How to Supercharge the Amazon T2: Observations and Suggestions

Jiawei Wen, Lihua Ren
College of William and Mary
Williamsburg, VA, USA
{jwen01, lren01}@email.wm.edu

Feng Yan
University of Nevada
Reno, NV, USA
fyan@unr.edu

Daniel J. Dubois, Giuliano Casale
Imperial College London
London, UK
{daniel.dubois, g.casale}@imperial.ac.uk

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

***Abstract*—Cloud service providers adopt a credit system to allow users to obtain periods of performance bursts without additional cost. For example, the Amazon EC2 T2 instance offers low baseline performance and the capability to achieve short periods of high performance using CPU credits. Once a T2 instance is created and assigned some initial credits, while its CPU utilization is above the baseline threshold, there is a transient period where performance is boosted and the assigned CPU credits are used. After all credits are used, the maximum achievable performance drops to baseline. Credits accrue periodically, when the instance utilization is below the baseline threshold. This paper proposes a methodology to increase the performance benefits of T2 by seamlessly extending the duration of the transient period while maintaining high performance. This extension of the high performance transient period is combined with proactive migration to further take advantage of the initially assigned credits. We conduct experiments to demonstrate the benefits of this methodology for both single-tier and multi-tier applications.**

I. Introduction

Cloud computing [1], [2] has become nowadays a well established paradigm [3], [4]. Many cloud platforms, such as Amazon EC2 [5], Google Compute Engine [6], and Microsoft Azure [7], offer flexible cloud computing services, enabling cloud users to quickly launch jobs by requesting the desired amount of resources through the Internet without maintaining their own hardware, paying only for the need-based resources.

As a widely-used cloud platform, Amazon EC2 has been extensively studied [8], [9], [10], [11], [12], [13], [14]. These studies range from comparing the performance of EC2's offerings to other cloud platforms, to the peculiarities of specific instance offerings of EC2, and to performance comparisons when applications (single-tier and multi-tier) are migrated from a traditional data-center environment to Amazon EC2.

The focus on this paper is on the most opportune usage of Amazon EC2's T2 instance types. T2 instances are initially allocated CPU credits, which can be spent to allow bursts of performance above the baseline, and periodically accrued then the load is under the baseline. These instances are designed for applications that do not pose consistent demands to CPU resources, for example, web services characterized by short bursts of heavy load. In this work we focus not only on making the most efficient use of T2 instances for this type of cloud services, but also for cloud services that require non-bursty, i.e., consistent CPU demands.

We consider two types of benchmarks: single tier benchmarks with different intensities in CPU demands and TPC-W, a multi-tier application, that is a well accepted transactional web benchmark that simulates a business oriented transactional web server [15]. For the single-tier benchmarks, we show that by using `cpulimit` [16] it is possible to increase the initial transient time period where CPU is used well-above its baseline T2 performance. It is possible to extend this transient period of high performance up to five times at the beginning of the instance launch. For such cloud services, it is possible to estimate the duration of this transient period where performance is sustained at a much higher level than baseline and estimate when CPU throttling stabilizes and periodic bursty allocation starts. For the multi-tier case, we also observe that the use of `cpulimit` can be tremendously beneficial for user-perceived performance, but the duration of the transient, high performance period cannot be easily predicted. Yet, in both cases we show that for applications that execute for a short term (up to several hours for the case of TPC-W), it is possible to achieve superior performance than the baseline one offered by T2. Beyond the transient period, where the CPU is throttled by the burstiness mechanism, the improvements of `cpulimit` are small. This makes us consider proactive migration, i.e., enable a checkpoint/restart mechanism by launching a new instance when the steady-state low performance period starts. Previous work [17] illustrates how to launch several *t2.micro* instances across different locations to avoid the CPU throttling penalty for long-duration jobs, but its major shortcoming remains the migration frequency, which can result in significant performance penalties. In this paper we show that `cpulimit` can significantly reduce the migration frequency of long-running jobs on T2. Our experiments demonstrate that we can effectively reduce the migration frequency up to 80% and that the proposed proactive approach enables a seamless migration scheme.

The rest of the paper is organized as follows: we first introduce some necessary background about T2 instance in Section II. Then we conduct workload characterization to understand the relationship between application performance and the variation of CPU credits in Section III. Based on the observations from Section III, we propose our `cpulimit` and proactive *migration* approach in Section IV. In Section V, we evaluate the proposed approach. We discuss related works in Section VI. Finally, we conclude this study in Section VII.

II. BACKGROUND

An Amazon EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (EC2) for running applications on the Amazon Web Services (AWS) infrastructure. AWS offers a host of different types of instances, each optimized for different user needs. T2 instances are “burstable performance instances" that provide a baseline CPU performance which can be temporarily increased in case of need. T2 instances include *t2.nano*, *t2.micro*, *t2.small*, *t2.medium*, and *t2.large* variations. The f1 and g1 instances in Google Compute Engine [6] are another example of burstable performance instances. In this paper, we focus on characterizing and making better use of *t2.micro* instances (as the algorithm of Google f1 and g1 instances is not disclosed).

According to the AWS documentation [18], T2's CPU performance is controlled by a mechanism based on CPU credits. One CPU credit provides the performance of a full CPU core for one minute, i.e., one CPU credit is equal to one vCPU running at 100% CPU utilization for one minute. The same rule also applies to other combinations of vCPUs, utilization, and time. Equivalently, one CPU credit allows one vCPU running at 50% utilization for two minutes. In addition, each newly started instance is granted a healthy initial CPU balance, which is increased at a fixed rate by continuously receiving credits according to the instance size. When a T2 instance uses CPU resource that is equal to or less than its baseline performance level, the difference between earned and spent CPU credits are stored in the credit balance for up to 24 hours. The 24 hours expiration mechanism establishes the cap.

