Application-specific configuration selection in the cloud: impact of provider policy and potential of systematic testing

Mohammad Hajjat†, Ruiqi Liu⋆, Yiyang Chang†, T. S. Eugene Ng⋆, and Sanjay Rao†
†Purdue University, ⋆Rice University

***Abstract*—Provider policy (e.g., bandwidth rate limits, virtualization, CPU scheduling) can significantly impact application performance in cloud environments. This paper takes a first step towards understanding the impact of provider policy and tackling the complexity of selecting configurations that can best meet the cost and performance requirements of applications. We make three contributions. First, we conduct a measurement study spanning a 19 months period of a wide variety of applications on Amazon EC2 to understand issues involved in configuration selection. Our results show that provider policy can impact communication and computation performance in unpredictable ways. Moreover, seemingly sensible rules of thumb are inappropriate – e.g., VMs with latest hardware or larger VM sizes do not always provide the best performance. Second, we systematically characterize the overheads and resulting benefits of a range of testing strategies for configuration selection. A key focus of our characterization is understanding the overheads of a testing approach in the face of variability in performance across deployments and measurements. Finally, we present configuration pruning and short-listing techniques for minimizing testing overheads. Evaluations on a variety of compute, bandwidth and data intensive applications validate the effectiveness of these techniques in selecting good configurations with low overheads.**

I. Introduction

The performance of applications in public cloud environments is influenced by *policy* decisions made by cloud providers such as (i) *bandwidth rate-limits,* imposed on virtual machines (VMs) to control network utilization; (ii) *VM packing decisions,* which determine how many VMs are packed into a given physical machine; and (iii) *CPU scheduling policies,* which determine when and how often a VM must run, and whether all virtual CPUs (vCPUs) of a VM with multiple vCPUs should run concurrently.

Cloud providers typically support multiple configuration options (e.g., VM sizes). Further, modern cloud datacenters (DCs) exhibit significant hardware heterogeneity in terms of multiple models of CPU, disk, network interface, and memory [12], [19]. Policy decisions may be made at the granularity of individual configurations (VM size and hardware). For instance, rate limit and CPU scheduling policies are often different across VM sizes.

While cloud provider policy is likely to significantly impact application performance for a given configuration, there is little public documentation about policies. The choice of configuration (both VM size and hardware) for an application affects not only its performance, but also its cost – ideally, developers would like configurations that meet their performance requirements at acceptable costs.

This paper seeks to shed light on the interplay among provider policy, configuration choice, and application performance. To this end, we perform an extensive measurement study on Amazon EC2 over multiple applications. Our measurements span a 19 months period from 2012 to 2014, and include bandwidth-intensive applications in inter-DC and intra-DC settings, a variety of compute- intensive applications, Cassandra [8] (a popular key-value store) and MPI-Fast Fourier Transform (a network and CPU intensive application).

Our measurement approach has been guided by pragmatic constraints. First, academic budget constraints have led us to be selective in terms of the configuration space to test, rather than exploring all provider offerings. Our measurement study has already cost several thousand dollars. Second, we are required to adopt a *black-box* approach in interpreting our findings given limited support from the cloud provider. We have attempted to circumvent this to the extent possible through information publicly available on provider forums (e.g., [2], [3]), and by conducting numerous auxiliary measurements that allow us to better explain our findings.

Findings: Our measurement study reveals that cloud provider policy indeed impacts the relative performance of different configurations in surprising ways. Specifically:

• Different rate limiting policies are employed across hardware generations even within VMs of the same size, potentially because of the complexity of updating policy for older hardware. Larger VM sizes may not necessarily see higher rate limits since they may be provisioned with more conservative statistical multiplexing assumptions.

• VM packing policies, and their interaction with hardware features (e.g., hyper-threading) can lead to significant diversity in inter-DC TCP throughput across different receiver CPU types of the same VM size.

• The relative performance of configurations for CPU intensive apps is only partially explained by compute resources (CPU model, RAM). In some cases, larger VM size performs consistently worse than smaller VM size, potentially due to the interaction between CPU type and CPU scheduling policy.

Systematic configuration selection: Motivated by these find-

	Virtual CPU	RAM	Net perf	Hourly cost
Small	1 vCPU / 1 ECU	1.7 GB	Low	$0.044
Medium	1 vCPUs / 2 ECUs	3.75 GB	Low	$0.087
Large	2 vCPUs / 4 ECUs	7.5 GB	Moderate	$0.175
Extra large	4 vCPUs / 8 ECUs	15 GB	High	$0.350

TABLE I
GENERAL PURPOSE VMS (M1) IN EC2.

	CPU	Speed (GHz)	Cache (MB)	Release	Cores	Hyper-threading
A	E5430	2.66	12	Q4 2007	4	No
B	E5645	2.40	12	Q1 2010	6	Yes
C	E5507	2.26	4	Q1 2010	4	No
D	E5-2650	2.00	20	Q1 2012	8	Yes

TABLE II
CPU TYPES (ALL INTEL(R) XEON) FOUND IN AMAZON AWS DCS (NORTHERN VIRGINIA AND CALIFORNIA).

ings, and using data that we collected, we systematically evaluate strategies for configuration selection across a range of applications. Our evaluations seek to characterize (i) the effectiveness of the techniques in choosing configurations that could best meet the performance and cost requirements of an application; and (ii) the costs that the techniques incur.

The strategies we evaluate include (i) PerConfig, a technique that inspects all configurations; (ii) iPrune, a technique which *conservatively* prunes options for each configuration dimension (e.g., VM Size, CPU type) in an iterative fashion based on aggregate performance of all configurations which involve that option; and (iii) Nearest neighbor (NN), a technique which short-lists potentially better performing configurations for an application based on the performance testing results of other applications that fall within the same general class. NN could then be combined with PerConfig or iPrune to test the short-listed configurations and pick the best among them.

