Rethinking Multicore Application Scalability on Big Virtual Machines

Jianchen Shan, Weiwei Jia, Xiaoning Ding
Department of Computer Science, New Jersey Institute of Technology
Email: {js622, wj47, xiaoning.ding}@njit.edu

***Abstract*—Virtual machine (VM) sizes keep increasing in the cloud. However, little attention has been paid to analyze and understand the scalability of multicore applications on big VMs with multiple virtual CPUs (VCPUs), assuming that application scalability on VMs can be analyzed in the same ways as that on physical machines (PMs). The paper demonstrates that, since hardware CPU resource is dynamically allocated to VCPUs, the executions of multicore applications on VMs show different scalability from those on PMs. The paper systematically studies how the virtualization of CPU resource changes execution scalability, identifies key application features and system factors that affect execution scalability on VMs, and investigates possible directions to improve scalability.**

The paper presents a few important findings. First, the execution scalability of applications on VMs is determined by different factors than those on PMs. Second, virtualization and resource sharing can improve scalability by nature. Thus, applications may show better scalability on VMs than on PMs. Linear scalability can be achieved even when there is substantial sequential computation. Third, there is still much space to further improve execution scalability by enhancing system designs. Better scalability can be achieved by increasing allocation period length and/or matching resource allocation and workload distribution.

***Index Terms*—scalability, multicore, virtual machine, cloud computing, resource sharing**

I. Introduction

In the cloud, virtual machine (VM) sizes increase steadily to meet the demand for increasing computing power in each VM and to utilize the growing core counts in underlying physical machines. For example, a X1 instance on Amazon EC2 platform now has as many as 128 virtual CPUs (VCPUs) [1]. With increasing VM sizes, an important question to answer is how well applications can scale and take advantage of the computing power of bigger VMs to improve performance.

Common practice assumes that virtual machines have a similar architecture to their hosting physical machines (PMs), and thus the execution scalability of multicore applications on VMs can be analyzed in the same way as that on dedicated physical machines. As an evidence, VMs with multiple VCPUs on x86 architecture are called SMP-VMs or virtual SMPs [2], and Amdahl's law is used to analyze scalability.

However, due to the sharing of physical CPU resource on virtualized platforms and the dynamic CPU resource allocation for enabling the sharing, VCPUs show substantially different behaviors and performance features than physical computing cores. Thus, applications show different scalability on VMs than they do on physical machines. For example, research has shown that some multicore programs may suffer lower scalability on VMs, because the VCPUs in a VM may not make progress continuously and simultaneously [2], [3].

Although a few scalability problems have been noticed on VMs and the specific reasons have been analyzed, how the scalability of multicore applications in the cloud is changed by the virtualization of CPU resource has not been systematically studied. The paper analyzes and verifies with experiments how CPU resource sharing in virtualization impacts application scalability, identifies key application features and system factors affecting application scalability, and explore the potential and design alternatives for improving application scalability.

First, by re-defining speedup based on resource utilization efficiency as a measurement of scalability, the paper analyzes and reveals the fundamental reasons for multicore applications showing different scalability on VMs than they do on PMs (Section II). Second, based on the analysis, the paper identifies two key application features for scalability and shows with experiments how virtualization affects scalability differently for the applications with different features (Section III). Though applications are usually considered to have similar or lower scalability on VMs, the paper shows that virtualization tends to improve scalability. However, the improved scalability might be offseted by frequent synchronization and long scheduling delay. Third, though some applications already show better scalability on VMs than they do on PMs with existing system design, the paper shows that there is still much space to further improve scalability on VMs. Thus, the paper identifies the system factors that can be leveraged for improved scalability. The paper investigates two factors that have not been mentioned or studied before in other literatures — allocation period length and the matching between resource allocation and workload distribution. With experiments, the paper demonstrates that improved scalability can be achieved by increasing allocation period length or matching resource allocation and workload distribution (Section IV).

II. Resource Sharing's Impact on Scalability

Due to the sharing of CPU resource, the methods and models developed to analyze the execution scalability of applications on dedicated hardware, e.g., Amdahl's law, cannot be used for understanding the execution scalability of applications on VMs. This section first introduces how CPU resource is managed and shared on virtualized platforms. Then, it provides a new method which measures scalability based on

resource utilization efficiency. With the method, it explains how resource sharing affects execution scalability on VMs.

A. Resource Sharing between VMs

On a physical machine hosting multiple VMs, a virtual machine monitor (VMM) is used to manage and dynamically allocate hardware resource to each VM. For CPU resource, a physical CPU (PCPU) is usually time-shared by multiple VCPUs. The VMM treats VCPUs as independent schedulable entities and allocates CPU time to them. Inside each virtual machine, threads are further scheduled onto VCPUs by the guest OS. Thus, by having multiple VCPUs in each VM, the threads in the VM can eventually run on multiple PCPUs, achieving higher performance than that with a single VCPU.

When allocating CPU time, the VMM first allocates CPU time to VMs based on their weights, and then distributes CPU time to VCPUs in each VM. As in typical OS implementations, to guarantee responsiveness, CPU time is usually allocated periodically to VCPUs as their timeslices in each period. For brevity, we refer to the period in which CPU time is distributed to VCPUs as an **Allocation Period**.

