Less can be More: *micro*-Managing VMs in Amazon EC2

Jiawei Wen
College of William and Mary
Williamsburg, VA, USA
jwen01@email.wm.edu

Lei Lu
College of William and Mary
Williamsburg, VA, USA
lulei.wm@gmail.com

Giuliano Casale
Imperial College London
London, UK
g.casale@imperial.ac.uk

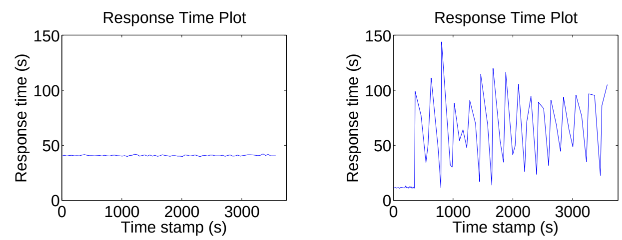
Evgenia Smirni
College of William and Mary
Williamsburg, VA, USA
esmirni@cs.wm.edu

***Abstract*—*Micro* instances (*t1.micro*) are the class of Amazon EC2 virtual machines (VMs) offering the lowest operational costs for applications with short bursts in their CPU requirements. As processing proceeds, EC2 throttles CPU capacity of *micro* instances in a complex, unpredictable, manner. This paper aims at making *micro* instances more predictable and efficient to use. First, we present a characterization of EC2 *micro* instances that evaluates the complex interactions between cost, performance, idleness and CPU throttling. Next, we define adaptive algorithms to manage CPU consumption by learning the workload characteristics at runtime and by injecting idleness to diminish host-level throttling. We show that a gradient-hill strategy leads to favorable results. For CPU bound workloads, we observe that a significant portion of jobs (up to 65%) can have end-to-end times that are even *four times shorter* than those of the more expensive *m1.small* class. Our algorithms drastically reduce the long tails of job execution times on the *micro* instances, resulting to favorable comparisons against even *small* instances.**

I. Introduction

Micro VMs (*t1.micro*) are a class of lightweight virtual machines that are part of the Amazon EC2 offering. According to the official documentation [1] they provide: (i) a small amount of consistent CPU resources and (ii) additional short bursts of CPU capacity when spare cycles become available. *Micro* instances can provide up to two EC2 compute units, but this capacity is offered only for short periods of time; no stable performance is guaranteed for the remaining time. One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. In order to compensate for the lack of performance predictability, *micro* VMs are then offered on-demand at much cheaper rates than any other VM class [2]. This leaves to the user the burden of devising the most appropriate management policy for a *micro* instance, which is a complex task.

To better understand the risks and unknowns that arise when using the *micro* offering, consider the following experiment. We instantiate a *micro* VM (*t1.micro*) and a *small* VM (*m1.small*) in EC2 (Virginia) and run *avrora*, a CPU intensive benchmark from the DaCapo suite [3], repeatedly for about 1 hour. A *small* provides 1 vCPU, 1 ECU, and 1.7GiB of RAM. Figure 1 the illustrates response times (i.e., runtimes) for each *avrora* execution. Consistently with our expectations, in the *small* instance, response times are stable[1]. Instead, in a *micro* VM, *avrora* begins with very short response times for about 8 minutes. Then, performance degrades sharply due to host-level CPU throttling and fluctuates widely over time, almost in a periodic pattern. In addition, average response times increase over time, casting doubts on the ability of these VMs to sustain continued application load.



(a) *small* VM (b) *micro* VM

Fig. 1: Typical *m1.small* and *t1.micro* performance behaviors

We focus on applications that run at the timespan of minutes or hours, even though possibly serving smaller units of work, and devise novel management techniques for *micro* VMs. Our main contributions are as follows. First, we provide a statistical characterization of the performance of *micro* VMs, focusing on the impact of artificially limiting their CPU consumption by injecting delays. This is useful as it increases our understanding of this cloud offering model. Second, we expose an interesting, previously unnoticed, behavior of *micro* VMs. Depending on the workload characteristics, it is often possible to inject delays in-between periods of CPU consumption of a *micro* VM to make it *simultaneously* cheaper and under some conditions even better performing than a *small* VM, even across timespans of hours. While it is known that extended idleness allows a *micro* VM to reclaim its initial high-performance characteristics, idleness also degrades throughput. Devising the optimal delay is difficult, particularly with a static choice, since it depends on the workload characteristics and the specific VM instance. To address this, we propose management algorithms for automatic delay injection *at runtime* in *micro* VMs and evaluate their performance showing promising results. Depending on the user's target, the algorithm may focus on finding the optimal delay to minimize end-to-end response time or to maximize application throughput. While Amazon's official documentation recommends usage of *micro* VMs for applications with short-term CPU burst requirements, the algorithms we propose can enable efficient longer-term usage of *micro* VMs.

Summarizing, our investigation answers the following:

- What is the trade-off between response time and host-level CPU throttling in *micro* VMs?
- Is it efficient to use *micro* VMs for continuously running applications?

[1] On I/O intensive benchmarks, reported later, no noticeable differences with respect to stability can be seen between *small* and *micro* VMs.

- What algorithms can we use to manage at runtime *micro* VMs?

The paper is organized as follows. Definitions and methodology are given in Section II, followed by a characterization study in Section III. Section IV introduces the runtime management algorithms, which are evaluated in Section V. In Section VI we review related work. Section VII outlines conclusions and future work.

II. Methodology

A. Reference Benchmarks

We begin by defining the reference benchmarks that we use throughout this paper. Our experiments use the following benchmarks from the *DaCapo* [3] and *Sysbench* [4] suites:

- *Avrora* that simulates a number of programs run on a grid of AVR micro controllers;
- *Luindex* that uses lucene [5] to index a set of documents;
- *Sysbench CPU* that calculates prime numbers up to a specified value;
- *Sysbench IO* that performs file I/O creation operations.