Evaluations confirm the effectiveness of our techniques while keeping overheads acceptable. iPrune reduces the number of tests by $40\% - 70\%$ across a wide range of applications while picking configurations that perform within 5% of the best. NN when tested on a mix of CPU-intensive applications as well as a distributed CPU and communication-intensive application, selects a configuration that performs within 20% of the best with *no testing* on the target application; accuracies improve to within 6% of the best when the technique short-lists 4 configurations and is combined with PerConfig to pick the final configuration.

Our contributions in this paper include (i) a deeper understanding of how cloud provider policy impacts performance of cloud configurations; (ii) our extensive measurements of a real-cloud deployment, and the insights gleaned - we intend to release our data under an open source license to the wider community; and (iii) our techniques, and a systematic characterization of their overheads and accuracies, which represent an important first step towards application-specific configuration selection in cloud environments.

II. Measurement methodology

Our study is conducted in the context of Amazon EC2 public cloud. We mainly use US-East (N. Virginia) and US-West (N. California) regions for our measurements. We use the general purpose M1 VMs in our experiments (summarized in Table I) since they were widely used as the standard instances throughout our study. While Amazon started offering M3 instances as the standard recently (late January 2014), M1 instances continue to be supported and used widely, and our methodology is generic and can be extended to other instance types as well. There are four CPU types that we could find in both regions for all VM sizes, summarized in Table II. We consistently refer to the CPU types by the abbreviations A, B, C, and D throughout the paper. When convenient, we abbreviate VM sizes as S, M, L and X denoting small, medium, large, and extra large, respectively.

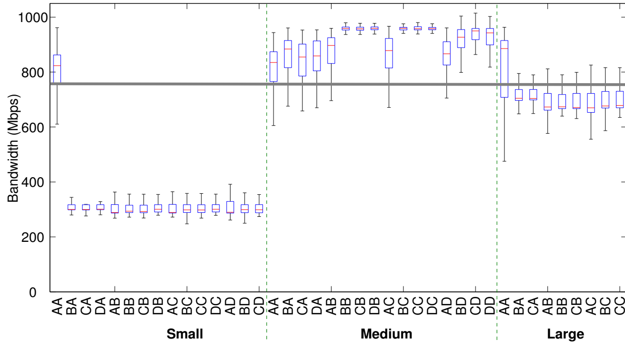
Amazon uses the notion of EC2 compute units (ECU) as an attempt to provide each VM a consistent and predictable amount of CPU capacity [4] regardless of which CPU it runs on. One ECU provides the equivalent CPU capacity of a 1-1.2 GHz 2007 Opteron or 2007 Xeon processor. There is no public documentation available on what an ECU translates to on different CPU types and how Amazon allocates VMs to physical processors. However, our experiments with dedicated instances in §III-A indicate that the number of large VMs that may be assigned to B, C and D processors is equal to the number of cores on these processors.

It is relatively straight-forward to obtain the CPU information of an instantiated VM using the Linux command `/proc/cpuinfo`. However, we found information about many other hardware features, such as memory, disk and network interface is not exposed in virtualized EC2 environments, and a variety of commands that we tried such as `lshw` and `dmesg` commands did not give us sufficient information to infer more details about other hardware information. Hence, in this paper, we focus on CPU information.

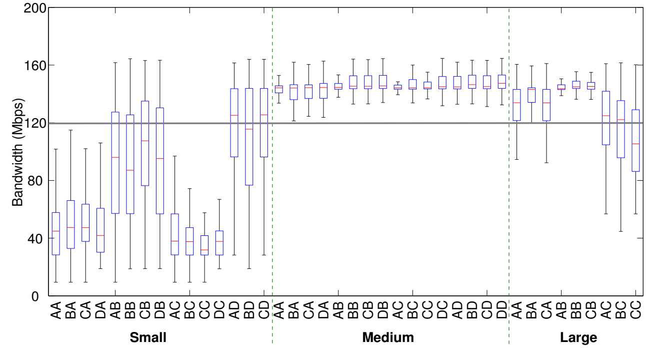
We use *iperf* to measure TCP throughput between VMs inside a single DC, and across DCs in two different geographic regions. The window size in *iperf* is set to 16 MB (recommended in [17], and which we experimentally confirmed to result in the best throughput), and we sufficiently increase the OS send and receive buffer sizes. We study CPU intensive applications using the SPEC CPU2006 benchmark suite which consists of 29 widely used applications (12 integer intensive and 17 floating point intensive). These applications include gcc, video compression, Go game player and chess player, path-finding in 2D maps, protein sequence analysis, etc. We measure the time for single-threaded executions per core.

We refer to a *configuration* as a unique combination of choices that could be made by a user. For CPU intensive apps, we consider every $<$Size, CPU$>$ combination, where Size can be either S, M, L or X, and CPU can be A, B, C or D. For *iperf*, we consider every $<$Size, SrcCPU, DstCPU$>$ combination. E.g., SAD denotes a configuration of small VM sizes where the source and destination CPUs are of type A and D, respectively. For each configuration, we test multiple *deployments* to get a statistically sound measure of performance with that configuration. Further, we collect multiple *measurements* from each deployment since the performance of each deployment can vary depending on various factors.

For CPU intensive apps, we obtain 10 deployments for each configuration and conduct 3 complete runs each. Small



(a) Intra-DC measurements



(b) Inter-DC measurements

Fig. 1. *iperf*'s measured TCP throughput across different configurations. The bottom and top of each boxplot represent the 25th and 75th percentiles, and the line in the middle represents the median. The vertical line (whiskers) extends to the highest datum within 3*IQR of the upper quartile (covers 99.3% of the data if normally distributed), where IQR is the inter-quartile range. CPU type D was not available in one of the DCs for small VMs, and in both DCs for large VMs.

