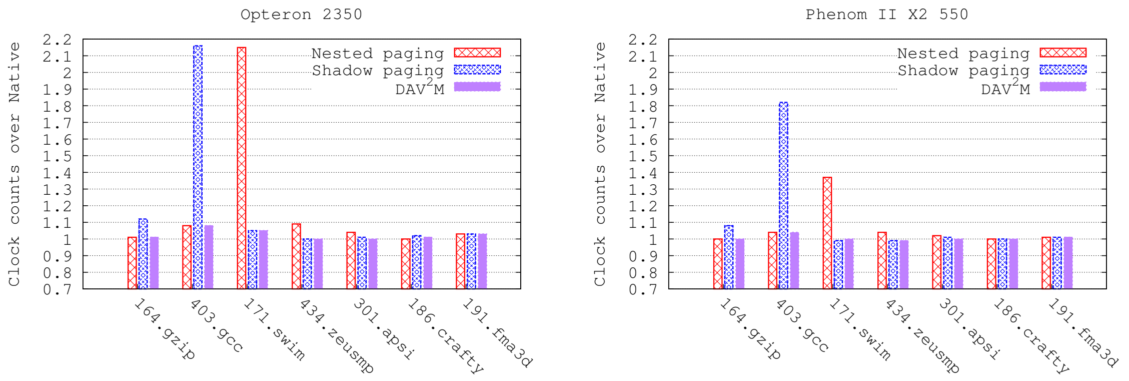


Figure 10: Performance of DAV^{2}M on two different processors. DAV^{2}M is able to provide virtually identical performance to the best static policy for every benchmark on both machines.

Application performance. Figure 10 presents the application performance results comparing DAV^{2}M with the static approaches of either shadow paging or nested paging. The format of each graph is identical to that of Figure 4. The bars compare to native performance, lower bars are better. The left hand graph presents an evaluation on the same Opteron 2350 as previously described, while the right hand graph presents the same evaluation done on a newer machine. The newer machine is equipped with with an AMD Phenom II X2 550 processor, 4GB RAM, 3GHz clock speed, and 6 MB of L3 cache. The most important observation is that DAV^{2}M is able to provide virtually identical performance to the best static paging approach on all of the benchmarks on both machines.

There are two important things to point out at this point. First, the measurements given in Figure 10 (and Figure 4) are of the number of cycles needed to run the benchmark—they reflect the total execution times of the benchmarks. These should not be confused with the CPI metric (Section 4) that DAV^{2}M uses internally to heuristically determine application execution rate. Secondly, recall that our evaluation focused on a set of benchmarks that induced the most significant differences between the two paging approaches (Section 3). For benchmarks where there is little difference (two are included), DAV^{2}M correctly does not affect performance.

Deeper analysis. We now focus on the results for the Opteron 2350 machine and illustrate how DAV^{2}M is working, and what its overheads are.

It is possible to group the benchmarks into three sets based on which virtual paging mode is best for performance:

1 171.$swim$ and 434.$zeusmp$ are best under shadow paging.

2 403.gcc and 168.$gzip$ are best under nested paging.

3 301.$apsi$, 186.$crafty$ and 191.$fma3d$ perform similarly.

For (1) and (2), DAV^{2}M quickly chooses the right approach. For (3), DAV^{2}M quickly chooses *an* approach and avoids switching.

Figure 11 illustrates the number of switches of paging mode that occur for each benchmark. Even in 403.gcc, only 13 transitions occur. If we consider 403.gcc's behavior in Figure 5, we can see why this benchmark might cause a larger number of transitions: there is phase behavior in which short phases where shadow paging is preferable occur. Note that despite this switching, 403.gcc under DAV^{2}M performs as well as the best static policy: the costs of switching are counterbalanced by the increased performance in those phases. For the other benchmarks, we can see very little switching, as we would hope for.

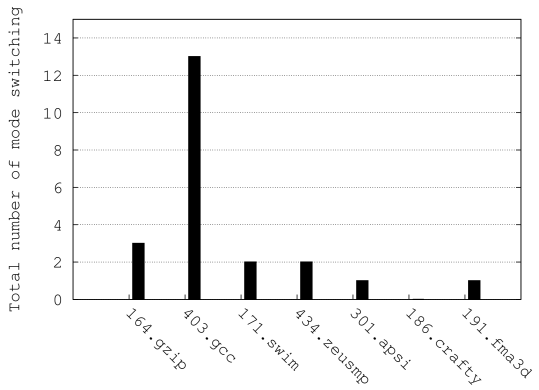


Figure 11: Number of transitions seen during execution.