We list the configurations of *t2.micro*, *t2.small* and *t2.medium* in Table I [18], including the initial CPU credit balance, the receiving rate of CPU credits, the baseline performance level as a percentage of a full core performance, and the maximum earned CPU credit balance. Using *t2.micro* as an example, assuming that 1 CPU credit can support 1-minute full performance, its initial CPU credit (which is 30 in this example) should be depleted in 30 minutes; the value of the columns *CPU credits earned per hour* and *Baseline performance* show that a *t2.micro* instance earns 6 CPU credits per hour, and *Baseline performance*, the maximum CPU utilization allowed; the ceiling amount of CPU credits of a *t2.micro* instance is 144, which is the number of credits earned in 24 hours when CPU utilization is 0%.

III. WORKLOAD CHARACTERIZATION

In this section, we characterize the performance of benchmark applications running on CPU-credit-based *t2.micro* instance and motivate our approach. We select three single-tier benchmarks: Hmmer [19] from SPEC CPU2006, Avrora [20] from the DaCapo, and Sysbench hybrid [9] customized from the Sysbench suite [21]. Since CPU credits on T2 are accrued/subtracted according to CPU usage, we select three benchmarks that differ in their intensity of CPU demand.

- **Hmmer** : a CPU intensive benchmark used for searching sequence databases for sequence homologs and for making sequence alignments using hidden Markov models. This is a very CPU intensive benchmark, with minimal IO usage.
- **Avrora** : a collection of programs that run on a grid of AVR micro controllers. CPU is the dominating resource again but its usage is less intensive comparing to Hmmer.
- **Sysbench hybrid** : a customized workload that spends nearly equal time on CPU and IO proposed in [9]. It performs prime number calculations and file operations. Essentially, it consists of a combination of the two standard Sysbench benchmarks: *Sysbench CPU* and *Sysbench IO* [21].

According to AWS, T2 instances are best suited for workloads that do not use CPU consistently, yet occasionally can take advantage of a CPU burst. T2 instances are therefore appropriate for general purpose workloads, including web servers and developer environments. Using the above benchmarks, we evaluate how is CPU alloted to them across time. We report on the *service rate* (requests/minute) provided by *t2.micro*, i.e., on the number of requests completed per minute. Figure 1 illustrates the service rate as a function of time and each point represents the average service rate for 5 minutes. We observe that there are two periods for each plot: a *credit depleting period* (CDP) and a *stable period* (SP). The credit depleting period starts from the moment an instance is launched to the point that the instance has used all its CPU credits. The stable period starts after the credit depleting period. Because the instance has depleted all the CPU credits, during its stable period, CPU-throttling occurs and only 10% of CPU utilization is allowed. Therefore, the average service rate during the stable period is significantly lower than during the credit depleting period. See, for example, how the curve significantly drops across all benchmarks and not just only for Hmmer which is the least suitable for T2. Another important observation is that the credit depleting period is short across all benchmarks (regardless whether it is CPU intensive or CPU/IO balanced, see the similarities across Hmmer and Sysbench). Therefore, migrating the application to a new instance after the credit depleting period [17] is an option that we intent to adopt, as the migration performance penalty may be tolerable compared to the low performance of the stable period. The figure also clearly illustrates the periodic performance bursts after the initial depleting period. This is because Amazon uses a conservative way to measure CPU utilization, so a small number of credits can still be earned during the baseline experiment. This makes the performance spike a bit when Amazon recalculates the performance periodically and detects a credit balance that is larger than zero.

To further understand the performance change in different periods, we plot the CDF of service times, i.e., the user perceived performance, for four half-hour windows, as shown in Figure 2. The figure illustrates the user-perceived performance due to CPU throttling as credits are depleted. Within each half hour period, service times can be diverse especially for Avrora and Sysbench, and deteriorate as time progresses.

IV. METHODOLOGY

In this section, we propose a methodology to maximize the “value" of CPU credits. One naive way is to introduce explicit time periods within the application (e.g., “sleep" periods) where CPU is not used. This action may delay credit depletion,

	Initial CPU credit	CPU credits earned per hour	Baseline performance	Maximum allowed CPU credit balance
t2.micro	30	6	10%	144
t2.small	30	12	20%	288
t2.medium	60	24	40%	576

TABLE I: Configuration information for *t2.micro*, *t2.small*, and *t2.medium* . The configure information includes initial CPU credit allocation received at launch, the rate at which CPU credits are received, the baseline performance level as a percentage of a full core performance, and the maximum allowed CPU credit balance that an instance can achieve.

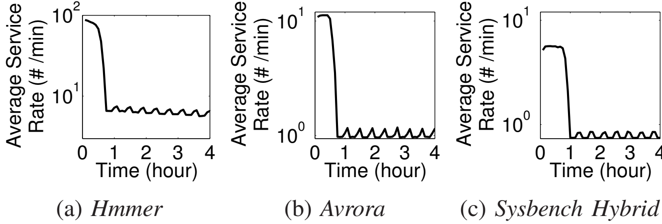


Fig. 1: Average service rate across time for different benchmarks using a *t2.micro* instance. The time window for computing average service rate is 5 minutes.