VMs need 24 to 29 hours to finish a single run of all CPU intensive apps; M, L and X VMs need 8 to 10 hours. Our *iperf* experiments are more expensive since they involve a much larger set of configurations, and inter-DC bandwidth must also be paid for. Hence, we focus our explorations on S, M and L VMs, and restrict ourselves to combinations where the source and destination VM sizes are the same (i.e., a total of 48 possible configurations). We run *iperf* in both intra-DC and inter-DC settings. In each setting, we use 32 deployments for each configuration (i.e., 48×32 deployments in total). We obtained 6 *iperf* measurements every hour for each deployment, each measurement lasting sufficiently long to obtain steady state TCP throughput. Finally, with both CPU intensive apps and *iperf*, we were unable to collect data for a small number of configurations due to the difficulty of obtaining a CPU of desired type at the time of the experiment and do not include them in our analysis (e.g., XA and XC with CPU intensive apps experiments).

Our measurements started in late 2012. The insights gleaned from initial results guided our large-scale systematic data collection in mid 2013, which provides the primary data used in most of our analysis. Since then, we have been collecting data at a smaller scale on an ongoing basis (most recently in mid 2014), which have helped reconfirm our observations. We have also collected substantial auxiliary data – e.g., UDP loss rate, traceroute data, and measurements using tools like *lmbench* [18] – to shed deeper light on our findings.

III. Measurement Results

A. Impact of policy on intra-DC throughput

Figure 1(a) shows the variability in TCP throughput with *iperf* across different configurations for intra-DC traffic. Each boxplot includes all measurements of all deployments of a given configuration, whose name is annotated on the X-axis.

Figure 1(a) shows that for almost all S configurations, the TCP throughput is limited to under 300 Mbps. While Amazon no longer provides official documentation for the rate limits employed, our measurements are consistent with anecdotal information reported on forums [1], [7]. Interestingly, there does

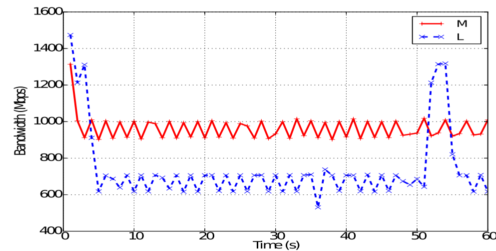


Fig. 2. Rate limit behavior for a typical dedicated M and L VM pair. VMs in each pair are not co-resident on the same physical host.

exist one S configuration (SAA) which performs extremely well, and quite comparably to M and L configurations. Further, we find the corresponding L configuration (LAA) also has a higher throughput than the other L configurations, which are limited to about 650 Mbps. We note that A processors are the oldest generation hardware dating back to 2007. We believe that Amazon's rate-limiting policies have evolved since the initial offerings – it is possible that the new policies were only employed with later generation hardware, and the overheads of updating rate-limiting policies on older hardware were not deemed worthwhile given the management complexity and overheads associated with policy reconfiguration.

Another surprising observation from Figure 1(a) is that M VMs achieve higher TCP throughputs than L VMs, with the median being 35% higher – for all M configurations, more than 75% of the samples exceed 750 Mbps, while for all L configurations (except LAA) less than 25% of the samples exceed 750 Mbps. We repeated the measurements with multiple parallel TCP connections (2 to 8). The result showed that for both M and L, the sum of TCP throughputs across all connections was nearly unchanged when the number of parallel connections varied between 1 and 8, and M still performed better than L regardless of the number of parallel TCP connections. We also repeated our experiments with L VMs by pinning the *iperf* process to a single vCPU, to observe whether L configurations see lower bandwidth owing to context switch between different vCPUs. We found that doing so did not change the TCP throughput achieved by L.

Rate limiting under controlled multi-tenancy with dedi-

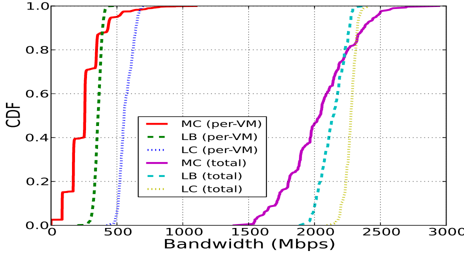


Fig. 3. Per-VM and total throughput of all VMs when all dedicated VMs packed on same physical CPU are concurrently active as sources, for different combinations of VM size and CPU type.

cated VMs: To further explore distinctions in rate-limiting policies used for M and L VMs, we conducted experiments using dedicated VMs, which are physically isolated at the host level from VMs that belong to other tenants. Dedicated VMs however can be aggregated onto the same host if they belong to the same tenant and the same VM size [2]. Experiments on dedicated VMs enable us to study rate-limiting policies without interference from traffic generated by other tenants.

Figure 2 shows typical TCP throughput behaviors of a pair of M and a pair of L VMs. Each pair consists of two dedicated VMs that are not co-resident. We adapted a technique [6] utilizing XenStore [10] to identify if two VMs are co-resident. The figure shows (i) even without multi-tenancy, M VMs achieve higher rates than L VMs, and (ii) L VMs can tolerate higher bursts than M VMs. We repeated the experiment for several pairs of M and L VMs and different CPU combinations, and found the trend holds.

Per-host per-tenant rate-limiting policies: We next explore rate-limiting policies applied to multiple VMs on a single physical host. To study this, we invoked many M (L) dedicated VMs back-to-back, leveraging our empirical observation that Amazon tends to pack a tenant's dedicated VMs with the same size onto the same physical host as far as possible. We found that with different underlying CPU types and VM sizes, the largest number of VMs packed on a certain machine is different – e.g., we found up to 6 L VMs on a B CPU, 8 L VMs on a D CPU, and 4 L or 8 M VMs on a C CPU.

We conducted experiments to measure bandwidths of concurrent *iperf* TCP connections, whose sources are VMs resident on the same host and whose sinks are located on different hosts. We separate the sinks to avoid them being bottlenecks. Figure 3 shows a CDF of TCP throughputs achieved by one source VM, as well as the total throughput aggregated across all source VMs, in terms of different combinations of VM size and CPU type. Interestingly, the total TCP throughput is around 2 Gbps for all combinations. Given typical Ethernet NIC bandwidth is 1 Gbps or 10 Gbps, this points to the use of rate-limiting policies over the aggregate traffic sourced by co-resident VMs. Our hypothesis is that rate-limiting policies are being used over all VMs belonging to the same tenant on a given physical host, though it is also possible the policy is being applied on a per-host level regardless of tenant. The throughput per VM simply depends on the number of VMs packed on each host, which in turn depends on VM size, and the number of cores in the sender CPU – thus, the per-VM throughput for MC is lower than LB, which in turn is lower than LC. Further, both total throughput and per-VM throughput are less variable for L compared to M VMs.