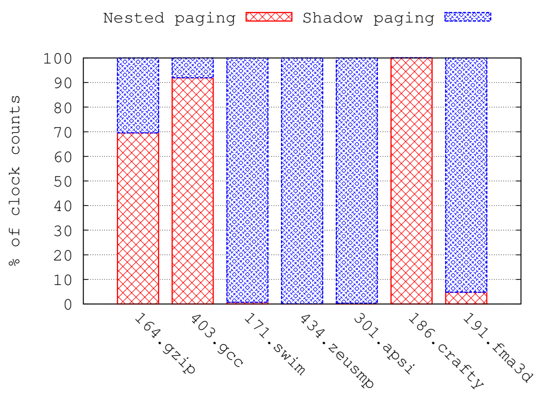


Figure 12: Percentage of time spent in each mode under DAV^{2}M.

Figure 12 shows the percentage of time that the benchmarks spend in each mode when run under DAV^{2}M. Here the three sets of benchmarks are quite evident. For the third set, in which the paging approach does not matter, we see that DAV^{2}M has made opposite decisions (171.$swim$ vs. 434.$zeusmp$), but with no real consequences for performance. At most one switch occurred.

8. RELATED WORK

The virtualization of paging has a history as long as virtual machine monitors themselves. For the x86 platform, the initial descriptions of the software-based approaches of shadow paging in VMware [20] and paravirtualized shadow paging in Xen [9] set the stage. Bhargava et al [11] give a detailed treatment of the hardware-based approach of nested paging and its optimization. Barr et al [10] propose new approaches to page walk caching that could be used to further accelerate nested paging. Adams and Ageson [4] compare hardware and software techniques in x86 virtualization, while Karger [17] compares x86 and DEC Alpha virtualization, a comparison that includes a excellent treatment of the aspects of x86 paging that make it particularly challenging to virtualize with high performance.

Wang et al [21] also propose, implement, and evaluate an adaptive approach to paging approach selection. $\mathrm{DAV^2M}$ uses a different mechanism and policy, and is evaluated in the context of a different VMM. Both papers find adaptive paging to be highly promising.

Our paper fits in the context of our group's efforts to achieve low overhead virtualization for high performance computing (see v3vee.org for more). Prior publications have analyzed the performance differences between different forms of shadow and nested paging [8, 19, 18], identified the opportunity for adaptive paging in HPC [8], and considered other hardware approaches to paging at the nested level [15].

9. CONCLUSIONS

We have made a case that no single approach to virtualizing virtual memory is best for maximizing application performance, focusing on the choice between shadow paging and nested paging. Rather, the choice is workload-dependent, and it may even vary over the life of a virtual machine. In response, we created a mechanism in our Palacios VMM for changing the paging approach at any time, and a policy, $\mathrm{DAV^2M}$, for driving that mechanism to increase application performance. We demonstrated that $\mathrm{DAV^2M}$ is able to adapt to workload, providing performance at least as good as the best statically chosen paging approach for the workload. Although our implementation is available in the Palacios VMM, the general idea could be applied in any VMM that supports multiple paging approaches.

10. REFERENCES

[1] Pinpoints. http://www.pintool.org/pinpoints.html.

[2] Puppy linux. http://puppylinux.org.

[3] SPEC CPU. http://www.spec.org.

[4] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (October 2006).

[5] AMD, INC. *AMD64 Architecture Programmer's Manual Vol 2: System Programming*, June.

[6] AMD, INC. *Software Optimization Guide for AMD Family 10h and 12h Processors*, December.

[7] AMD, INC. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, 3.3 ed., February 2006.

[8] BAE, C., LANGE, J. R., AND DINDA, P. A. Comparing approaches to virtualized page translation in modern vmms. Tech. Rep. NWU-EECS-10-07, Department of Electrical Engineering and Computer Science, Northwestern University, April 2010.

[9] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[10] BARR, T., COX, A., AND RIXNER, S. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (June 2010).

[11] BHARGAVA, R., SEREBRIN, B., SPANINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2008).

[12] BHATTACHARJEE, A., AND MARTONOSI, M. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (October 2009).

[13] BIENIA, C., AND LI, K. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (June 2009).

[14] GORMAN, M., AND HEALY, P. Performance characteristics of explicit superpage support. In *Proceedings of the 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (June 2010).

[15] HOANG, G., BAE, C., LANGE, J., ZHANG, L., DINDA, P., AND JOSEPH, R. A case for alternative nested paging models for virtualized systems. *Computer Architecture Letters 9*, 1 (January 2010), 17–20.

[16] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide Part 2*, January 2011.

[17] KARGER, P. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)* (June 2007).

[18] LANGE, J., PEDRETTI, K., DINDA, P., BRIDGES, P., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM International Conference on Virtual Execution Environments (VEE)* (March 2011).

[19] LANGE, J., PEDRETTI, K., HUDSON, T., DINDA, P., CUI, Z., XIA, L., BRIDGES, P., GOCKE, A., JACONETTE, S., LEVENHAGEN, M., AND BRIGHTWELL, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April 2010).

[20] WALDSBURGER, C. Memory resource management in vmware esx server. In *2002 Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[21] WANG, X., ZANG, J., WANG, Z., LUO, Y., AND LI, X. Selective hardware/software memory virtualization. In *Proceedings of the 2011 ACM International Conference on Virtual Execution Environments (VEE)* (March 2011).