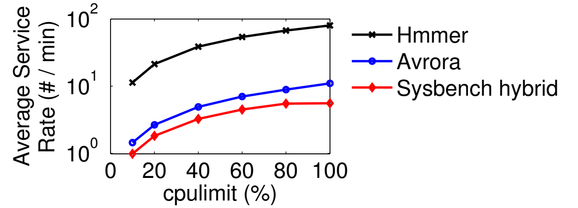


Fig. 3: Service rate as a function of `cpulimit` for different benchmarks. Note that `cpulimit` of 100% equals to the no `cpulimit` case, which is the case shown in Figure 1.

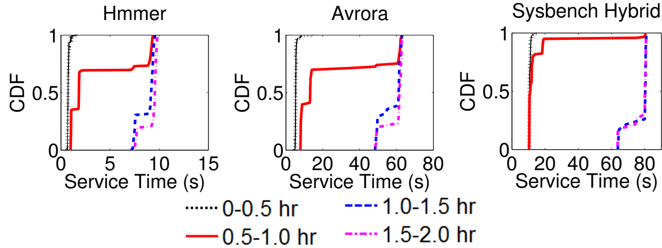


Fig. 2: CDF of service times for different benchmarks. The time window is 0.5 hours. There are 48 CDF plots for the entire 24-hour experiment. Here we only show the first four plots in the interest of space. The CDFs for half hour intervals beyond the $1.5-2$ hour window are very similar to the ones during the $1.5-2$ hour window.

but it often results in longer user end-to-end execution time [9]. Here, we first propose to use `cpulimit`, a tool which limits the CPU usage of a process (expressed in percentage, not in CPU time). The basic premise is to extend the credit depleting period for each instance via `cpulimit`, but after credits are depleted to renew them via the migration to a new instance.

A. Extending the Credit Depleting Period

We propose to extend the initial transient period by reducing credit consumption via judicious use of the `cpulimit` mechanism. In general, the more `cpulimit` reduces CPU consumption, the longer the extension of the credit depleting period. Yet, if this is done too aggressively, the performance benefits of reducing CPU usage diminish. For instance, if we limit CPU utilization at 10%, it is equivalent to the CPU throttling that Amazon enforces after all CPU credits have been used. Clearly, there is no performance benefit in this situation.

`Cpulimit` allows the user to quantify CPU usage, e.g., `cpulimit` = 90% means we limit the CPU utilization of a full CPU core to a max of 90%. In order to get a better understanding of how `cpulimit` impacts the performance of an application, we perform experiments with various `cpulimits` using the benchmarks described in Section III. We report the average service rate of credit depleting period under different `cpulimits` for different benchmarks, see Figure 3. For the CPU intensive application Hmmer, the service rate degradation is almost proportional to `cpulimit` reduction. For the less CPU intensive applications Avrora and Sysbench hybrid, the service rate degradation has a similar changing trend as Hmmer but smaller degradation rate when lower `cpulimit` is applied. In order to meet the performance requirement of cloud services while making the most of CPU credits, it is important to identify the optimal `cpulimit` that maximizes the credit depleting period while still maintaining an acceptable user response time. Since there are many factors that can impact the performance effect of `cpulimit` (e.g., instance type, application characteristics), it is not straight-forward to model accurately the performance change as a function of `cpulimit`. Here, we propose to build a performance reference table by running profiling experiments. One important observation from Figure 3 is that the curves are nearly linear, therefore it is possible to only run a few experiments and populate the reference table via linear interpolation. With the performance reference table, we can search the optimal `cpulimit` using a given performance requirement.

B. Proactive Migration

`Cpulimit` extends the credit depleting period but not infinitely, eventually throttling is going to be enforced by AWS. One natural way to get new CPU credits is to migrate to a new instance. Migration is expensive: service is interrupted as AWS does not support live migration. Here we propose a proactive migration approach aiming at minimizing the service interruption impact during migration.

The migration overhead consists of two components: first, the time to start the new VM; second, the time to copy the

data and current application status from the old VM to the new VM. The average time to start a new VM is 90 seconds as reported in [22]. The time to copy data and current application varies across applications and can also be different for the same application at different time points. In Amazon EC2, data can be stored in Networking File System (NFS) so that both the old and the new instances have the same access to the data: this saves the time to copy data from old instance to new instance in the case of using VM local storage during migration. Regarding copying the application status, we use CRIU [23] to checkpoint the application status to NFS and then resume it in the new VM. The time to do it is a function of the data cached in memory and the transfer speed between memory and NFS. This process usually takes less than one minute [17]. Note that it is not possible to copy the application status in advance as the status is updated instantly. However, if we can predict the time when migration needs to be enabled, then we can start the new VM in advance to reduce migration duration. With the optimal `cpulimit` identified in Section IV-A, we do profiling experiments to measure the length of the credit depleting period under the optimal `cpulimit`. Then the time to start the new VM can be computed as $T_{new} = T_{CDP} - T_{launch}$, where T_{CDP} is the profiled duration of credit depleting period and T_{launch} is the time for starting a VM. Using proactive migration, the migration cost is reduced to (i) the time to checkpoint and resume the current application status; (ii) the time to start a new VM (the time to copy data across instances is already eliminated).

C. Multi-tier Cloud Services

Cloud services are often multi-tier applications and can benefit from the T2 burstiness offered by AWS. Our aim is to illustrate here that even for multi-tier applications, it is possible to use `cpulimit` and proactive migration to maximize user-perceived performance. We propose to use a central coordinating component, see Figure 4 that can identify the ideal `cpulimit` and schedule the migration of the different tiers. The central coordinating component collects the results from profiling experiments by adjusting the `cpulimit` at different tiers. Differently from the single-tier case, there can be multiple combinations of `cpulimit` at different tiers that can meet the performance requirement. To identify the optimal `cpulimit` combination from candidate combinations, we need to compare the overall migration overhead between different candidate combinations. Note that when one of the tiers is migrating, the entire service is interrupted, so the migration overhead is the accumulated migration duration across different tiers.