A VCPU consumes its timeslice when it runs on a core. For improved efficiency, a VCPU is descheduled when it stops making progress (e.g., when it becomes idle or busy-waiting[1]), and stops consuming timeslice. If a VCPU is not rescheduled for long time, it is possible that the VCPU cannot consume up its timeslice in an allocation period. In such a case, the VMM usually does not roll over the unused timeslice or a part of the unused timeslice to the next allocation period, in order to prevent a VCPU from accumulating too much timeslice and starving other VCPUs on the same core.

B. Efficiency-Based Scalability Measurement

To analyze scalability on VMs, we introduce a new method, which measures scalability base on the utilization efficiency of CPU resource. Specifically, the scalability of an application is determined by how efficiently the increased resource is utilized during the execution of the application. The higher the efficiency (i.e., less waste) is, the more the application can be accelerated, and the higher the scalability is.

Scalability is how much an application can be accelerated if allocated with more resource, with speedup being a measurement. For CPU resource, the speedup of the execution on N processing unit (PU, i.e., cores in PMs or VCPUs in VMs) against that on 1 processing unit is as follows.

$$\begin{aligned} Speedup &= \frac{\text{execution speed on N PUs}}{\text{execution speed on 1 PU}} \\ &= \frac{\text{work finished on N PUs in an unit of time}}{\text{work finished on 1 PU in an unit of time}} \end{aligned}$$

Without loss of generality, the paper assumes that the amount of work finished is proportional to the CPU time utilized for effective computation. Since the total CPU time available on N PUs is N times of that on 1 PU, the above equation can be rewritten as follows.

$$\begin{aligned} &Speedup \\ &= \frac{\text{total CPU time utilized on N PUs in an unit of time}}{\text{CPU time utilized on 1 PU in an unit of time}} \\ &= N \times \frac{\frac{\text{total CPU time utilized on N PUs in an unit of time}}{\text{total CPU time available on N PUs in an unit of time}}}{\frac{\text{CPU time utilized on 1 PU in an unit of time}}{\text{CPU time available on 1 PUs in an unit of time}}} \\ &= N \times \frac{\text{overall utilization efficiency with N PUs}}{\text{utilization efficiency with 1 PU}} \end{aligned}$$

In the above equation, **utilization efficiency is the ratio between the amount of utilized CPU time and the amount of available CPU time**, and the utilized CPU time is that consumed for effective computation. The CPU time spent on busy-waiting does not count. Assume the utilization efficiency of 1 PU (i.e., serial execution) is 100%. The speedup with N PUs can be simplified as follows.

$$\begin{aligned} Speedup &= N \times \text{overall utilization efficiency with N PUs} \\ &= N \times \frac{\text{CPU time utilized by computation}}{\text{available CPU time during computation}} \\ &= N - N \times \frac{\text{unutilized CPU time}}{\text{available CPU time during computation}} \end{aligned}$$

In the equation, *unutilized CPU time* refers to the CPU time that is not utilized by the application to make progress.

The above definition of speedup is consistent with that based on execution time. For example, based on Amdahl's law, if in an application 20% of computation can only be executed sequentially and 80% of computation can be fully parallelized without overhead, when executed on a 4-core machine, the speedup against the execution on a single core machine is $1/(0.2 + 0.8/4) = 2.5$. The performance does not scale linearly. This is because, when the sequential portion is executed on one core, other cores are idle. This reduces the utilization efficiency to 50% on these cores. The overall utilization of the 4 cores is $(100\% + 3 \times 50\%)/4 = 62.5\%$, and the speedup based on the above definition is $4 \times 0.625 = 2.5$.

The above definition of speedup can be used to understand both the scalability on physical machines with dedicated resources and the scalability on VMs with shared resources. To highlight the reasons causing different scalabilities on these platforms, we adapt the speedup calculation for physical machines and virtual machines respectively as follows.

For the executions on a physical machine, *unutilized CPU time* is the CPU time wasted on idling and busy-waiting during the execution. Thus, the speedup of an application on a physical machine with N cores can be calculated as follows.

$$\begin{aligned} \text{Speedup\_PM} &= N - N \times \frac{\text{time on idling and busy-waiting}}{N \times \text{execution time}} \\ &= N - \frac{\text{time on idling and busy-waiting}}{\text{execution time}} \end{aligned}$$

The CPU resource for a virtual machine is its timeslice. For the executions on a virtual machine, *unutilized CPU time* consists of two parts. The first part, *unused timeslice*, is the timeslice that cannot be depleted by the VCPUs in an

[1]Most multicore processors have equipped with mechanisms, such as Intel PLE and AMD PF, to detect and interrupt busy-waiting.

allocation period and cannot be rolled over to later allocation periods (Section II-A). The second part is the CPU time used to handle idle VCPUs and spinning VCPUs. Due to resource sharing, idleness and spinning are handled in a substantially different way on VMs than on PMs. To improve the utilization of shared CPU resource, hardware and the VMM usually try their best to detect and deschedule VCPUs that are not making progress, including idle VCPUs and spinning VCPUs. Thus, CPU time is not wasted on idling and busy-waiting. However, time must be spent to switch out these VCPUs. Therefore, the speedup of the execution on a VM with N VCPUs (against that on a VM with a single VCPU) can be calculated as follows[2].

$$\text{Speedup\_VM} = N - N \times \left(\frac{\text{overhead of switching out idle/spinning VCPUs}}{\text{timeslice allocated to the VM}} + \frac{\text{unused timeslice of the VM}}{\text{timeslice allocated to the VM}}\right)$$

C. Virtualization's Impact on Scalability

The impact of virtualization and CPU resource sharing on scalability can be identified by comparing the equations for calculating Speedup_PM and Speedup_VM. On a dedicated physical machine, resource provisioning is static. The scalability is mainly determined by the behavior of the application, i.e., whether the application can engage all the cores in useful work. Any idleness and busy-waiting are translated into lower resource utilization and then lower scalability. The reduction of scalability is proportional to the durations of idleness and busy-waiting. This also explains Amdahl's law and other models for analyzing execution scalability on dedicated hardware, which co-relate scalability with the operations causing idleness and busy-waiting, such as sequential computation, tasks on critical path, and synchronizations.