Overall, our results suggest that rate-limiting policies are employed both at per-VM level and at per-host per-tenant level. The per-VM limit depends on both VM size and CPU type. Surprisingly, L VMs are limited to lower average throughputs than M VMs – however, L VMs are afforded higher burst sizes, and achieve higher rates and less variable performance with multiple simultaneously active tenants. We hypothesize more conservative rate-limiting policies are used with L VMs to ensure more predictable performance under multi-tenancy, or potentially to ensure more reserved capacity for handling higher priority or provisioned I/O.

B. Impact of policy on inter-DC throughput

Figure 1(b) shows the variability in TCP throughput with *iperf* for inter-DC traffic. Inter-DC throughput is less than that in intra-DC settings across all configurations, and does not hit the rate-limiting level, indicating it is impacted by other factors. Further, based on the observed throughput, configurations can be roughly separated into three groups. While S configurations with A/C receiver types are well below the 120 Mbps threshold, all M and most L configurations see over 75% of their samples above 120 Mbps. The intermediate group includes S configurations with B/D receiver types (if normalized for costs, these S configurations may be considered better than M and L configurations), and more surprisingly L configurations with C receiver type. We next seek to understand the potential causes behind our observations.

Is the network responsible for performance divergence? To understand this, we collected UDP loss rate and traceroute data. Our UDP measurements were conducted for a sender transmitting at 150 Mbps using small VM pairs to multiple receivers, for both inter- and intra-DC settings. Our measurements show that UDP loss rates with receivers of CPU types A/C are higher (median loss rate $> 2.5\%$) than loss rates with receivers of CPU types B/D in both settings (median loss rate about 0.1%). Surprisingly, the magnitudes of the loss rates for the same receiver type are similar for both inter- and intra-DC settings, indicating that the losses do not occur over the WAN. We note that while UDP loss rates in inter- and intra-DC settings are similar, TCP throughput is vastly different because: (i) TCP throughput is inversely proportional to round trip times (RTT) [13]; and (ii) bandwidth is rate limited to about 300 Mbps for small VMs.

We have also analyzed traceroute data collected from multiple sources in the US-West to multiple receivers of different CPU types in the US-East DC. Since EC2 allows for multi-path routing, we conducted over 150 traceroute measurements for each source and receiver pair. For a particular source VM to destination CPU types of B and C, our results show that all IPs in the first 8 hops (which included the WAN hops to US-East) overlapped. Similar results were observed with other sources. This suggests the divergence in performance is not related to

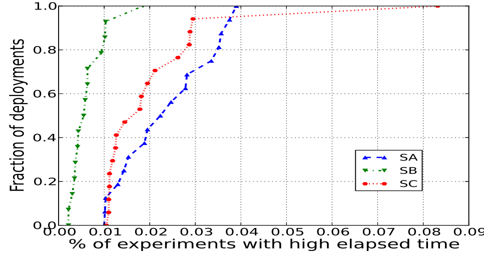


Fig. 4. Prevalence of high elapsed time for small VMs. Each curve corresponds to a different CPU type. Each point shows the fraction of deployments for which less than a certain % of experiments see elapsed time > 10ms.

the WAN. Further, our traceroute analysis from sources inside the US-East DC to multiple receivers of different CPU types in the same DC indicate that the paths from a common source to the different receivers largely overlap. This further indicates that any variation in throughput should relate to issues very close to the receivers.

Does performance divergence occur due to end hosts? We analyzed UDP losses on receiver nodes and noticed that packets are typically dropped in large bursts, with median losses for A/C receivers greater than 300 packets. Recent work [24] has shown that mixing delay-sensitive jobs on the same physical node with several CPU-intensive jobs can lead to longer than expected scheduling delays, owing to the scheduling algorithm used in the Xen hypervisor being unfair to latency-sensitive applications [5], [9]. To evaluate if this is a factor, we ran experiments on different VM instances which involved having a process sleep for a short period (1 ms), and observe the actual elapsed sleep time. We repeated this experiment systematically over tens of instances for every VM size and CPU type combination, conducting 600,000 sleep experiments on each instance. For all instances, the elapsed time is typically 1 ms, however occasionally much higher (often multiples of 30 ms, which corresponds to the CPU allocation time slice in Xen).

Figure 4 shows that for small VMs, A/C CPU types experience more episodes of high elapsed times than CPU type B. While 90% of B deployments see high elapsed time in less than 0.01% of the runs, the corresponding value for C is 0.03% and is even higher for A. While not shown, medium and large VMs have fewer episodes of high elapsed time than small ones across all CPU types (with B/D CPU types continuing to see fewer episodes than A/C CPU types), which explains their generally better performance. We believe B/D CPU types are less affected because both these CPUs support hyper-threading, which allows two threads to be simultaneously scheduled on a single core, thus minimizing scheduling delays for latency-sensitive applications. Further, the number of VMs per core for B, C and D CPU types is identical (§III-A). These results indicate that the VM allocation is not customized to CPU characteristics like hyper-threading, leading to poorer performance for processors that are not hyper-threaded. Finally, we believe configurations with LC as receivers perform worse than those with MC as receivers owing to scheduling policies that impact CPU type C differently from other CPU types – we explore this further in §III-C.