Assume there are M tiers and each tier has a migration frequency of F_m and migration duration of T_m, where $m = 1...M$. If there is no overlap between the migration of different tiers, the migration overhead is $\sum_{m=1}^{M} F_m * T_m$. However, if different tiers can migrate at the same time, then the migration overhead can be reduced. Here we define the tier with the smallest migration frequency as the main tier and schedule migration of other tiers when the main tier needs migration. We denote the migration frequency of the main tier as F_{max}. When all tiers migrate together, the migration duration is dominated by the tier with the longest migration duration T_{max}. To minimize the migration overhead, we select from the candidate combinations the one with $min(F_{max} * T_{max})$.

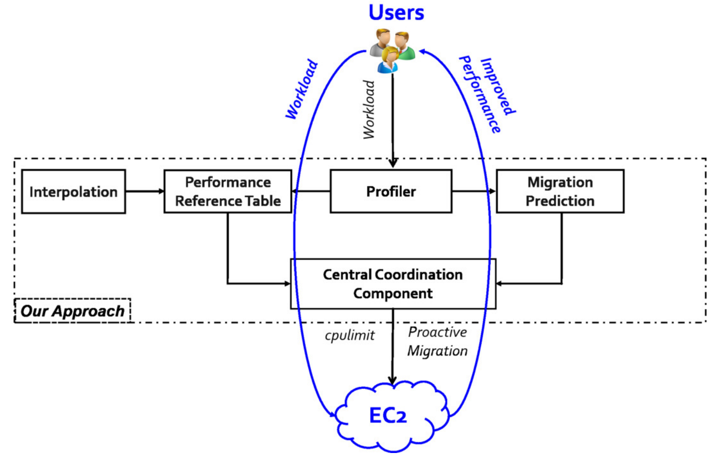


Fig. 4: Schematic view of the proposed approach.

V. Evaluation

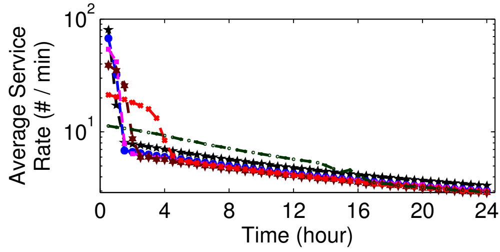
In this section, we evaluate our methodology proposed in Section IV. We first evaluate the effectiveness of using `cpulimit` to extend the credit depleting period. Then we evaluate our proactive migration approach. Finally, we compare our approach against an upgraded instance type.

A. Extend credit depleting period using `cpulimit`

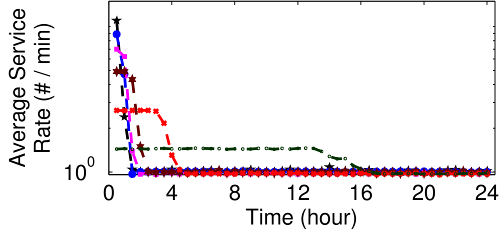
To verify the effectiveness of extending credit depleting period by using `cpulimit`, we consider both single-tier and multi-tier benchmarks. All experiments are run on Amazon T2 instances using the default Amazon Machine Image (AMI) with Ubuntu Server 12.04 LTS in the us-east-1a (Virginia) availability zone.

1) Single-Tier Cloud Service: We use three benchmarks: Hmmer, Avrora, and Sysbench hybrid, which are introduced in Section III. For each benchmark, we experiment with `cpulimit` from 10% to 80%, as well as the baseline case (i.e., no `cpulimit`). We report average service rate for a 24-hour period, see Figure 5. As expected, lower `cpulimit` level yields better extension in credit depletion period, e.g., for the cases with `cpulimit` levels equal to or smaller than 40%, the credit depletion period is significantly longer in all benchmarks. To take a closer look, we zoom-in the first 3-hour plot for `cpulimit` of 60% and 80% for Avrora, see Figure 5 (d). Compared with the baseline, we can see that even with only 60% and 80% `cpulimit`, the credit depletion period is still noticeable longer.

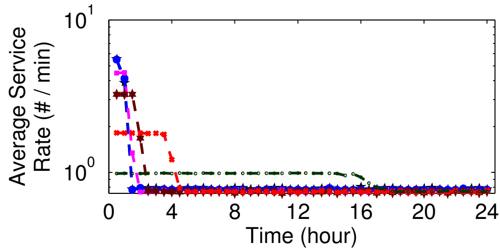
2) Multi-Tier Cloud Service: We use the multi-tier benchmark TPC-W [15] in our evaluation. TPC-W is implemented as a typical three-tier architecture: client, front-end web server (web), and back-end database server (db). Each tier is set up as one *t2.micro* instance. In our experiments, three types of TPC-W mixes are considered: browsing, ordering, and shopping. We first run the baseline experiments, i.e., without `cpulimit`. We observe that the front-end never depletes its CPU credit while