Virtualization affects scalability in two ways. On one hand, dynamic resource allocation helps improving scalability. For VCPUs, resource is not consumed if there is not useful work on them. Thus, even if an application cannot always engage the VCPUs in useful work, high scalability may still be achieved, as long as the overhead incurred by VCPU switches is low and most of the timeslice of the VM is eventually consumed by the end of resource allocation periods. An execution on VM may achieve linear scalability even if there is substantial sequential computation. Thus, conventional methods and models (e.g., Amdahl's law) become inapplicable when used to understand application scalability on VMs.

On the other hand, scalability on VMs is limited by new factors — VCPU switch overhead and unused timeslice of the VM. These factors are determined not only by the natures of the computation in applications but also the resource management at the system level. Specifically, VCPU switch overhead is proportional to the frequency of VCPU switches, which are usually incurred by the synchronizations on VCPUs. The more frequent the synchronizations are, the lower the scalability is. A few factors affect the amount of unused timeslice. First, the amount of unused timeslice is determined by whether there is enough workload in each resource allocation period to consume timeslice. Second, timeslice and workload are distributed to each VCPU. Thus, the amount of unused timeslice is also determined by the distribution of workload and the distribution of timeslice to VCPUs. When a VCPU is allocated with more timeslice than needed by the workload on it, some timeslice will not be used. Finally, VCPU scheduling may also significantly affect the amount of unused timeslice. If a VCPU is scheduled late, the workload on it may not have enough time to consume the timeslice available to the VCPU by the end of a resource allocation period, increasing unused timeslice.

In Section III, we identify and experimentally verify the application features affecting the scalability on VMs, and in Section IV, we investigate CPU resource management in the VMM and identify system-level factors affecting execution scalability.

III. Application Features Affecting Scalability

This section identifies two key application features for scalability and shows how these features affect application scalability on VMs.

A. Key Application Features and Scalability Indications

Based on the analysis in Section II, we have identified two key scalability features of applications. One feature is *workload parallelism*, which describes to what degree an application can parallelize its workload in order to utilize increased CPU resource. During the execution of an application, its workload parallelism can be measured by the number of threads in the application that are active and making progress. In a time period, the higher the workload parallelism is, the more progress the application can make if provided with more resource. An application with higher workload parallelism tends to show higher execution scalability on both physical machines (because of less idle time) and virtual machines (because of less unused CPU time).

The other feature is *the frequency of blocking synchronizations* (referred to as *synchronization frequency* for brevity). Blocking synchronizations can incur the switches of VCPUs, which reduce scalability in two ways. First, when the threads on a VCPU are blocked, the VCPU is descheduled, and another VCPU (probably from another VM) is scheduled. The switch of VCPU incurs high overhead. Second, when a thread on the descheduled VCPU is unblocked and becomes ready to make progress, the VCPU may not be able to be rescheduled immediately. Due to this *rescheduling delay*, the VCPU may not be able to fully utilize the timeslice allocated to it by the end of allocation periods, increasing unutilized timeslice.

[2]With existing system designs, spinning that is very brief or at the user-level of VMs may not be detected. Such spinning still consumes CPU time. The paper chooses to neglect the CPU time used by such spinning because 1) minimal CPU time is used by brief spinning and 2) excessive spinning at the user-level should be prevented using co-scheduling or interrupted using hardware facilities similar to Intel PLE and AMD PF.

TABLE I
A SUMMARY OF FOUR TYPES OF APPLICATIONS BASED ON THEIR KEY SCALABILITY FEATURES ON VMS

Type	Work. Para.	Sync. Freq.	Scalability	Benchmarks
1	high	low	high	p.freqmine, p.swaption, p.x264, p.ferret, p.vips, s.water_nsquared, s.barnes, s.lu_ncb, s.raytrace, s.radix
2	low	high	low	p.bodytrack, p.dedup, p.facesim, s.ocean_cp, s.volrend, s.cholesky
3	high	high	mediocre	p.fluidanimate, p.streamcluster, s.ocean_ncp
4	low	low	mediocre	p.canneal, p.raytrace, p.blackschole, s.fmm, s.radiosity, s.water_spatial s.fft, s.lu_cb

Based on these features, multicore applications can be categorized into four types, as summarized in Table I. Applications with high workload parallelism and low synchronization frequencies (*type 1*) usually show high scalability on VMs; applications with low workload parallelism and high synchronization frequencies (*type 2*) usually show low scalability on VMs; applications with high workload parallelism and high synchronization frequencies (*type 3*) and applications with low workload parallelism and low synchronization frequencies (*type 4*) show mediocre scalability.