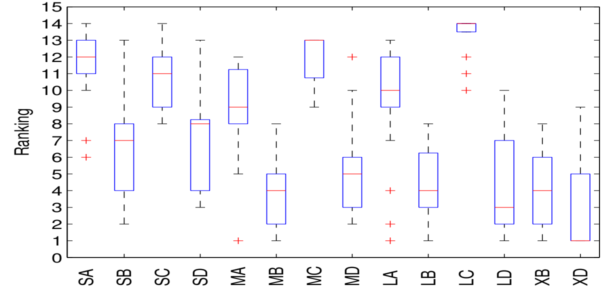


Fig. 5. Distribution of performance ranking of each configuration across all CPU intensive apps. For each application, the best configuration (with lowest execution time) gets a ranking of 1. XA and XC were not available during the time of the experiment.

C. Resource and policy's impact on CPU intensive apps

We next shift to the CPU intensive apps described in §II. Figure 5 shows a distribution of performance ranks for each configuration across all 29 CPU intensive apps (lower rank is better). Configurations are ranked based on the 90th percentile execution time of each application among 30 runs (§II). Though Amazon uses performance benchmarking to normalize performance of VMs of the same size on different CPU types [4], the figure shows that performance differences across different CPU types are strong in reality.

Overall, CPU types B and D are relatively better than A and C, because B and D have newer micro-architectures and larger cache sizes. Larger VMs perform better because of their advantage in memory capacity. However, much variability exists in the performance of each application across different configurations that is not captured by the general trends. For example, the highly resourceful configuration XD is not ranked in the first 4 places for one fourth of the applications. As another example, the oldest CPU type A achieves good performance for the molecular dynamics simulator – MA and LA are the top 2 configurations. These exception cases suggest that generic rules of thumb that prefer larger VMs and newer CPUs are inadequate for configuration selection.

One surprising finding is that MC outperforms LC for all single threaded CPU intensive integer apps, but other M VMs do not perform better than their corresponding L VMs. We initially hypothesized that MC VMs were located on physical processors with lower memory/cache contention as compared to LC, but found two pieces of evidence to rule this out. First, we found LC VMs take 10% more time than MC (measured over many runs) when doing single threaded multiplication of two integer constants, an operation that essentially has no memory access. Second, *lmbench* results showed that MC VMs have lower read latency and less variation than LC counterparts for all levels of the memory hierarchy. If memory/cache contention were the main explanatory factor, only latencies of memory and L3 cache (shared across cores) would be lower and not latencies of L1 cache (private to each core). In contrast, read latencies are similar for M and L VMs

of CPU types A, B and D for all memory hierarchy levels. The better performance of MC VMs in the experiments above further suggests that for C CPU type, each vCPU of L VMs is scheduled slightly less frequently than M VMs.

We also conducted experiments where we launched threads on two different vCPUs of L VMs and measured the delay between the time the two threads started running over many iterations. LC VMs incur higher delay than L VMs of other types – for LA, LB and LD, $> 99.8\%$ iterations have a delay of less than $0.1ms$, while for LC, 3.2% iterations have a delay of more than $1ms$. These results further indicate scheduling policy discrepancy between LC VMs and other L VMs.

IV. Systematically choosing configurations

We now investigate techniques for systematically choosing configurations. We formulate a general configuration selection problem in which the cloud provider allows customers to choose from arbitrary combinations of CPU types and VM sizes to best match application performance requirements. This formulation makes sense from the provider's perspective because the provider could monetize its resources more effectively through differentiated pricing of configurations. The formulation also matches the recent trend in EC2 of creating explicit instance families for different processor types, essentially allowing explicit CPU and VM size selection. We propose and evaluate two testing techniques that tackle this general configuration selection problem.

A. Metrics for comparing schemes

Application developers may use a variety of metrics when comparing configurations (e.g., raw performance, dollar cost, etc.) Our framework is generic and can use any metric; however, for concreteness, our evaluations use a metric which combines both performance and cost, reflecting common considerations of cloud customers. We *score* each configuration according to application's performance and cost (lower scores are more favorable).

Bandwidth-intensive applications are scored based on the total dollar cost associated with each unit byte of data transfer. We consider the cost of VMs needed for testing, in addition to the cost of bandwidth transfer for inter-DC settings (intra-DC communication is free). In CPU intensive apps, we use the total cost (product of the cost per unit time and the total execution time) of running an application with a given configuration. Since the score of any given configuration is itself variable (across deployments and measurements), we compare configurations based on a desired percentile score: all our evaluations consider either the median or 90th percentile.

We evaluate the **relative error** in score achieved by the configuration recommended by a scheme compared to the score achieved by the best configuration. We use measurements collected across multiple deployments of all configurations (§II) as ground truth to determine the best configuration. We capture testing costs using (i) the *number of tests,* i.e., the total number of measurements conducted across all deployments of all configurations; and (ii) *total dollar costs* involved across all

```
Procedure IterativePruning( )
Let α: probability threshold to determine if samples in one distribution are
greater than another
Let m_{x,i}: measurements from all deployments that have choice c_x for
dimension d_i
While more testing is required
  GetKDeploymentsPerDimension(K)
  // Pruning poor choices within each dimension
  Foreach dimension d_i, and choices c_x and c_y in d_i, c_x ≠ c_y
      p ← P(m_{x,i} > m_{y,i})          // compare the two choices c_x vs. c_y
      If (p > α)        // decide if c_x need to be pruned from the dimension
        Mark c_x in d_i for pruning
      End If
    End Foreach
  Remove deployments associated with choices marked for pruning
End While
```

Fig. 6. Iterative pruning algorithm.

measurements, computed by considering VM and bandwidth costs as described above.

B. Schemes for testing configurations

• **Per-Configuration Testing (PerConfig).** PerConfig is a strawman approach that exhaustively explores all configurations. For each configuration, PerConfig randomly samples D deployments from the ground-truth data, and randomly picks M measurements per deployment. The best performing configuration is selected from the sampled data.

• **Iterative Pruning (iPrune).** To reduce testing overheads, we propose iPrune, which iteratively prunes poor choices in each dimension and performs more testing with better performing choices – e.g., the VM size, src-CPU, and dest-CPU are three dimensions of *iperf* data, and the choices for the VM size dimension are S, M or L.