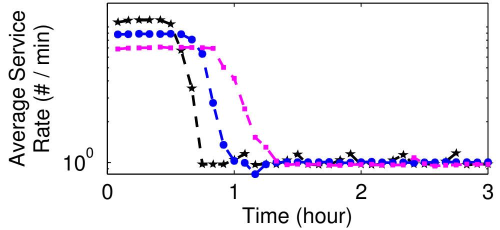
(a) *Hmmer*



(b) *Avrora*



(c) *Sysbench hybrid*



(d) *Avrora: zoom-in view for* `cpulimit` *of* 60% *and* 80%

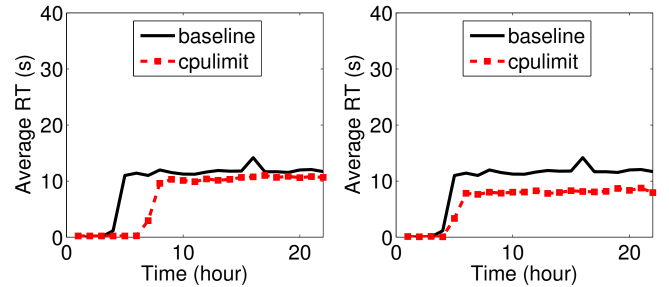
Baseline
CPU Limit 80%
CPU Limit 60%
CPU Limit 40%
CPU Limit 20%
CPU Limit 10%

Fig. 5: Average service rate of the *t2.micro* instance under different `cpulimits` for *Avrora*, *Hmmer*, and *Sysbench hybrid*.

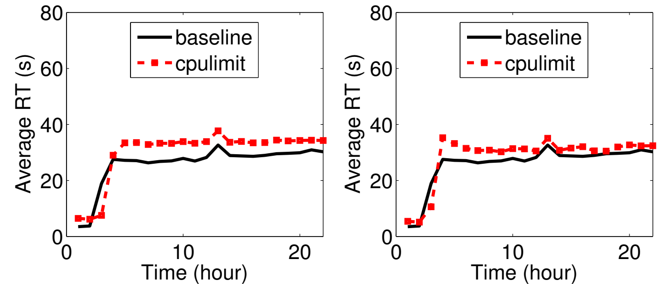
the back-end database server runs out of CPU credits very fast and becomes the bottle-neck throughout the rest of experiment. Therefore, no `cpulimit` is necessary for the front-end server. For the db tier, we apply `cpulimit` to extend the credit depleting period to improve its performance.

For multi-tier cloud service, we use the client response time (RT) as the performance measure of interest. We report the average RT per hour for browsing, shopping, and ordering, see Figures 6, 7, and 8 respectively. All figures present results for two load levels: 300 and 700 clients. In Figure 6, we observe that credit depleting period can be well extended under low request arrival load (e.g., 300 clients). A relatively low `cpulimit` (e.g., 30%) can get a significant extension of credit depleting period, therefore maintaining good performance for a longer period comparing to the baseline case. For higher arrival load (e.g., 700 clients), we can see that for both baseline and `cpulimit` experiments, the credit depleting periods are much shorter than for the low load case. To further illustrate the performance change, we also report the CDF of RT for 3-hour window using `cpulimit` of 70%. In the interest of space, we only show the first 3 windows and the rest windows are very similar to the third window, see Figure 9.

In general, if *aggressively* limiting the CPU usage (e.g., 30%), the credit depleting period is extended at the cost of less user performance improvement. If *conservatively* limiting the CPU usage (e.g., 70%), there is less improvement in extending the credit depleting period, but more benefits for user performance during this credit depleting period. Another interesting observation is that when `cpulimit` is applied, the response time can be even better than the baseline case. This is a direct outcome of the fact that `cpulimit` helps regulate flow across tiers. The reason is that flows of arrivals from the non-bottleneck tier to the bottleneck tier is regulated [24]. This effect persists in shopping and ordering mixes.



(a) *Client* = 300, `cpulimit` = 30%
(b) *Client* = 300, `cpulimit` = 70%

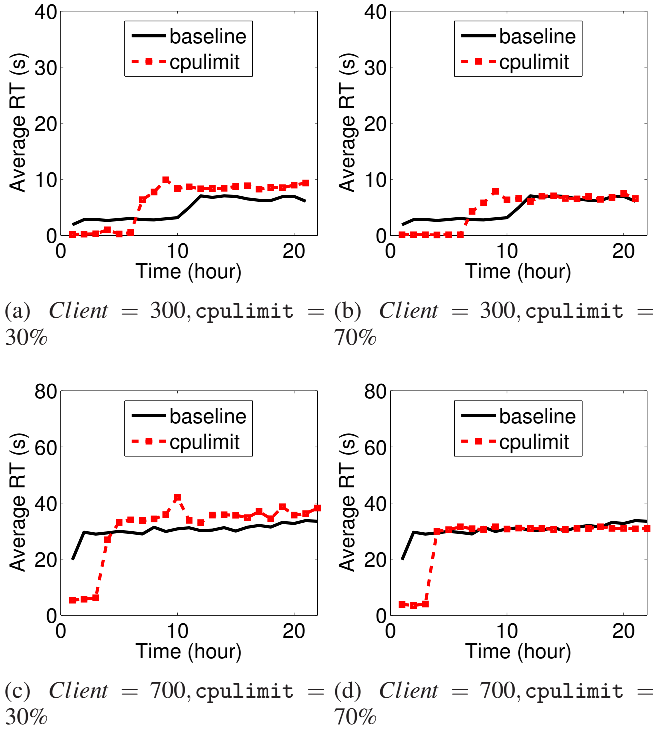


(c) *Client* = 700, `cpulimit` = 30%
(d) *Client* = 700, `cpulimit` = 70%

Fig. 6: Average response time (*t2.micro*) for *browsing*: different `cpulimit` with different arrival intensities.