B. Experimental Verification

To verify the scalability indications of the aforementioned application features through experiments, we select the benchmarks in PARSEC 3.0 suite [4], including native PARSEC benchmarks and SPLASH2X benchmarks. We attach a prefix 'p.' before the name of each native PARSEC benchmark, and attach a prefix 's.' before the name of each SPLASH2X benchmark, in order to differentiate these two sets of benchmarks. We also refer to native PARSEC benchmarks as PARSEC benchmarks for brevity. We used the parsecmgmt tool in the PARSEC package to run the benchmarks with native input and to control the number of concurrent threads in each execution.

Experiments were conducted a Dell PowerEdge R720 server with 64GB of DRAM and two 2.40GHz Intel Xeon E5-2665 processors. Each processor has 8 cores. On the server, we created 4 VMs. Each VM has 16GB of memory and 16 VCPUs. The VMM is KVM. The host OS and the guest OS are Ubuntu version 14.04 with the Linux kernel version updated to 3.19.8. The VCPUs in each VM were laid out on the cores in a way to prevent VCPU stacking for better performance [2].

We first profiled the benchmarks to obtain their scalability features when we run them on the physical server. The number of thread in each execution is 16. During the execution of each benchmark, we collected the number of active CPU cores involved in the benchmark computation periodically and the number of voluntary context switches[3]. The workload parallelism of the benchmark is the average number of active cores during its execution, and its synchronization frequency is the number of voluntary context switches per second.

Figure 1 shows the categorization of the benchmarks based on their scalability features. If a benchmark keeps at least 75% of the cores (i.e., 12 cores in our system) active on average during its execution, it is considered to have high workload parallelism. If the time period between two consecutive synchronizations is shorter than the timeslice allocated to a thread in an allocation period (i.e., a thread is blocked at least once

[3]Voluntary context switches are context switches caused by threads blocking their execution voluntarily, i.e., blocking synchronizations.

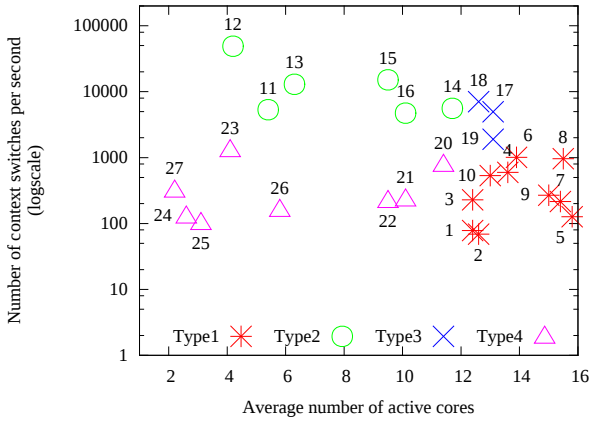

Fig. 1. The types of the PARSEC benchmark and their scalability features. The numbers are the indexes of the benchmarks, which are indexed as follows. 1: p.freqmine, 2: s.water_nsquared, 3: s.barnes, 4: s.lu_ncb, 5: p.swaption, 6: p.x264, 7: p.ferret, 8: p.vips, 9: s.raytrace, 10: s.radix, 11: p.bodytrack, 12: p.dedup, 13: p.facesim, 14: s.ocean_cp, 15: s.volrend, 16: s.cholesky, 17: s.ocean_ncp, 18: p.streamcluster, 19: p.fluidanimate, 20: s.lu_cb, 21: s.water_spatial, 22: s.fmm, 23: p.canneal, 24: s.fft, 25: p.raytrace, 26: p.blackschole, 27: s.radiosity

before it uses up its timeslice), the benchmark is considered to have high synchronization frequency.

The benchmarks of each type are summarized in Table I. Note that an application with low workload parallelism only means that the application lacks enough active threads to keep all the cores/VCPUs busy. As we will show later that an application with low workload parallelism may still achieve decent scalability on a VM. At the same time, the workload parallelism is relative to the scale of the system (e.g., the number of cores/VCPUs in a server/VM). An application with high workload parallelism may become one with low workload parallelism on a larger system.

Then, we run the benchmarks in a VM consolidated with three other VMs on the same physical server. To obtain stable measurement, we run a CPU-bound program in each of three VMs, which keeps increasing a counter on all the VCPUs of the VM. We show the speedups of the benchmarks in Figure 2. The concurrency level (i.e., the number of threads in the benchmark, the number of VCPUs in each VM, and the number of cores used in the PM) is 16. The speedup is relative to the performance with concurrency level equal to 1.

The benchmarks of the first type show the highest scalability, and the speedups are similar on the PM and the VM. The average speedups are both 13.9. Their high speedups are achieved for two reasons: 1) high workload parallelism ensures that CPU resource is fully utilized; and 2) there are no factors reducing the utilization of CPU resource.