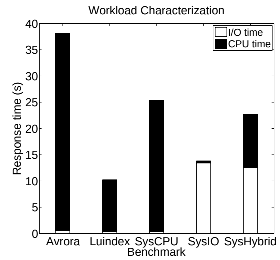
We also create a customized workload, *Sysbench hybrid*, that combines both *sysbench CPU* and *sysbench IO* to perform prime number calculations and file operations, essentially an alternation of the two standard *sysbench* benchmarks. *Sysbench hybrid* spends nearly equal time on CPU and IO.

Experiments are repeated on both *small* (*m1.small*) and *micro* (*t1.micro*) instances to help distinguish characteristics specific of *micro* VMs. For all VM instances, we use the default Amazon Machine Image (AMI) with Ubuntu Server 12.04 LTS in the *us-east-1a* (Virginia) availability zone.

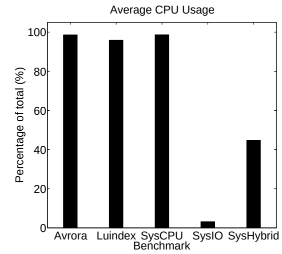
B. General Characterization Results

First, we characterize the resource usage of each benchmark to better understand the different resource requirement on *small* instances. The benchmarks are run for 1 hour using a single EC2 *small* instance. We run *avrora*, *luindex*, *sysbench CPU*, *sysbench IO*, and *sysbench hybrid* and measure their response time (i.e., runtime). We collect the CPU and I/O time using the `sar` utility. Figure 2(a) shows the overall response time split in its CPU and I/O time components for each benchmark. In this diagram, we assume the difference between the execution time and CPU time to be the I/O time.[2] The results indicate that the time spent on I/O for *avrora*, *luindex*, and *sysbench CPU* is so small that is hardly visible on the respective bars. Figure 2(b) presents the CPU utilization distribution. The CPU utilization of *avrora*, *luindex*, and *sysbench CPU* log at the 90% to the 95% level, while *sysbench IO* is as low as 5%. For *sysbench hybrid* on the other hand, this measure becomes 40%. System I/O read and write amounts are given in Figure 2(c) and Figure 2(d), respectively. *Sysbench IO* and *sysbench hybrid* have moderate I/O read operations and significant I/O write operations.

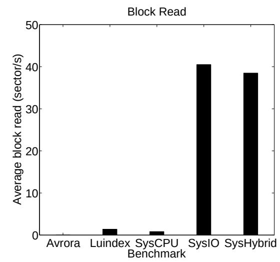
[2] We do this because both *micro* and *small* are configured with 1 vCPU and we configure *sysbench* benchmarks to run with single thread thus CPU and I/O time should be interleaved. These values might not be accurate for DaCapo benchmarks since the may be multi-threaded. However, there is not much disk activity for *avrora* and *luindex*.



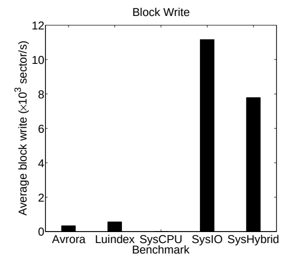
(a) Response time (CPU and I/O)



(b) CPU utilization



(c) Block read



(d) Block write

Fig. 2: Benchmark characterization on *small* VMs

These baseline experiments show that *avrora*, *luindex*, and *sysbench CPU* have very limited I/O demand but have very high CPU utilization. Yet, they have clearly different average runtimes, thus providing different scenarios for the evaluation of throughput. Such differences can be attributed, for example, to different cache behavior and internal multi-threading [6]. *Sysbench IO* is I/O-bound and we found that most of the time the CPU is waiting for I/O. *Sysbench hybrid* represents a “balanced" workload that spends half of its time in CPU and half in I/O. These benchmarks are then ideal for our study as they offer simplicity of interpretation of the experimental results and at the same time cover a broad enough workload spectrum. Although an analysis of workloads that are cache/memory bound or bandwidth intensive is also needed, we defer their analysis as part of our future work.

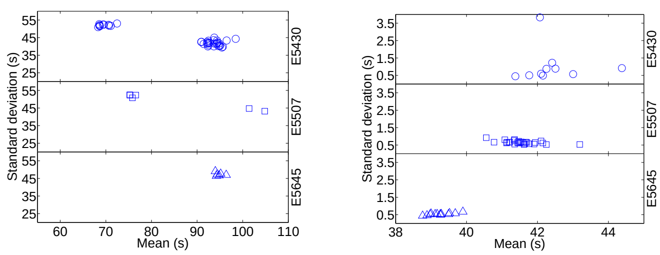
III. Workload Characterization

We are interested in describing the relationship between CPU throttling, performance, costs, and the effects of artificially injected delays that are equal to 0 seconds (no delay), and 10, 30, 60, and 90 seconds. For each choice of artificial delay, including the case of no-delay, we start simultaneously $m = 50$ spot instances for each benchmark. Thus, the resulting dataset amounts to 250 instance runs of 6 hours for each benchmark and choice of delay, for a total of $1,250$ experiments and $7,500$ hours. Our bid was sufficiently high to make sure that no spot instances were terminated by EC2 before the completion of the 6 hours period. The goal is to provide a statistical characterization of these results. Characterizing these properties requires to consider the time dimension, since throttling is amplified over time as reported in the official documentation [7, page 115–117].

A. Time and Heterogeneity Effects on Performance

Previous work on EC2 has highlighted how the heterogeneity of hardware characteristics is a source of performance variability [8]. But only marginally addressed the *t1.micro* class. In our experiments, we observe the performance effects of different hardware in *micro* and *small* instances across all benchmarks, suggesting that also the placement of *micro* VMs suffers from hardware heterogeneity.