Figure 6 illustrates iPrune. Rather than testing all configurations, iPrune simply ensures there is measurement data for at least K deployments for each choice along every dimension d, with M measurements for each deployment. Then a total of $K \times \max\{|d_1|, |d_2|, \ldots, |d_n|\}$ deployments need to be tested in the first round, where $|d_i|$ is the total number of choices along dimension d_i. Measurements from all deployments that have choice c_x for dimension d_i are grouped together. For each dimension, a choice that is clearly worse than another is dropped. Poor performing choices are eliminated and the process is repeated with the remaining choices. When no further pruning is possible, the best configuration is found by picking the best choice along each dimension.

Figure 7(a) illustrates how iPrune picks the best median score with *iperf*. After the first round, L VM sizes are eliminated (higher costs and poorer performance than M), but S and M VM sizes remain since it is hard to say one choice is better than another. Likewise, at the end of the first round, options A and C are eliminated on the Dst-CPU dimension since they are clearly inferior, but no options are eliminated on the Src-CPU dimension. No further pruning is possible after the second round. The remaining data correspond to configurations $<$ S or M, *, B or D$>$. Choosing the best option for each dimension leads to selecting SAD (best median score).

A key aspect of iPrune is that it prunes choices within a dimension conservatively (Figure 6). It first computes the

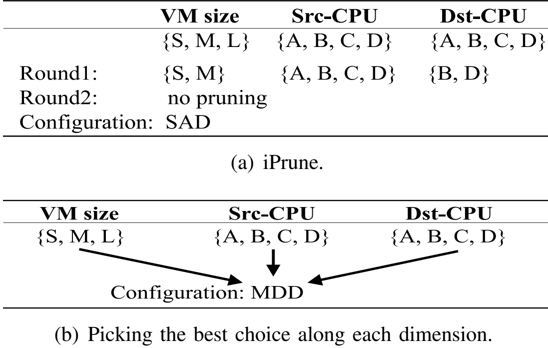


Fig. 7. Contrasting iPrune with an alternative approach for *iperf* (inter-DC settings), using ground truth data shown in Fig. 1(b). Each subfigure shows the final selected configuration, based on median score. The relative error is 0% in (a) and 70% in (b).

probability that choice $C1$ performs better than choice $C2$. If there are m and n samples for the two choices with performance samples $X_1 \dots X_m$, and $Y_1 \dots Y_n$ respectively, the probability is computed as $\frac{U}{mn}$, where U is the number of times X precedes Y in the combined order arrangement of the two samples X and Y. We note that the U statistic is closely related to the *Mann Whitney test* [14], which is a non-parametric test of the null hypothesis that two populations are the same against an alternative hypothesis that a particular population tends to have larger values than the other. Along each dimension, we prune those choices that have a high probability (0.9) of being out-performed by another choice.

A more aggressive approach that merely picks the best choice from each dimension after the first round would **not** work as well as illustrated in Figure 7(b), even with the ground-truth data. This method picks configuration choice MDD, which has a median score 70% higher than optimal. VM size M is selected since the median score of all M configurations in aggregate is better than the median score of S configurations. In contrast, iPrune is able to pick S since it prunes Dst-CPU choices A and C in the first round.

C. Evaluation of schemes

Our evaluations consider various applications:

• **Iperf:** Figure 8 shows the trade-off between testing overhead and accuracy in selecting configurations with *iperf*. For both schemes, the accuracy improves as the overall number of measurements increases. Further, iPrune (cluster of lines at bottom) performs much better than PerConfig. For a target relative error of 0.05, iPrune lowers the number of tests from around 3000 to 700 (> 70% reduction), and reduces dollar costs from \$60 to \$10 (> 80% reduction). The cost saving is especially significant for jobs of moderate length, or for larger-scale applications with high absolute testing costs.

• ***Cassandra***: This is a network-centric, widely used distributed key-value data store [8]. We run our experiments using the well-known *Yahoo! Cloud Serving Benchmark (YCSB)* [11]. We consider Workload-A consisting of 50% reads and 50% writes, and Workload-B, which consists of 95% reads and 5% writes. Each data record consists of 1 KB of data, split into 10 fields. A read operation reads the whole record,

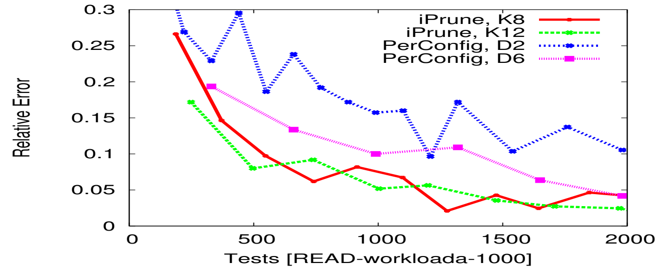


Fig. 9. Efficacy of iPrune and PerConfig for *Cassandra*, using YCSB's Workload-A and throughput $1000\frac{ops}{sec}$. For each choice of parameters (D/K and M), we calculate the relative error based on 90th percentile score. Each point represents the average error for a given choice of parameters across 200 runs.

while a write operation writes/updates one field at random.

We use 4 CPU types and 3 VM sizes (M, L, and XL) for a total of 11 configurations (we could not obtain LC). For each configuration, we obtain ground-truth data by starting 6 deployments (each of 2 nodes) in US-East. Each deployment is loaded with one million records (i.e., 1 GB of data, across the two nodes). Further, we disable replication so that we benchmark the baseline performance of *Cassandra*. We use a separate VM (of type `c1.xlarge`) for workload generation, and note that it has low utilization throughout the experiments. Each workload runs for 5 minutes, corresponding to up to 750K operations per run, for each deployment (an operation is either a read or write). Configurations are scored based on latency, and compared based on their 90th percentile score. In general, our data confirm the complexity of configuration selection - for a range of throughput, performance varies significantly across processor types. For instance, the instances with older CPU type A tend to perform better, and L instances perform worse than M instances for some CPU types.