Other benchmarks show different scalability behaviors on the VM than on the PM. On the PM, the speedups are mainly

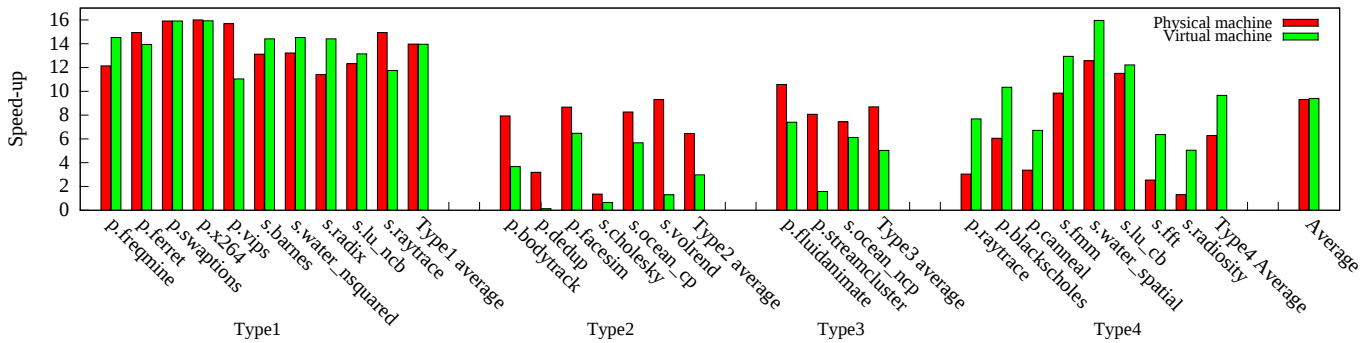

Fig. 2. Speedups of PARSEC and SPLASH2X benchmarks.

determined by the workload parallelism. The benchmarks of the third type also show high scalability, because they have high workload parallelism. The average speedup is lower than that of the first type, because their workload parallelism is lower (Figure 1). The benchmarks of the second type and the benchmarks of the fourth type show similar scalability. The average speedups are similar (6.5 and 6.3), despite the differences in synchronization frequency. The average speedups are lower than those of the first and the third types.

Speedups are determined by both workload parallelism and synchronization frequencies on the VM. Though benchmarks with higher workload parallelism still achieve higher speedups than those with lower workload parallelism (e.g., the benchmarks of the first type show higher scalability than those of the fourth type), synchronization frequencies tend to have a larger impact on scalability than workload parallelism. This is evidenced by the benchmarks of the fourth type achieving higher speedups (9.7 on average) than those of the third type (5 on average), though the benchmarks of the fourth type have lower workload parallelism. Synchronizations also make the benchmarks show lower scalability on the VM than on the PM. The average speedups of the benchmarks of the second and the third types are 6.5 and 8.7 on the PM, respectively, and are 3 and 5 on the VM, respectively.

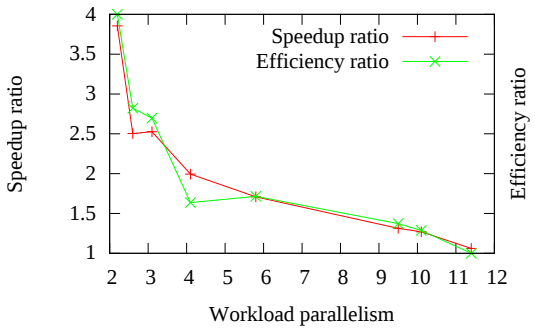

Fig. 3. Impact of virtualization on scalability for applications with different workload parallelism.

Interestingly, the benchmarks of the last type achieve better scalability on the VM than on the PM. The average speedups are 9.7 on the VM and 6.3 on the PM. This confirms that virtualization inherently improves scalability when synchronizations are infrequent. To better understand how virtualization improves scalability for these benchmarks. We collected the resource utilization efficiencies during their executions on the PM and the VM. Then, for each benchmark, we calculate the ratio between its speedup on the VM and its speedup on the PM, and the ratio between the efficiencies. Figure 3 shows that, for the benchmarks with different workload parallelism, their efficiency ratios are always greater than 1, and the speedup ratios change consistently with the efficiency ratios. This indicates that the scalability improvements are through making more efficient utilization of resources. Figure 3 also shows that the speedup ratios decrease with the growth of workload parallelism. This is because, with the growth of workload parallelism, the space for increasing scalability decreases.

IV. Improving Scalability at the System Level

At the system level, the management of CPU resource has great impact on application scalability on VMs. For best scalability, the system should allocate CPU time in a way that each VM can maximize the utilization of its timeslice. In this section, we first show that there is still much space for improving the existing system design to achieve better scalability. Then, we identify two system factors that can be leveraged to improve scalability.

A. Potential for Improving Scalability on VMs

To understand the potential for achieving better scalability with improved system designs, we estimated the resource utilization efficiencies that the benchmarks could possibly achieve if the VMM could support the VM to utilize its timeslice as fully as possible. We selected the benchmarks of the fourth type, because 1) we want to focus on improving the allocation of CPU time, instead of reducing the overhead of VCPU switches, and 2) there is space to further improve their scalability.

The estimation is based on profiling the benchmarks on the physical server. For each allocation period during the execution of a benchmark, we collect the CPU time utilization of the benchmark. If the utilization u is higher than the portion p of the CPU time that a VM can obtain (e.g., a 60% utilization vs. 25% of CPU time that a VM can obtain when it is co-located with another 3 VMs), we expect that, with a well-designed VMM, the VM can deplete the timeslice allocated to it and achieve an efficiency of 100% when the computation is executed on the VM. Otherwise, the benchmark does not

have enough computation to deplete the timeslice allocated to the VM. Thus, in the period, the efficiency is the ratio between u (utilization) and p (portion of CPU time allocated to a VM). The estimated resource utilization efficiency is the average efficiency during the execution.