Figure 3 illustrates the mean execution times and standard deviation from a 6-hour run on 50 spot instances of *avrora*. These 50 instances are allocated on different hardware (marked on the graph: E5645, E5507, and E5430, note that here we have three "stacked" graphs to ease comparison across different hardware). The graphs illustrate the mean execution time of *avrora* within each instance and its standard deviation. Across both *small* and *micro* instances, the effect of different hardware is strongly reflected on the mean values. The effect of CPU scheduling is reflected on the standard deviation: the values for *small* instances are very small, while for *micro* instances are very high. In addition, for *micro* instances we observe different clusters as defined by the mean execution time that can be almost 50% higher from cluster to cluster, even within the same hardware (see for example Figure 3a for E5430).



(a) *Micro* VMs

(b) *Small* VMs

Fig. 3: Performance results of 50 spot instances *avrora* on different hardware.

For illustrative purposes we also show representative experiments by plotting the moving average with a window size of 20 execution points across time. For *micro* and *small* instances, see Figures 4 and 5 respectively. The plotted values clearly illustrate the performance heterogeneity (the three selected experiments in each plot come from different hardware). If we had not plotted moving average values, we would have obtained a very jaggy plot for the *micro* case, where fluctuations are rapid as shown in Figure 1. As expected, for CPU intensive workloads, the longer the execution of the experiment, the worse the performance, irrespectively of the assigned hardware. On *small* instances, performance is stable across time (see Figure 5), with values been distinguished only by the hardware speeds. Both CPU and I/O intensive workloads have predictable and stable performance across the entire experiment.

The graphs in Figures 4 and 5 are just illustrative examples. A more systematic characterization is provided in Table I, which shows the $E[X_h^{small}]$ and $E[X_h^{micro}]$ values for $h = 1, 3, 6$ hours. The results confirm that as time passes the throughput of *micro* VMs is monotonically decreasing. Table II illustrates the results for *small* instances. Since performance is stable, we report a single hourly value. By comparison with Table I, it is interesting to see that the average performance of a *small* instance in 1 hour is matched by a *micro* instance in a variable timespan between 1 and 3 hours.

Table I and Table II also include columns for the CPU utilization steal percentage. This is the percentage of time where the VM could not use the host CPUs due to the hypervisor scheduling other VMs on it. As expected, *micro* VMs experience massive CPU utilization stealing, with the percentages being in the range 67%-81% and the standard deviation intervals suggesting that there are frequent periods where this peaks in a neighborhood[3] of 100%. Notice that moving from $h = 1$ to $h = 3$ there is a clear increase in the steal percentage that also grows, but slower, from $h = 3$ to $h = 6$; this provides some characterization of the time degradation of the CPU capacity for a continuously operating application.

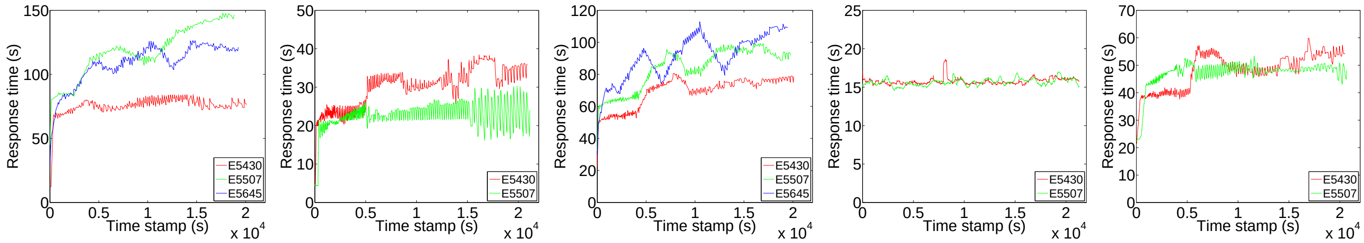
B. Static Delay Characterization

In this section we investigate whether it is possible to harness better performance by enforcing a certain amount of idle time on the *micro* VM CPUs to decelerate the rate of throttling. To this end, after each benchmark execution, we issue a sleep call to keep the *micro* VM idle for a fixed time before starting the next execution of the benchmark.

Figure 6 show the execution time CDF of each benchmark with different static delay values. Since we do not have any control on the assigned hardware by EC2, we opt to present results across all 50 instances in the form of a CDF. The collected benchmark response times correspond to measurements during the first hour of each VM instance. For CPU intensive workloads (see Figure 6(a), (b), (c), and partially (e)), adding delays between consecutive executions is beneficial: the tails of response times dramatically reduce, as also the mean execution times (marked with a diamond on each CDF), is significantly reduced. The longer the delay time, the further the execution time improves, and this effect is consistent across benchmarks. For I/O intensive workloads, see Figure 6(d), adding delays does not consistently help reducing the execution time. This is expected since the host throttles the CPU. However, we observe a static delay of 10s to result in slightly better performance. It is unclear if the improvement in this experiment is due to different hardware placement or to some improvement at the CPU level (e.g., resulting in decreased I/O handling time by the CPU) due to the injected delay. Across workloads, performance changes are significant enough to be attributed to the injected delays.