Figure 9 compares iPrune and PerConfig schemes for *Cassandra*. iPrune is able to lower the number of tests required for 5% error target from 1800 to 1000 (> 40% reduction). Results for other workloads and throughput values show similar trends, though iPrune's benefits are more prominent for higher throughput values when performance difference across configurations is more significant.

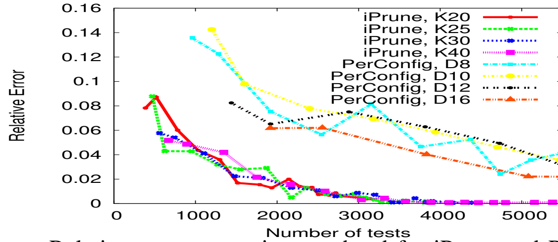
• **Other apps**: When evaluated on 29 CPU intensive apps (§II), iPrune achieves higher accuracy with fewer tests than PerConfig. For up to 100 tests, iPrune reduces relative error by more than 71% and 33% for the median and 90th percentile of runs, respectively (figure not shown for lack of space).

V. Short-listing configurations for testing

In contrast to pruning out poor configurations early on as iPrune does, another approach to save costs is to short-list good configurations and only test with them. Such an approach could be particularly valuable for short-running jobs where quick testing is required.

A. Short-listing based on similar applications

Measurements on CPU intensive apps in Figure 5 show that the best performing configurations for the same class of applications tend to be similar. Motivated by this, we



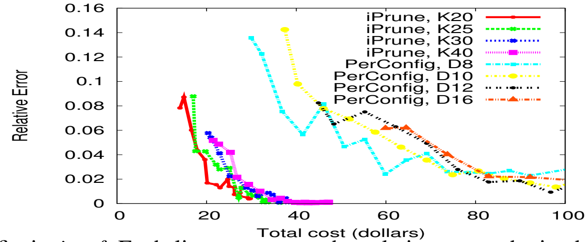


Fig. 8. Relative error vs. testing overhead for iPrune and PerConfig in *iperf*. Each line represents the relative error obtained for a given D (PerConfig) or K (iPrune) value. For each D and K value, we test the schemes using various number of measurements (M) per deployment. For each choice of parameters (D and M for PerConfig; K and M for iPrune), we record the resultant number of tests, testing cost and relative error (in the 90th percentile score). Each point represents the average relative error across 500 runs for a given choice of parameters.

evaluate a technique called nearest neighbor (NN for short) for short-listing configurations for a target application. NN requires the following as inputs: (1) a target application to choose a configuration for, (2) a set of other applications with prior configuration performance measurements, (3) a set of attributes for characterizing applications. The NN technique comprises the following steps: (i) for each attribute, normalize the application attribute values to the range of [0,1] to eliminate bias towards attributes with high absolute values; (ii) use the normalized attribute values to compute distances between the target application and every other application in the set; (iii) sort application distances in ascending order and obtain nearest neighbors; (iv) select top configurations of nearest applications to get short-listed configurations.

B. Evaluations of NN combined with PerConfig

To select the best configuration for a target application, we first short-list a set of good configuration candidates with NN, then use PerConfig or iPrune to test the target application on these candidates to pick the best. To characterize applications, we use `objdump` to disassemble the executable files into assembly code and use the counts of different instructions (e.g., mov|lea and add|sub|inc|dec) as attribute values. The results reported below are based on NN combined with PerConfig testing (NN_PerConfig for short) with $D5M1$ (5 deployments per configuration, and 1 measurement per deployment). Two variants of random selections are used as the baseline for comparison: randomly picking a configuration (random configuration) and selecting the best configuration of a randomly picked application (random application).

We demonstrate how the number of nearest applications and the number of top configurations affect accuracy of NN_PerConfig. We use *Nearest* n *Top* k (*NnTk* for short) to denote parameters in NN, which means using the top k configurations for each of the nearest n applications as candidates. To compare configurations, we use dollar cost per used core (e.g., dollar cost for an L VM is $\$0.175/2$) as the *score*, and the configuration with the lowest 90th percentile *score* for an application as the best one (can easily extend to other metrics). Relative error as stated in §IV-A is used to measure effectiveness. The average relative error out of 100 experiments is reported.

Network and compute intensive MPI FFT: MPI FFT is a distributed computation and communication intensive application. We use the CPU intensive apps as the other applications

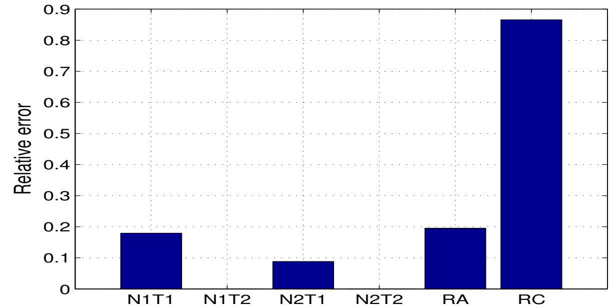


Fig. 10. Average relative error of NN_PerConfig when picking the best configuration for MPI FFT using 29 CPU intensive apps. RA means random application selection, RC means random configuration.

in NN_PerConfig to get a good-performing configuration for MPI FFT. As shown in Figure 10, NN_PerConfig outperforms both random strategies. More importantly, picking *Top 2* configurations reduces relative error to 0. The best configuration for MPI FFT ranks second for its nearest application (a biomolecular system simulator), and once the best configuration is included in the candidate set, a small number of PerConfig tests can reveal it. Thus, NN_PerConfig can reduce testing costs dramatically compared with basic PerConfig.

Other apps: Figure 11 shows the average relative error when only using the nearest application to choose top configuration candidates (*N1Tk*) for CPU intensive apps. For more than 60% of applications, the average relative error of NN_PerConfig is within 1% of *N1T1*. As the number of configuration candidates increases, the average relative error reduces. We have also experimented with including more nearest applications, and found that the average relative error is reduced sharply. Specifically, using *N2T1*, the worst case relative error falls below 2% (the worst case relative error for *N1T1* is over 15%). In contrast, using basic PerConfig to pick the best from all available configurations needs more than 100 tests to achieve a relative error less than 2%. NN_PerConfig is able to reduce overhead to 10% of the basic PerConfig overhead by short-listing configurations.