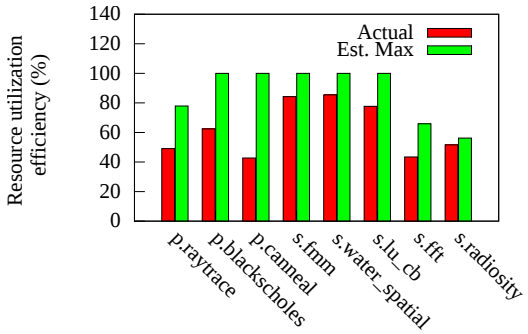

Fig. 4. Actual resource utilization efficiency and estimated maximal resource utilization efficiency of the benchmarks of the fourth type on a VM.

Figure 4 shows the actual resource utilization efficiency and the estimated maximal efficiency of the benchmarks with a concurrency level of 16. On average, the actual resource utilization efficiency is 62.1% (average speedup is 9.7), and the estimated maximal efficiency is 87.5% (corresponding to a speedup of 16*87.5%=14). The benchmark *p.canneal* shows the largest potential (from 43% to 100%). This clearly shows that there is still much space to further improve the management of CPU resource to achieve higher scalability.

B. Possible Optimizations on CPU Time Allocation

Based on the analysis in Section II, application scalability on a VM can be improved by reducing the overhead incurred by VCPU switches and reducing unused timeslice. Extensive research has been conducted on the techniques reducing the impact of VCPU switches on application performance and scalability, such as improving VCPU scheduling at the VMM level and improving task scheduling at the guest OS level [2], [3], [5]–[8], or reducing scheduling latencies [9]–[11]. Thus, the paper focuses on investigating the factors in CPU time allocation to reduce unused timeslice.

•***Longer allocation periods***: A VCPU is allowed to use its timeslice within each allocation period. If a VCPU has light workload in an allocation period, it may not deplete its timeslice by the end of the period. The unused timeslice may not be rolled over for the VCPU to handle possibly heavy workload later. Increasing allocation period length can tolerate such workload fluctuation on VCPUs, and reduce unused timeslice of the VCPUs when they have light workload.

To verify the impact of allocation period length on scalability, we repeated the experiments described in section III for different allocation period lengths from 24ms (i.e., system default value) to 192ms, and show the speedups of benchmarks of the fourth type in Figure 5. Increasing allocation period length does significantly improve execution scalability for *p.canneal* and *s.lu_cb* and slightly improve execution scalability for *s.radiosity*, *p.blackscholes*, *s.fmm*, and *s.water_spatial*. We also notice that *s.lu_cb* even achieves linear scalability

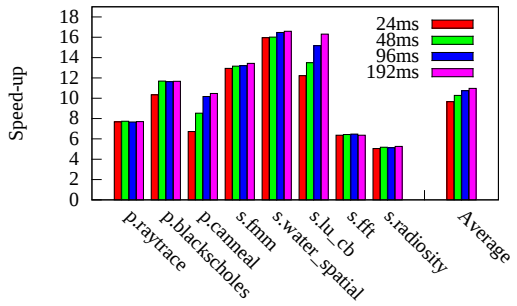

Fig. 5. Speedups when allocation period length is varied from 24ms to 192ms.

when allocation period length is increased to 192ms. Increasing allocation period length has different impact for different benchmarks because the workloads may fluctuate on different time scales and with different intensity. Though increasing allocation period length cannot increase scalability for *s.fft* and *p.raytrace*, the average speedup of these benchmarks is increased from 9.6 to 11.0 when the allocation period length is increased to 192ms.

These results confirm that longer allocation periods can really improve the execution scalability of multicore applications on VMs. However, increased allocation periods lengthen responding latencies, which may not be desirable for interactive workloads. Thus, long allocation periods may only be applicable to throughput-oriented workloads.

•***Matching resource allocation and workload distribution***: In a VM, workload is distributed to VCPUs for concurrent execution, utilizing the CPU resource (i.e., timeslice) allocated to the VCPUs. Desirable performance can only be achieved when the allocation of timeslice matches the distribution of workload. Allocating more timeslice to a VCPU than what is needed by the workload on the VCPU will cause some timeslice unused by the end of allocation periods. Allocating insufficient timeslice to a VCPU with heavy workload delays the computing tasks on the VCPU.

Matching workload distribution and CPU resource allocation can be done by task scheduling either in guest OSs or applications. Existing VMM designs try to allocate timeslice evenly to VCPUs within each VM. As we will show with an illustrative example in Figure 6, task schedulers can evenly distribute workload on the time scale of allocation periods to improve application scalability.

In the example, a program executes a loop, in which each iteration has 2 units of sequential tasks followed by 4 units of parallel tasks. If the program runs on a 4-core PM, the speed-up is 2 based on Amdahl's law. Figure 6 shows the executions of the program on a 4-VCPU SMP VM co-located with another VM, and compares the executions with different methods of distributing workloads to VCPUs. We assume that the length of an allocation period is 12 time units and each time unit can finish one unit of task. Two VMs have the same weight. Thus, in an allocation period, each VM is assigned with 50% of CPU time (i.e., 12*4/2=24 units of CPU time), and each VCPU receives 6 units of CPU time.