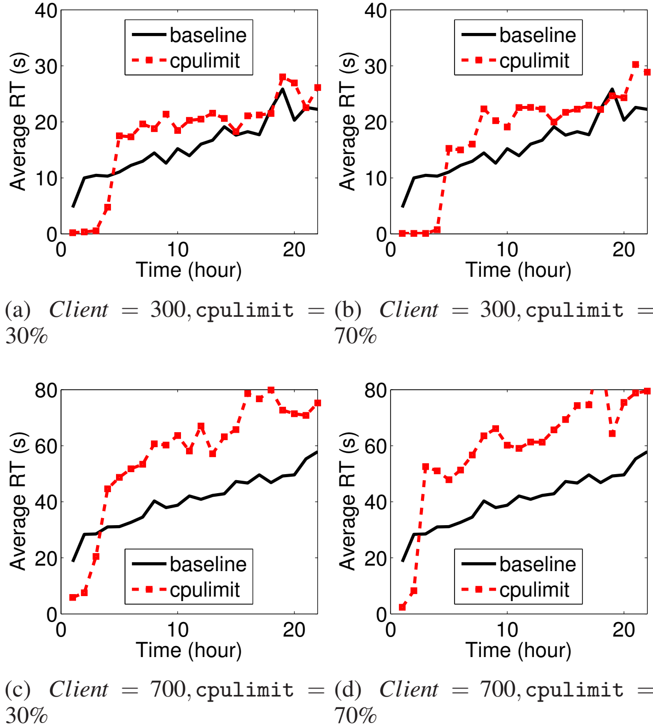
B. Minimize Migration Penalty

Our approach minimizes the migration penalty by reducing the migration frequency. To evaluate the ability to minimize migration penalty, we demonstrate credit depletion period length, migration frequency, reduced migration frequency compared to [17], as a function of `cpulimit`, see Figure 10. As shown



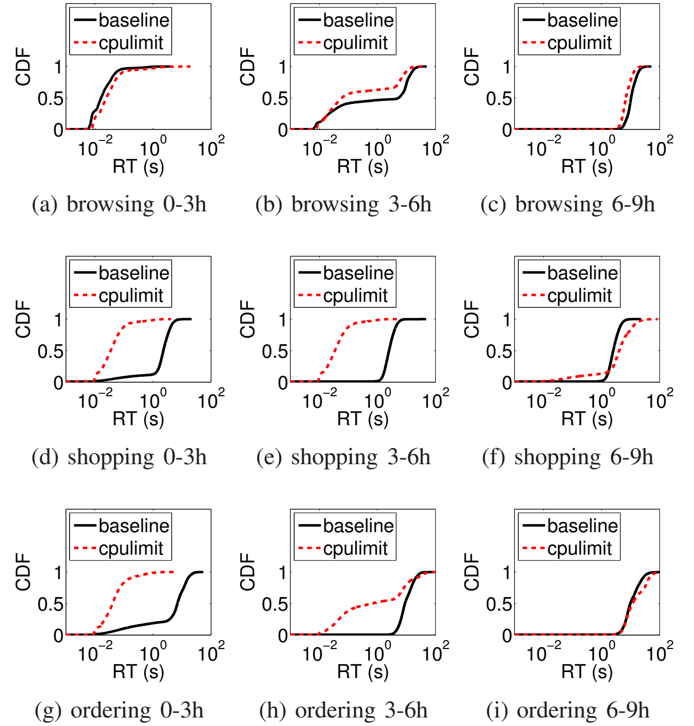
(a) $Client$ = 300, cpulimit = 30%
(b) $Client$ = 300, cpulimit = 70%
(c) $Client$ = 700, cpulimit = 30%
(d) $Client$ = 700, cpulimit = 70%

Fig. 7: Average response time (*t2.micro*) for *shopping*: different cpulimit with different arrival intensities.



(a) $Client$ = 300, cpulimit = 30%
(b) $Client$ = 300, cpulimit = 70%
(c) $Client$ = 700, cpulimit = 30%
(d) $Client$ = 700, cpulimit = 70%

Fig. 8: Average response time (*t2.micro*) for *ordering*: different cpulimits with different arrival intensities.



(a) browsing 0-3h
(b) browsing 3-6h
(c) browsing 6-9h
(d) shopping 0-3h
(e) shopping 3-6h
(f) shopping 6-9h
(g) ordering 0-3h
(h) ordering 3-6h
(i) ordering 6-9h

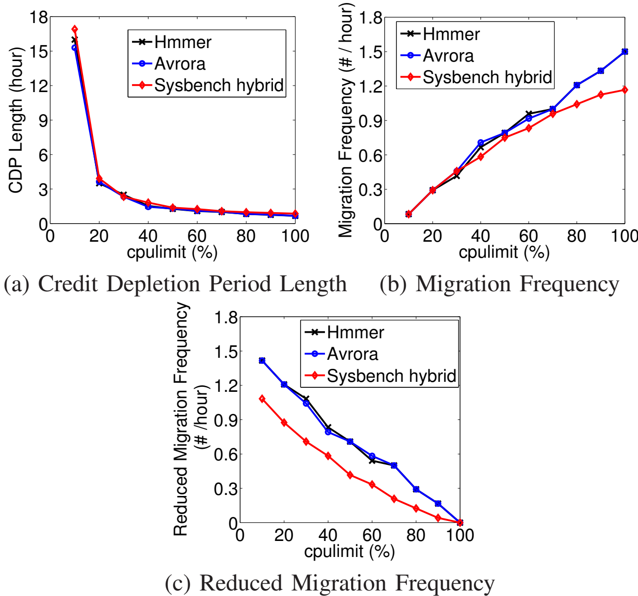
Fig. 9: Response time CDFs (*t2.micro*) for *browsing* (first row), *shopping* (second row), and *ordering* (third row) with cpulimit = 70%.