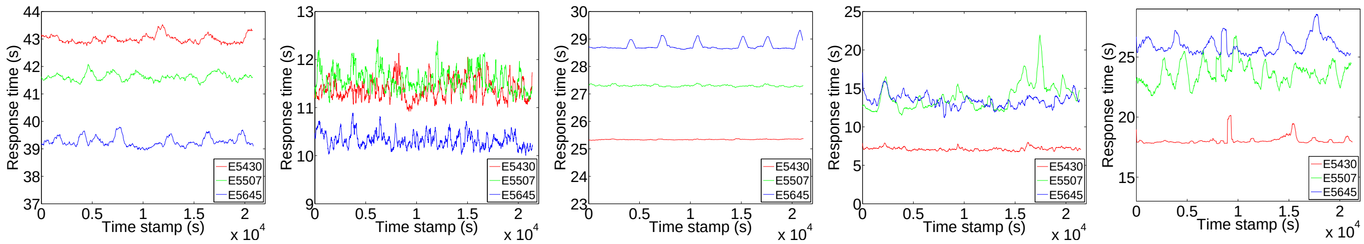
Figure 6 reports on the individual execution times but these times do not contain the VM sleep time between subsequent benchmark executions. Throughput, on the other hand, as a measure, encompasses the sleep times since it provides how many benchmarks completed execution per time unit. We compare throughput on *micro*s with delay and throughput without delay by calculating relative throughput which we define as $TPUT_{delay}^{micro}/TPUT_0^{micro}$, where $TPUT_0^{micro}$ is the average throughput on *micro* instances without delay. According to the above definition, the larger the relative throughput value, the better the performance. Figure 7 shows the relative throughput across the duration of the experiment. For some benchmarks, adding delay values can maintain or improve the overall system throughput. For *avrora*, see Figure 7(a), adding 10 seconds

[3] The fact that some standard deviations added to the means would *slightly* exceed 100% may be attributed to small measuring inaccuracies, note also that such effect is present only for the CPU intensive workloads; indeed the CPU steal value is upper bounded by 100%.



(a) *Avrora* (b) *Luindex* (c) *Sysbench CPU* (d) *Sysbench IO* (e) *Sysbench hybrid*

Fig. 4: Response time on *micro* VMs (moving average).



(a) *Avrora* (b) *Luindex* (c) *Sysbench CPU* (d) *Sysbench IO* (e) *Sysbench hybrid*

Fig. 5: Response time on *small* VMs (moving average)

TABLE I: Throughput $mean \pm std$ of completed jobs per h hours on *t1.micro*

	$h=1$		$h=3$			$h=6$		
Benchmark	$E[X_h^{micro}] \equiv TPUT_h^{micro}$	*CPU steal*	$E[X_h^{micro}]$	$TPUT_h^{micro}$	*CPU steal*	$E[X_h^{micro}]$	$TPUT_h^{micro}$	*CPU steal*
Avrora	62.5 ± 7.6	$73.8\% \pm 26.9\%$	139.9 ± 17.5	46.6 ± 5.8	$78.7\% \pm 21.7\%$	211.8 ± 29.9	35.3 ± 5.0	$80.2\% \pm 19.9\%$
Luindex	194.2 ± 23.3	$73.1\% \pm 27.4\%$	449.8 ± 53.8	149.9 ± 17.9	$76.3\% \pm 24.0\%$	692.0 ± 92.8	115.3 ± 15.5	$79.9\% \pm 20.1\%$
SysCPU	86.7 ± 11.1	$66.5\% \pm 34.5\%$	201.1 ± 25.8	67.0 ± 8.6	$70.3\% \pm 30.4\%$	361.0 ± 53	58 ± 7.6	$71.8\% \pm 28.8\%$
SysIO	227.3 ± 11.9	$1.06\% \pm 2.79\%$	678.7 ± 35.0	226.2 ± 11.7	$1.08\% \pm 2.88\%$	1358.9 ± 66.9	226.5 ± 11.2	$1.08\% \pm 2.92\%$
SysHybrid	103.5 ± 10.1	$39.7\% \pm 13.5\%$	261.3 ± 26.8	87.1 ± 8.9	$41.5\% \pm 12.4\%$	483.5 ± 58.5	80.6 ± 9.8	$41.9\% \pm 11.9\%$

TABLE II: Throughput $mean \pm std$ of completed jobs per h hours on *m1.small*, (throughput values across different h are the same).

Avrora		*Luindex*		*SysCPU*		*SysIO*		*SysHybrid*	
$TPUT_h^{small}$	*CPU steal*	$TPUT_h^{small}$	*CPU steal*	$TPUT_h^{small}$	*CPU steal*	$TPUT_h^{small}$	*CPU steal*	$TPUT_h^{small}$	*CPU steal*
87.7 ± 2.8	$55.6\% \pm 3.7\%$	322.2 ± 18.5	$54.1\% \pm 4.6\%$	132.5 ± 5.8	$43.9\% \pm 21.3\%$	310.4 ± 77.5	$2.0\% \pm 1.1\%$	172.6 ± 20.0	$22.6\% \pm 2.8\%$

delay can maintain nearly the same system throughput as in the no delay case. For *luindex*, adding 10 seconds delay increases the overall throughput in the 1h to 6h duration time (see Figure 7(b)), same for *sysbench CPU* and *sysbench hybrid*. For I/O intensive workloads such as *sysbench IO*, adding delays does not improve the system throughput as expected, see Figure 7(d). Similarly, if the delay is long (e.g., 90 sec), then throughput is bound to be poor. The last conclusion holds irrespectively of the number of hours of the experiment.

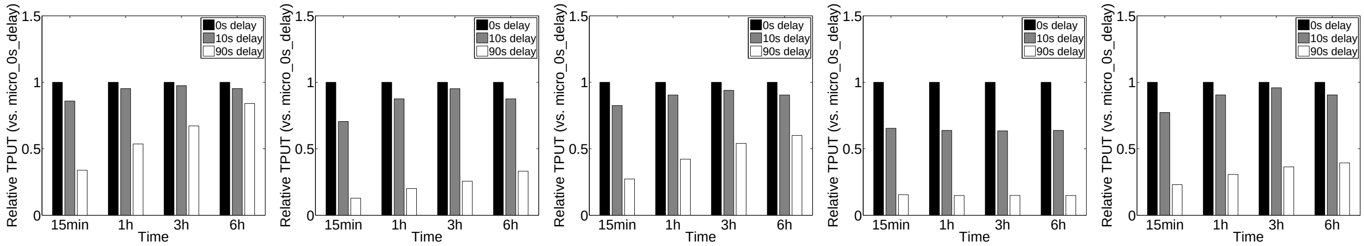
The analysis in this section shows that injecting delays can help performance across a timespan of hours. It dramatically reduces the response time tails (as well as response time means, especially for CPU intensive workloads such as *luindex*) while maintaining (in cases) almost the same throughput as the no delay scenario. Ideally, we need to strike a balance on selecting an ideal delay such that it reduces average execution time while maintaining high throughput.