Subfigure (A) shows the execution of the program on one VCPU. In an allocation period, 6 units of tasks are finished

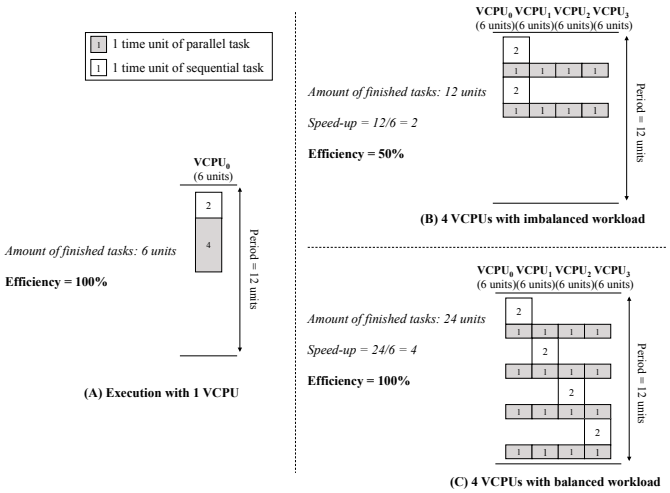

Fig. 6. An illustrative example to explain the benefit of evenly distributing workload and how even workload distribution can be achieved.

since the VCPU has 6 units of CPU time. Subfigure (B) shows the execution on four VCPUs with a conventional task scheduler. Though parallel tasks can be evenly distributed to the VCPUs, sequential tasks cannot, and are assigned to the same VCPU (VCPU0 in the figure). Thus, only two iterations can be finished within an allocation period. After the second iteration, VCPU0 consumes up its 6 units of CPU time, though other VCPUs only consume 2 units of CPU time each and still have unused timeslice. These VCPUs finish only 12 units of tasks in total in the allocation period, which are 2 times as many as those finished with one VCPU. Thus, the speedup is 2, the same as that on the PM.

Subfigure (C) shows the execution on four VCPUs with an improved task scheduler assigning sequential tasks onto different VCPUs in different iterations. Though the workload is not evenly distributed at every moment, the workload on the VCPUs is balanced on the time scale of allocation periods. With such workload distribution, every VCPU can consume its 6 units of CPU time and finish 6 units of tasks (four iterations) in an allocation period. With the improved task scheduler, linear speed-up can be achieved, i.e., a speed-up of 4 with 4 VCPUs.

Matching workload distribution and CPU resource allocation can also be done by adjusting the CPU time allocated to VCPUs based on the workload on them. The benefits can be illustrated with the execution shown in Figure 6(B). If the VMM can allocate 12 units of CPU time to VCPU0 and 4 units of CPU time to each of the other three VCPUs in each allocation period, the program can still finish 4 iterations in the period, achieving linear scalability (the figure only shows the first two iterations). Note that the amount of CPU time received by the VM is not increased. The increased scalability comes from distributing more CPU time to VCPU0, which has heavier workload than other VCPUs.

We tested the above approaches using a synthetic benchmark, which generates the workload of typical fork-join multicore programs. Specifically, the benchmark executes a loop, in which each iteration finishes eight units of computation

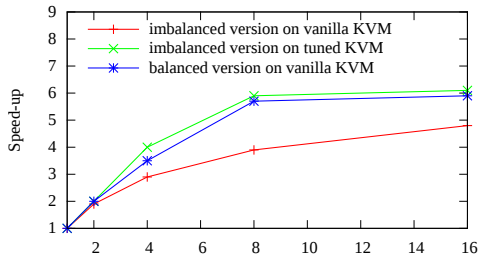

Fig. 7. Speedups of the synthetic benchmark.

that can be fully parallelized and one unit of computation that can only be executed sequentially. Each unit of computation takes about 1ms to finish. We developed two versions of the synthetic benchmark. In the *imbalanced* version, sequential tasks are executed by the same thread, as that shown in Figure 6(B). In the *balanced* version, sequential tasks in different iterations are assigned to threads in a round-robin manner. This is to emulate the execution with a task scheduler trying to balance the workload on VCPUs.

We first run both versions on a VM managed by the vanilla KVM, which tries to allocate CPU time evenly to VCPUs, and compare their performance. The VM is co-located with three other VMs, each of which runs an instance of the CPU-bound program incrementing a counter. Then, specifically for the imbalanced version, we tuned KVM settings to allocating CPU time to VCPUs proportionally to the workload on them, and rerun the imbalanced version on the VM.

Figure 7 shows the speedups of the benchmark when the concurrency level is increased from 1 to 16. The balanced version on vanilla KVM and the imbalanced version on tuned KVM show higher scalability than the imbalanced version on vanilla KVM. When the concurrency level is 16, the speedups are 5.9, 6.1, and 4.8, respectively. This shows that both approaches to match resource allocation and workload distribution are effective to improve scalability. The benchmark cannot achieve linear scalability mainly because there are frequent synchronizations incurred at the beginning and the end of the sequential computation in each iteration.

V. Related Work

To understand scalability, analytical models have been developed based on various workload characteristics, such as synchronization and communication [12], critical path [13], memory accessing traffic [14], and the amount of sequential computation. These models target physical machines with dedicated hardware resource, and cannot be directly applied to understand the execution scalability on virtual machines. Various models have been developed in computer architecture area to study how to distribute hardware resource, such as transistors and chip area, to the functional units in multicore processors to maximize application performance and scalability [15]–[18]. They are remotely related with our work.