in Figure 10 (a), the credit depletion period length is approximately 0.67 hours for the baseline case. When cpulimit decreases to 10%, the credit depletion period length is gradually extended to 16.0 hours. A longer credit depletion period implies that less migration is needed. Figure 10 (b) shows the migration frequency as a function of cpulimit. Compared to the baseline (i.e., cpulimit = 100%), the frequency is greatly reduced as cpulimit reduces. For the baseline case, 36 new instances are launched during the 24-hour experiment. In the case of cpulimit = 80%, the necessary number of new instances decreases to 29; in the case of cpulimit = 10%, the number of new instance needed further decreases to 2. Figure 10 (c) shows how much migration frequency is reduced compared to the approach in [17]. As shown in the plot, the amount of reduced migration frequency increases as cpulimit becomes smaller and there are significant savings compared to [17].

In summary, the state-of-the-art approach [17] still suffers from a high migration cost due to the high migration frequency. By employing cpulimit and *proactive migration* together, our approach can effectively reduce migration frequency and minimize migration penalty, providing much better performance benefits.

C. Performance and Cost Compared to Upgrading the Instance Type

A direct way to improve performance is to use an upgraded instance type such as *t2.small*. The only drawback of such



(a) Credit Depletion Period Length (b) Migration Frequency

(c) Reduced Migration Frequency

Fig. 10: Credit Depletion Period length, Migration Frequency, and Reduced Migration Frequency as a function of `cpulimit` level, where Migration Frequency represents the number of instances launched per hour, and Reduced Migration Frequency represents the reduced migrations compared to the no-`cpulimit` case.

approach is the fact that *t2.small* instances cost twice more than micro ones. Here, we compare the performance and cost when using our approach in *t2.micro* instance compared to using the native mechanism of Amazon (i.e., no cpulimit nor migration) in *t2.small* instance. We also use the approach in [17] (i.e., equivalent to `cpulimit` of 100% case) as a reference. For easy comparison, we still use Hmmer, Avrora, and Sysbench hybrid. We first compare the performance using the CDF of service times, see Figure 11. It is clear that micro instance, using any `cpulimit` in our approach beats the Amazon default mechanism running on small in all the three cloud services.

Next we compare the cost between our approach and Amazon default mechanism, see Figure 12. It is clear that micro instances with `cpulimit` result in significant performance improvement over the micro baseline but even the small baseline case. Note that for different `cpulimit`, the cost is slightly different due to the different migration frequency during which there are two instances on at the same time, which results in double costs. Overall, our approach has obvious advantage in both performance and cost compared to upgrading the instance type.

VI. Related Work

Infrastructure-as-a-Service (IaaS) cloud instances have been extensively studied in the literature. Some works focus on performance exploration of single-tier cloud services. The work in [8] compares the performance (execution time of multiple types of workloads) of three popular Cloud IaaS providers: Amazon EC2, ElasticHosts, and BlueLock. The work in [9] characterizes EC2 T1 instances and propose an adaptive algorithm to diminish host-level throttling. The idea is managing CPU consumption through idleness injection decided based on the workload characteristics at runtime. In [10], the authors analyze the performance and scalability variations when a multi-tier application is migrated from a traditional datacenter environment to Amazon EC2. The authors in [25] experimentally compare performance and costs between Amazon EC2 and RDS through *mysqlslap* and TPC-W benchmarks, showing that EC2 is more suitable for simple workloads and low-cost scenarios

Amazon EC2 T2 instances are controlled by a credit-based bursting mechanism and there are few studies on this type of instance. The work in [26] is one of such studies, which explores performance and cost efficiency of T2 instances on single-tier cloud services. The authors also state that the instances controlled by credit-based bursting mechanisms (e.g., T2) are more suitable for CPU-bound applications with an average utilization of less than 40%. The work in [17] measures performance characteristics of EC2 *t2.micro* at different locations over time, and finally proposes a proactive migration approach to avoid CPU-throttling penalty for long-duration jobs. Although the migration approach enables applications to avoid experiencing performance penalty of throttling, it might introduce a large number of migrations during which the running benchmark is suspended, leading to large latency.

In [22], the authors study the startup time of cloud VMs across Amazon EC2, Windows Azure, and Rackspace: they analyze 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 acquired at the same time, thus providing the necessary bases for our proactive migration approach.

In this work, we focus on transparently extending the high performance period while minimizing the cost introduced in migration. In this way, users can get supercharged performance of T2 instances without paying the high premium of upgrading to a more performant instance.

VII. Conclusion

In this work, we propose a methodology for making better use of Amazon T2 instances. The proposed methodology extends the high performance periods offered by Amazon EC2 T2 and employs a proactive migration approach to reduce the cost in generating new high performance periods. The core of this methodology is judiciously selecting the `cpulimit` for transparent extension of high performance period and reducing the frequency of VM migrations. We evaluate our approach using both single-tier and multi-tier cloud services. The experimental results demonstrate the effectiveness and efficiency of our approach in improving the performance of cloud services (up to one order of magnitude) and reducing the VM migration cost (up to 80%).

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *FGCS*, vol. 25, no. 6, pp. 599–616, 2009.