IV. Algorithm Design

In this section, we focus on designing adaptive algorithms for deciding the optimal delay to be injected for an application running inside a *micro* VM. We do not take any specific assumption on the workload characteristics, except for the ability to periodically monitor the execution "progress" of the application during a control window. We focus on two measures: the application response time and system throughput. Some of the adaptive algorithms are based on throughput, thus if jobs have long response times the update of the throughput value to reflect this may take a long time. This may impact negatively on the management algorithm performance. Thus, we assume that the application offers a mechanism that allows one to monitor the progress of currently running jobs.

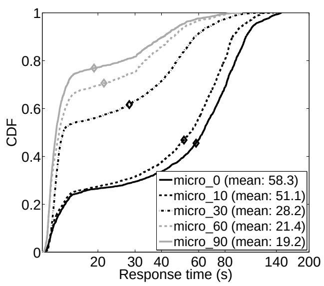
A. Stochastic Approximation

The stochastic approximation (SA) algorithm allows to statistically maximize a quantity (e.g., system throughput) online subject to noise and it is popular in control theory [9]. We used SA to define and implement Algorithm 1 which aims at maximizing the system throughput. The purpose is to derive the ideal value of the current delay *cur_delay* in an iterative manner. The algorithm depends on two parameter sequences, a_k and c_k, see lines 4 and 5 in Algorithm 1, that depend on the iteration number k, these values are suggested in the original paper [9]. For each SA iteration, the algorithm

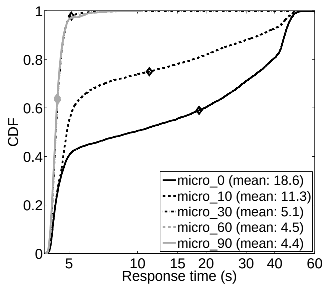


(a) *Avrora* (b) *Luindex* (c) *Sysbench CPU* (d) *Sysbench IO* (e) *Sysbench hybrid*

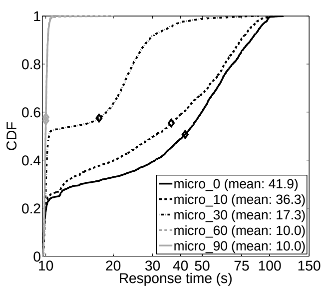
Fig. 7: Relative throughput on *micro* VMs



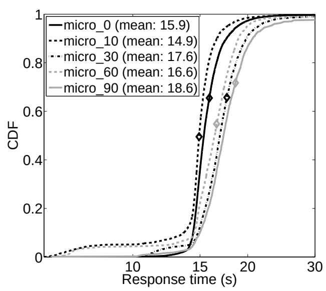
(a) *Avrora*



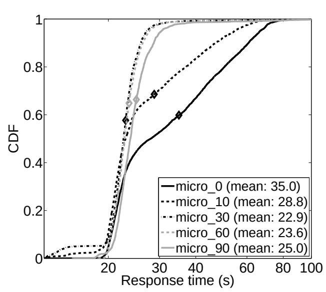
(b) *Luindex*



(c) *Sysbench CPU*



(d) *Sysbench IO*



(e) *Sysbench hybrid*

Fig. 6: Response time CDF on *micro* VMs with delays within one hour, the x-axis is in log scale.

executes jobs in two *consecutive* time windows and records the achievable throughput in each. Jobs execute with a different delay value in each window, see lines 6 and 7 in algorithm 1. The variables *cur_delay* and *delta* hold the delay in the current window and the magnitude of allowed delay change. The function *runjobwithdelay*() runs jobs in each window and returns the number of finished jobs X. Based on these values, the algorithm updates *cur_delay* based on the difference of the number of finished jobs in the two windows (see line 8). Note that it is possible for the difference of X_+ and X_- value to result in a negative number, this suggests that the throughput with a smaller delay is better, therefore it will be advantageous to reduce the delay in the next iteration. If however the computed new delay value (see line 9) is negative, then the jobs are scheduled with no delay, although the computed delay value retains its value for the next iteration where it is again adjusted. Convergence rate depends on some regularity conditions for X, however in general we noticed SA to converge slowly.

Algorithm 1: Stochastic Approximation

1 $cur\_delay \leftarrow C$;
2 $k \leftarrow 1$;
3 **while** true **do**
4 $\quad a_k \leftarrow 1/k$;
5 $\quad c_k \leftarrow k^{-1/3}$;
6 $\quad X_+ \leftarrow$ `run_job_with_delay`$(cur\_delay + delta, window)$;
7 $\quad X_- \leftarrow$ `run_job_with_delay`$(cur\_delay - delta, window)$;
8 $\quad cur\_delay \leftarrow cur\_delay + a_k(X_+ - X_-)/c_k$;
9 $\quad k{+}{+}$;

For the experiments presented in the following section we set *delta* equal to 5 seconds and *window* equal to 120 seconds. We selected these values after experimenting with several options, which resulted in varying degrees of reactive adjustment to the delay value. The obtained values are those that provided the best results for SA in our experiments.