Targeting application performance on VMs, existing work mainly focuses on characterizing the interference caused by the contention of the shared hardware resource on memory

hierarchy (e.g., processor cache and memory bandwidth) between co-located VMs [19]–[24]. The main purpose is to understand and alleviate the performance degradation incurred by the interference. This paper is to identify the application features and system factors affecting the execution scalability of applications on VMs. Existing research on application performance on VMs is orthogonal to our research.

To improve the execution scalability of multicore applications on VMs, various techniques have been attempted at all the system layers, from hardware support (e.g., PLE) [25]–[27], VMM [2], [5]–[7], guest OSs [3], [28], [29], to programming framework [30]. These techniques only target the virtualization overhead on communication/synchronizations between application threads. The paper discusses application scalability on VMs in a wider scope, with communication/synchronization being one of the scalability factors.

VI. Conclusion

The paper aims to understand how virtualization and CPU resource sharing affect the execution scalability of multicore applications. It does not mean to exhaustively investigate all the factors affecting application scalability (e.g., memory latency, I/O operations). It focuses only on the factors related to CPU time, which is the most important resource for achieving high performance. With analysis and experiments, the paper shows that application scalability on VMs is mainly affected by a few application features, including workload parallelism and synchronization frequencies, and a few system factors in CPU resource management, including allocation period lengths and the matching between workload distribution and CPU resource allocation. We hope the findings of the paper can help cloud computing users gain better understand on the performance of their programs in the cloud and make better choices between physical machines and different types of VM instances. We also hope the paper help motivating system researchers and developers to consider the factors limiting scalability and explore practical solutions ameliorating the impact of these factors.

VII. Acknowledgment

This research was supported by the National Science Foundation (NSF) under Grants No. SHF 1617749 and CNS 1409523. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. The United States Government is authorized to reproduce and distribute reprints notwithstanding any copyright notice herein.

References

[1] (2017) Amazon ec2 instance types. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[2] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *EuroSys 2011*. ACM, 2011, pp. 257–272.

[3] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications," in *USENIX ATC '14*, 2014, pp. 73–84.

[4] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS '09*, vol. 2011, 2009.

[5] L. Cheng, J. Rao, and F. Lau, "vScale: automatic and efficient processor scaling for smp virtual machines," in *EuroSys '16*. ACM, 2016, p. 2.

[6] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *VM 2004*, 2004, pp. 43–56.

[7] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," in *ASPLOS '13*, 2013, pp. 369–380.

[8] X. Ding, P. Gibbons, and M. Kozuch, "A hidden cost of virtualization when scaling multicore applications," in *HotCloud '13*. USENIX, 2013.

[9] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *Micro '14*. IEEE Computer Society, 2014, pp. 394–405.

[10] S. Wu, Z. Xie, H. Chen, S. Di, X. Zhao, and H. Jin, "Dynamic acceleration of parallel applications in cloud platforms by adaptive time-slice control," in *IPDPS '16*, 2016, pp. 343–352.

[11] B. Teabe, A. Tchana, and D. Hagimont, "Application-specific quantum for multi-core platform scheduler," in *EuroSys '16*, 2016, p. 3.

[12] L. Yavits, A. Morad, and R. Ginosar, "The effect of communication and synchronization on amdahl's law in multicore systems," *Parallel Computing*, vol. 40, no. 1, pp. 1–16, 2014.

[13] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *ISCA '10*, 2010, pp. 362–370.

[14] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *IPDPS '10*, 2010, pp. 1–12.

[15] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[16] N. Gunther, S. Subramanyam, and S. Parvu, "A methodology for optimizing multithreaded system scalability on multicores," *Programming multi-core and many-core computing systems*, pp. 363–384, 2011.

[17] T. Oh, H. Lee, K. Lee, and S. Cho, "An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor," in *ISVLSI'09*, 2009, pp. 181–186.

[18] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *MICRO '10*, 2010, pp. 225–236.

[19] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *ISPASS '11*, 2011, pp. 76–86.

[20] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *SC '12*, 2012, p. 51.

[21] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2: Cpu performance isolation for shared compute clusters," in *EuroSys 2013*. ACM, 2013, pp. 379–391.

[22] D. M. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments." in *USENIX ATC '13*, 2013, pp. 219–230.

[23] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *ISPASS '07*. IEEE, 2007, pp. 200–209.

[24] Y. Zhao, J. Rao, and Q. Yi, "Characterizing and optimizing the performance of multithreaded programs under interference," in *PACT '16*. ACM, 2016, pp. 287–297.

[25] M. Righini, "Enabling Intel® virtualization technology features and benefits," Intel, Tech. Rep., 2010.

[26] AMD. (2015) Amd64 architecture programmer's manual volume 2: System programming. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf

[27] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *MICRO '14*, 2014, pp. 394–405.

[28] J. Ouyang and J. R. Lange, "Preemptable ticket spinlocks: Improving consolidated performance in the cloud," in *ACM VEE 2013*, 2013, pp. 191–200.

[29] S. Kashyap, C. Min, and T. Kim, "Opportunistic spinlocks: Achieving virtual machine scalability in the clouds," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 9–16, 2016.

[30] Y. Peng, S. Wu, and H. Jin, "Towards efficient work-stealing in virtualized environments," in *CCGrid '15*. IEEE, 2015, pp. 41–50.