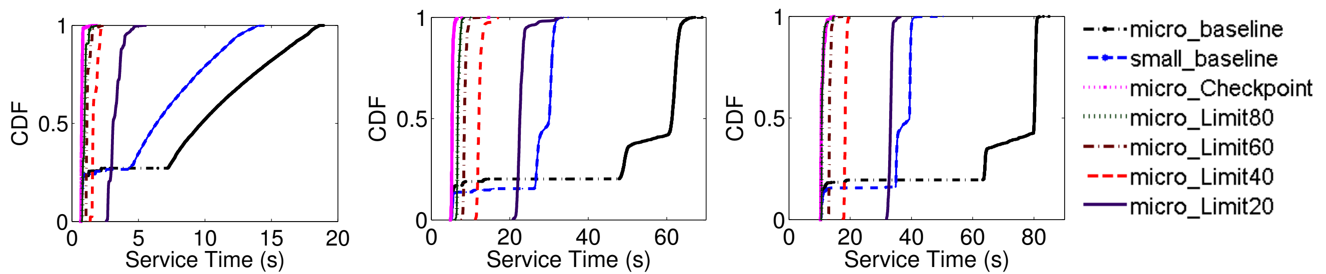


Fig. 11: Performance comparison between our approach running on *t2.micro* and baseline mechanism running on *t2.small*. We also show the Checkpoint based approach running on *t2.micro* as a reference.

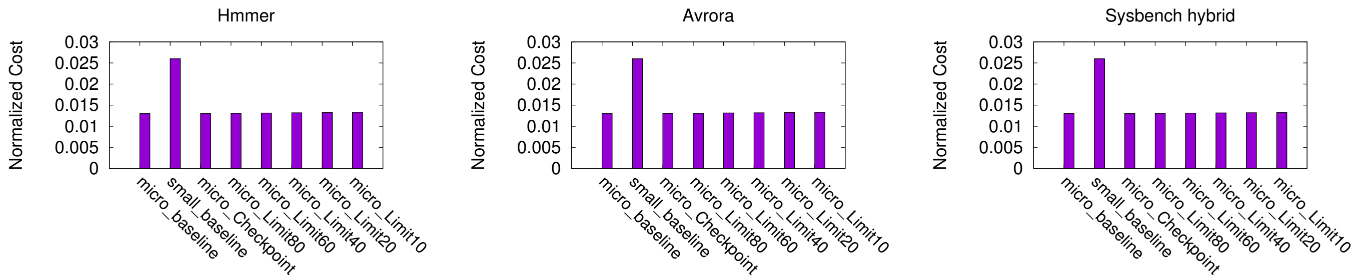


Fig. 12: Cost comparison between our approach running on *t2.micro* and baseline *t2.small*. We also show the checkpoint-based approach running on micro as a reference.

[2] B. P. Rimal, A. Jukan, D. Katsaros, and Y. Goeleven, "Architectural requirements for cloud computing systems: An enterprise cloud approach," *JGC*, vol. 9, no. 1, pp. 3–26, Mar. 2011.

[3] L. A. Barroso, "Warehouse-scale computing: Entering the teenage decade," *SIGARCH CAN*, vol. 39, no. 3, 2011.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[5] Amazon, "Amazon EC2."

[6] Google, "Google Compute Engine," https://cloud.google.com/.

[7] Microsoft, "Microsoft Azure," http://azure.microsoft.com.

[8] K. Salah, M. Al-Saba, M. Akhdhor, O. Shaaban, and M. I. Buhari, "Performance evaluation of popular cloud iaas providers," in *ICITST*, 2011, pp. 345–349.

[9] J. Wen, L. Lu, G. Casale, and E. Smirni, "Less can be more: Micro-managing vms in amazon EC2," in *IEEE CLOUD*, 2015, pp. 317–324.

[10] D. Jayasinghe, S. Malkowski, J. Li, Q. Wang, Z. Wang, and C. Pu, "Variations in performance and scalability: An experimental study in iaas clouds using multi-tier workloads," *IEEE TSC*, vol. 7, no. 2, pp. 293–306, 2014.

[11] B. Kaminski and P. Szufel, "On optimization of simulation execution on amazon EC2 spot market," *SMPT*, vol. 58, pp. 172–187, 2015.

[12] V. Persico, P. Marchetta, A. Botta, and A. Pescapè, "Measuring network throughput in the cloud: The case of amazon EC2," *CN*, vol. 93, pp. 408–422, 2015.

[13] P. C. Kokkinos, T. A. Varvarigou, A. Kretsis, P. Soumplis, and E. A. Varvarigos, "Sumo: Analysis and optimization of amazon EC2 instances," *JGC*, vol. 13, no. 2, pp. 255–274, 2015.

[14] M. Kiran, K. Maiyama, H. Mir, B. Mohammed, and A. Al-Ou'n, "Agent-based modelling as a service on amazon EC2: opportunities and challenges," in *UCC*, 2015, pp. 251–255.

[15] W. D. Smith, "TPC-W: Benchmarking an ecommerce solution," 2000.

[16] A. Marletta, "CPU Usage Limiter for Linux," http://cpulimit.sourceforge.net/.

[17] R. K. Mehta and J. Chandy, "Leveraging checkpoint/restore to optimize utilization of cloud compute resources," in *LCN Workshops*, 2015, pp. 714–721.

[18] "Amazon EC2 T2 instances."

[19] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH CAN*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

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

[21] A. Kopytov, "SysBench manual," 2014.

[22] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *IEEE CLOUD*, 2012, pp. 423–430.

[23] "CRIU."

[24] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Middleware*. Springer-Verlag New York, Inc., 2008, pp. 265–286.

[25] Y. Yamasaki and M. Aritsugi, "A case study of iaas and saas in a public cloud," in *IC2E*, 2015, pp. 434–439.

[26] P. Leitner and J. Scheuner, "Bursting with possibilities - an empirical study of credit-based bursting cloud instance types," in *UCC*, 2015, pp. 227–236.