B. *Adaptive Micro-Management (AMM)*

Adaptive micro-management (AMM) is a new algorithm we propose for managing *micro* instances. AMM is a gradient-hill method for continuously updating the injected delay in a *micro* VM. Several strategies are possible to compute gradients online. The idea pursued here is to consider control windows and explicitly compute gradient values by dynamically altering the delay within successive sub-windows. Another idea is to continuously probe the application and start and stop delays instantaneously based on observations. We tested these ideas, but we only found effective the AMM approach described in this section. Due to limited space, we do not document these parallel efforts.

The AMM algorithm, summarized in Algorithm 2, determines at runtime the delay to inject in a *micro* VM and can be either throughput driven or response time driven. The algorithm automatically injects a delay between two consecutive job executions. In our implementation, this is done with simple sleep functions, but in a general scenario it needs a *cgroups* [10] implementation or explicit coordination between the controller and the application.

The control window is initially divided into three sub-control windows (line 1). The idea is to continuously compute the gradient of the throughput (or response time) by making small changes at runtime of the delay value and updating the delay itself based on the best throughput (or response time) observed. To achieve this, we maintain a global variable *cur_delay* that holds the delay of the current window and a variable *delta* that represents the magnitude of allowed delay change for the sub-windows. The algorithm uses three different delay times: $cur\_delay + delta$, *cur_delay*, and $cur\_delay - delta$ in the three sub-windows. The idea is to evaluate the change in throughput (or response time) following from a *delta* change of delay and accept the modification that provides the best result. To do this, $run\_job\_with\_delay()$ runs during each sub-window and returns the number of completed jobs for the throughput version or returns the average response time for the response time version, see line 6 in Algorithm 2. At the end of each *window*, the algorithm sets the next *cur_delay* value to the delay for which *get_delay* (see Algorithm 3) records the best throughput (tput) or response time (rt).

Algorithm 2: AMM (tput/rt) Algorithm Pseudocode

```
1 sub_win ← window/3;
2 while true do
3     /* results: num_of_jobs for the tput version, avg_rt for
      the rt version */
4     results[3] ← {0};
5     for i=0, 1, 2 do
6         results[i] ← run_job_with_delay(cur_delay +
          (1 − i) ∗ delta, sub_win);
7     cur_delay ← get_delay(cur_delay, results, delta);
```

Algorithm 3: get_delay(*cur_delay*, *results*, *delta*)

```
  input  : current delay cur_delay, results array results, delta
           value delta
  output: Delay value for next round
1 /* Execute the following line only for the tput version */
2 value ← max (results[0], results[1], results[2]);
3 /* Execute the following line only for the rt version */
4 value ← min (results[0], results[1], results[2]);
5 for i=0, 1, 2 do
6     if value = results[i] then return
      cur_delay + (1 − i) ∗ delta;
```

Similarly to the SA algorithm, we experimented with different values for *window* and *delta*. The range of values we considered was $\{5s, 10s, 20s, 30s\}$ for *delta* and $\{30s, 60s, 300s, 600s\}$ for *window*. The analysis was repeated for *Avrora*, *Luindex*, *Sysbench CPU*, and *Sysbench IO*; due to limited space we do not report these experiments. Our results indicated that the optimal value of these parameters depends on the benchmark used, but the combination $delta = 10s$ and $window = 60s$ produced consistently good results across benchmarks. In particular, we noticed that larger *window* values tend to reduce the throughput gains compared to the no delay case, whereas *delta* values of $5s$ or $30s$ can occasionally yield bad results. Even though we recommend $delta = 10s$ and $window = 60s$ as default parameters for AMM, we suggest in general to perform a sensitivity analysis to establish the optimality of these values on the specific workload used.

V. Performance Evaluation

To evaluate the effectiveness of the algorithm, we run AMM and SA on all five reference benchmarks on 50 VM instances for a total execution time of 6 hours for each benchmark. We present CDFs of the achieved response times for all benchmarks. Figure 8 presents the expected response time per benchmark execution during the first hour of the experiment. Results throughout the entire period are very similar (i.e., the relative performance ranking of policies remains the same as in the first hour) and not presented here due to lack of space. In addition to the AMM and SA results on *micro*s, we also report results achieved on *micro* VMs with no delay and delays equal to 10 and 90, as well as on results with *small* VMs. On each CDF line we also mark the average value with a diamond (averages are also reported on the legend).

Figure 8 clearly illustrates that *small* VMs display consistent results across all benchmarks, with the only exception of *sysbench IO*. Across nearly all benchmarks (with the exception of *sysbench IO* and *sysbench hybrid*), the AMM response time version achieves nearly the same average as the one with the *small* VMs. Surprisingly, we also see a significant portion of jobs ranging from 40% (see *avrora*) to 65% (see *luindex*) where the response time is significantly less than the one of *micro*s. These values are consistently *half as much* as those of *small* VMs, at the expense of longer tails.

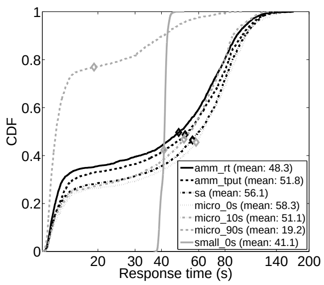
Across all experiments we see consistently that AMM (its response time version) achieves CDFs that lie between those of *micro* with no delay and *micro* with 90 delay. Naturally, for CPU intensive workloads, experiments on *micro*s with large delays of 90 seconds remains overall very competitive with respect to response time but do poorly with respect to throughput, while for I/O intensive workloads we observe slower response times than in the *small* instances.

For many benchmarks, SA is marginally better than *micro* with no delay. Indeed, after inspection, we see that SA converges in some periods to negative delays, which we handle by injecting no delay. However, this affects the reactiveness of the method since it may take a longer time before SA returns to positive delays. Indeed, one may force this by artificially limiting the delay to remain non-negative, but it is unclear how this changes the properties of the general SA algorithm. We left this extension to future work.