VI. Related work

Many studies [12], [15], [16], [17], [19], [20], [22], [23], [25] have observed that application performance can vary greatly in the cloud. [22] shows how computational load on physical hosts impacts network performance, but does not

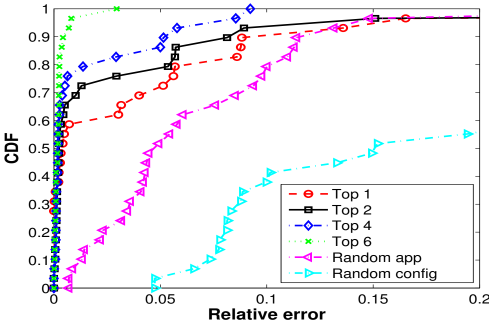


Fig. 11. Average relative error of NN_PerConfig for 29 CPU intensive apps using the nearest application and different numbers of top configurations, compared with random selection strategies.

consider heterogeneity of CPU types and its impact. [12], [19] observe that CPU heterogeneity partly explains variation in performance across the same sized VMs, but do not consider the impact of provider policy. Our work is distinguished in that we *simultaneously* consider VM sizes and CPU types to understand performance variability, and by our focus on provider policy and configuration selection.

Further, [12], [19] evaluate "trial and error" strategies for optimizing performance that involve starting exploratory VMs, and shutting down and replacing VMs with CPU types that do not perform well. Kingfisher [21] uses heavy weight testing to profile the cost-performance of different VM sizes (but it is CPU type-agnostic) and migrates an application from one configuration to another over time. In contrast, we focus on systematically determining the best configuration for an application and reducing the test costs considering a more general formulation where the cloud provider allows customers to choose from arbitrary combinations of CPU types and VM sizes. This approach is advantageous when dealing with stateful applications (e.g., Cassandra) that are tricky to migrate, jobs of moderate length, or when it is desirable to avoid potentially long convergence time during which performance could be poor.

VII. Conclusions

Our findings have demonstrated the importance of considering the interplay among provider policy, configuration choice, and application performance, and have revealed quite a few surprises – e.g., larger VM sizes may not necessarily see higher rate limits; for the same VM size, inter-DC TCP throughput may be markedly different; VMs with larger VM size sometimes perform consistently worse than VMs with smaller VM size; and more. Further, our many auxiliary experiments are able to shed light on the underlying reasons for these findings. These measurement results have exposed the complexity of selecting configurations in cloud settings and the limitations of simple rules of thumb.

This has motivated us to characterize the extent of testing required to pick a configuration with a desired performance for a range of strategies. Our results show the promise of iPrune and NN techniques in reducing testing overheads without sacrificing accuracy. While PerConfig might suffice for long-running jobs with few configuration dimensions, we expect the need for techniques such as iPrune and NN to be more critical for jobs of moderate length, or as the number of configurations grows (e.g., with more hardware dimensions, or when considering multi-tier applications with hundreds of application components). For extremely short-running jobs, techniques like NN that involve no testing are appropriate.

In the future we hope to collaborate with practitioners to gain experience with our techniques in large-scale production deployments, and in environments that expose a wider set of configuration choices, and repeat our experiments on other cloud platforms.

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) under Award No. 1162270 and 1162333. 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.

References

[1] Amazon discussion forums, connection speed. https://forums.aws.amazon.com/thread.jspa?messageID=77314.
[2] Amazon discussion forums, dedicated instances. https://forums.aws.amazon.com/message.jspa?messageID=529555#529555.
[3] Amazon discussion forums, number of cores of small VM. https://forums.aws.amazon.com/message.jspa?messageID=530348#530348.
[4] Amazon discussion forums, the attempt to equalize CPU capacity. https://forums.aws.amazon.com/thread.jspa?messageID=530539.
[5] Credit-scheduler in Xen. http://wiki.xen.org/wiki/Credit_Scheduler.
[6] Digging deeper in EC2. http://goo.gl/bmcI4V.
[7] Serverfault, bandwidth limits for Amazon EC2. http://serverfault.com/questions/460755/bandwidth-limits-for-amazon-ec2.
[8] The Apache Cassandra project. http://cassandra.apache.org/.
[9] Xen 4.2: New scheduler parameters. http://blog.xen.org/index.php/2012/04/10/xen-4-2-new-scheduler-parameters-2/.
[10] XenStore Reference. http://wiki.xen.org/wiki/XenStore_Reference.
[11] Brian F Cooper et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.
[12] Benjamin Farley et al. More for your money: Exploiting performance heterogeneity in public clouds. In *SoCC*, 2012.
[13] Sally Floyd et al. Equation-based congestion control for unicast applications. In *SIGCOMM*, 2000.
[14] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric statistical inference*, volume 168. CRC press, 2003.
[15] A. Iosup et al. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE TPDS*, 2011.
[16] K.R. Jackson et al. Performance analysis of high performance computing applications on the amazon web services cloud. In *CloudCom*, 2010.
[17] A. Li et al. CloudCmp: comparing public cloud providers. In *IMC*, 2010.
[18] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.
[19] Zhonghong Ou et al. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *HotCloud*, 2012.
[20] Jorg Schad et al. Runtime measurements in the cloud: observing, analyzing, and reducing variance. In *VLDB endowment*, 2010.
[21] Upendra Sharma et al. Kingfisher: cost-aware elasticity in the cloud. In *IEEE ICDCS*, 2011.
[22] Ryan Shea et al. A Deep Investigation Into Network Performance in Virtual Machine Based Cloud Environment. In *INFOCOM*, 2014.
[23] G. Wang et al. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.
[24] Yunjing Xu et al. Bobtail: Avoiding long tails in the cloud. In *NSDI*, 2013.
[25] M. Zaharia et al. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.