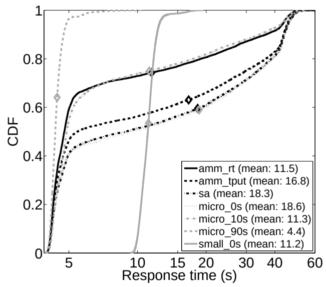
Overall, Figure 8 illustrates that the AMM is a very effective algorithm: its online adjustment of delay is effective and results in superior performance for a large percentage of jobs for all CPU intensive benchmarks, it approaches the performance of *small* for the balanced *sysbench hybrid* case and does as well as any fixed delay algorithm for *micro*s for *sysbench IO*.

Figure 9 plots the throughput [4] for $h = 1$ and $h = 3$ hours,

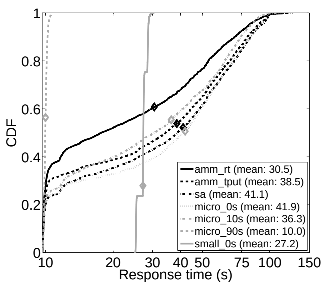
[4] For fairness, we divide the *small* throughput by 2.2 since it is the ratio of the hourly price of *small* to *micro*. This scaling ensures that the throughput is per unit cost, where we assume the price of a *micro* instance as the unit of reference.



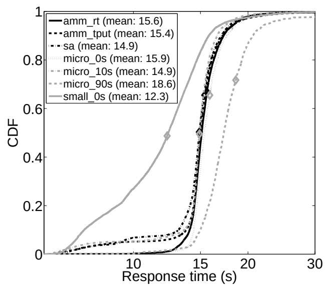
(a) *Avrora*



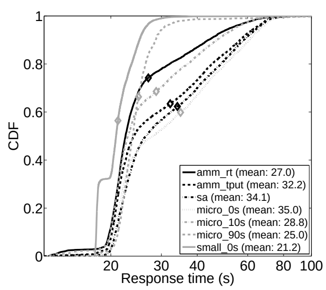
(b) *Luindex*



(c) *Sysbench CPU*



(d) *Sysbench IO*



(e) *Sysbench hybrid*

Fig. 8: Response time CDF within one hour, the x-axis is in log scale.

we observed that the longer the experiment, the worse the throughput of the *micro*s. Since trends tend to be monotonic, we limit to illustrate results for $h = 1$ and $h = 3$. Indeed SA results in conservative delays and approximates closely the throughput achieved by *micro* with no delay. This is also an immediate effect of the fact that the SA algorithm allows for negative delays (which we handle by injecting no delay at all) which results in a slower time to reach positive (i.e., actual) delays, benefiting throughput. The advantage of the AMM throughput version is also clearly illustrated across all experiments. It does almost as well as SA, which is another throughput-oriented algorithm.

Finally, Figure 10 illustrates yet another way to view the compromise among two conflicting measures. Here, we plot the ratios of average throughput over average response times, in an effort to capture both measures within one numeric value. We see that indeed the proposed adaptive algorithms (both AMM and SA) do at least as well as the *small* instances, with the response time version of AMM doing better.

VI. Related Work

Performance heterogeneity across different instance types in Amazon EC2 has been studied and explored in several works [8], [11]. Ou et al. [12] exploited hardware heterogeneity and its corresponding performance variation *within* the same type of VMs on EC2. Farley et al. [13] confirm that performance heterogeneity exists across supposedly equivalent instances and propose a placement gaming strategy to seek out better performing VMs.

Xu et al. [14] studied the long tail performance problem of Amazon EC2 instances and found that often long tails are due to co-scheduling CPU-bound and latency-sensitive tasks on the same node. The performance overhead due to virtualization on EC2 has been determined as the main culprit of TPC/UDP throughput instability and delay variations in bandwidth sensitive applications rather than the network load. Mao and Humphrey [15] studied the startup time of cloud VMs across Amazon EC2, Windows Azure, and Rackspace and analyzed the relationship between the VM startup time and different factors such as time of the day, OS image size, instance type, data center location and the number of instances.

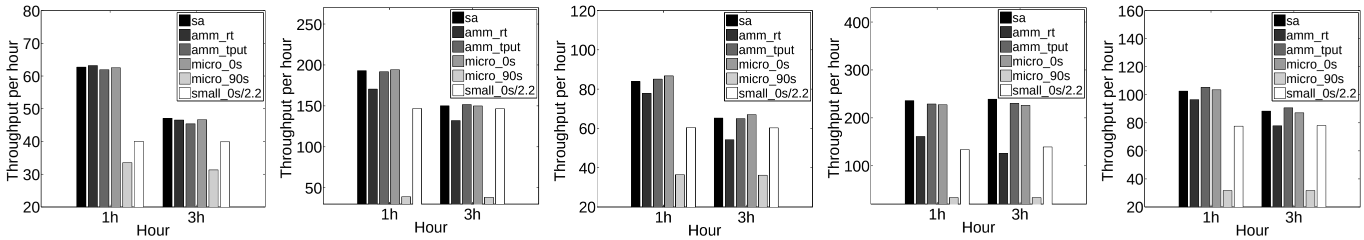
Walker [16] studied the performance of Amazon EC2 against a local equivalent processors cluster. The performance disadvantages of public clouds for parallel and scientific computing in comparison to grids and other parallel computing infrastructures have been documented in [17]. Optimizing cluster sizes across a range of workloads and goals via tools that can leverage residual or unused resources due to over-provisioning is proposed by [18]. Zhang et al. designed an evaluation framework that focuses on evaluating and selecting of different available underlying cloud computing platforms (e.g. *small*, *medium*, or *large* EC2 instances) and achieving desirable Service Level Objectives (SLOs) for MapReduce workloads [19].

Song et al. [20] design an auction mechanism for the data center spot market (DCSM). This mechanism is proved truthful (i.e., bidders cannot bid for the same instance using different price and cannot obtain a fraction of requested instances) and is based on a repeated uniform-price auction. Bidding flexibility is also incorporated such that bidders are able to change their bids after obtaining instances. Experimental results show that this proposed mechanism can outperform Amazon Spot Market (ASM) in all of the above four metrics.

Similar to the above studies, our work focuses on how to reduce the well-documented long tails on *micro*s [21]. To the best of our knowledge, besides the works that documented high variability in execution times of *micro* instances on EC2, no study exists that focuses on how to take best advantage of the current scheduling of *micro* instances to reduce response time tails while maintaining high throughput. The scheduling algorithms that we propose, which run at the user level and do not require any system changes, offer more consistent performance for *micro* instances, which are notorious for their capacity fluctuations.

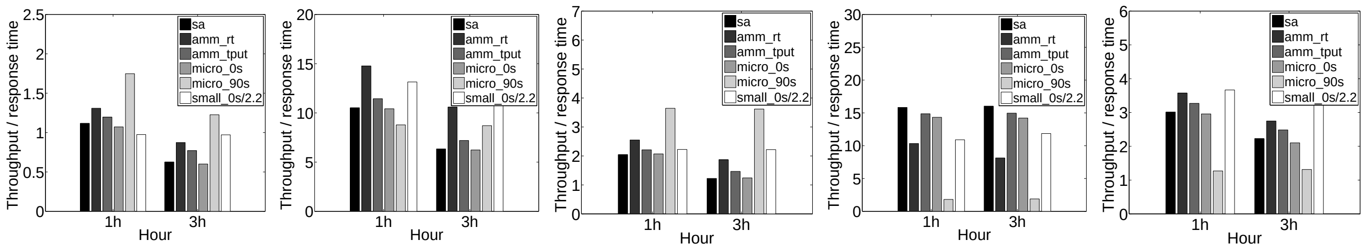
VII. Conclusion and Future Work

In this paper, we have studied the *t1.micro* VM instance offering of Amazon EC2. We have investigated experimentally



(a) *Avrora* (b) *Luindex* (c) *Sysbench CPU* (d) *Sysbench IO* (e) *Sysbench hybrid*

Fig. 9: Actual throughput



(a) *Avrora* (b) *Luindex* (c) *Sysbench CPU* (d) *Sysbench IO* (e) *Sysbench hybrid*

Fig. 10: Actual throughput / mean response time

the injection of artificial delays to optimize the performance and cost usage of *micro* VMs. By comparing different possible strategies, we have found that gradient-like approaches provide a good solution that is simple to implement.

Since the time these experiments were done, Amazon introduced the new class *t2* which provides a throttling mechanism for *small* and *medium* instances. Our preliminary results show that our algorithms perform also well on *t2*. We are currently investigating the effectiveness of our algorithms on the performance of bandwidth and cache/memory intensive benchmarks.

ACKNOWLEDGMENT

This work received funding from the MODAClouds project (FP7-318484) and the National Science Foundation grants CCF-0937925 and CCF-1218758.

REFERENCES

[1] "Amazon Elastic Compute Cloud," `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html`.

[2] "Amazon EC2 Pricing," `http://aws.amazonaws.com/ec2/pricing`.

[3] S. M. Blackburn et al., "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceeding of OOPSLA'06*, New York, NY, 2006, pp. 169–190.

[4] A. Kopytov, "Sysbench: a system performance benchmark," 2004.

[5] "Apache Lucene," `http://lucene.apache.org`.

[6] H. Cook et al., "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proceedings of ISCA '13*, New York, NY, 2013, pp. 308–319.

[7] "Amazon Elastic Compute Cloud User Guide," `http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf`.

[8] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, 2010.

[9] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *AOMS*, vol. 23, no. 3, pp. 462–466, 1952.

[10] P. Menage, "Control Groups Documentation," `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`.

[11] H. Zhuang, "Performance evaluation of virtualization in cloud data center," Ph.D. dissertation, KTH, 2012.

[12] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui, "Exploiting hardware heterogeneity within the same instance type of amazon ec2," in *Proceeding of HotCloud'12*, Berkeley, CA, 2012, pp. 4–4.

[13] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: exploiting performance heterogeneity in public clouds," in *Proceeding of SoCC'12*, New York, NY, 2012, pp. 20:1–20:14.

[14] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: avoiding long tails in the cloud," in *Proceedings of NSDI'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 329–342.

[15] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Proceeding of CLOUD '12*, Washington, DC, 2012, pp. 423–430.

[16] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," *Usenix Login*, vol. 33, no. 5, pp. 18–23, 2008.

[17] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 931–945, 2011.

[18] B. Clay, Z. Shen, and X. Ma, "Accelerating batch analytics with residual resources from interactive clouds," in *Proceeding of MASCOTS'13*, San Francisco, CA, 2013.

[19] Z. Zhang, L. Cherkasova, and B. T. Loo, "Optimizing cost and performance trade-offs for mapreduce job processing in the cloud," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, May 2014.

[20] K. Song, Y. Yao, and L. Golubchik, "Improving the revenue, efficiency and reliability in data center spot market: A truthful mechanism," in *Proceeding of MASCOTS'13*, San Francisco, CA, 2013.

[21] K. Raaen, A. Petlund, and P. Halvorsen, "Is today's public cloud suited to deploy hardcore realtime services?" in *LNCS - Lecture Notes in Computer Science*, Euro-Par LSDVE 2013. Springer, 2013.