On the Performance of Virtualized Infrastructures for Processing Realtime Streaming Data

Kathleen Ericson and Shrideep Pallickara
Colorado State University
Computer Science Department
Fort Collins, USA
{ericson, shrideep}@cs.colostate.edu

***Abstract*—Clouds have become ubiquitous and several data processing tasks have migrated to these settings. The dominant approach in cloud settings is to provision virtual machines (VMs) rather than provision direct access to the physical machine. One artifact of such provisioning is that multiple VMs may be collocated on the same physical machine and possibly interfere with each other. In this paper, we focus on the impact of virtualized infrastructures on realtime stream processing; we use the classification of electrocardiograms (ECG) as a motivating example. Stream processing in such a setting strains resources differently than the traditional web services or analytics on large datasets traditionally performed in the cloud. In streaming environments all processing per packet needs to be completed in a timely manner, and the number and rate at which these packets are generated is high. Our focus is to study the implications of various combinations of virtualization strategies on the performance of realtime stream processing. We have done extensive performance benchmarks (using Xen and KVM) the results of which form the basis for our recommendations for the trade-offs involved in these settings.**

Keywords-classification, health streams, Granules, Xen, KVM, virtualization, stream processing.

I. INTRODUCTION

Health sensors are becoming pervasive as devices become smaller and cheaper. These devices are invaluable in allowing people to keep out of hospitals and maintain their normal lifestyle when illnesses or age would previously have mandated a regulated living environment. In such a situation, a user would have multiple sensors constantly generating information about their well-being. Such sensors may include monitors for ECG, EEG, blood pressure, and even gyroscopes to determine user orientation. In the past we have worked with classifying EEG streams generated by electrodes placed on a person's scalp to allow them to interact with their environment [1]. Here we focus on classifying ECG data, determining if an abnormal heartbeat has occurred.

To keep up with the expected growth of health sensors we explore the capabilities of virtual machines in the context of stream processing. By moving from a standalone solution to a cloud of virtual machines we not only increase the amount of simultaneous users we are able to support, but we also are able to take advantage of data aggregation that has the potential to improve the accuracy of the inference algorithms operating on such data. Instead of only being able to build algorithms which monitor user health from a small pool of people, these algorithms could learn from thousands of users across a wide geographic region – possibly finding connections that lead to breakthroughs in the diagnosis of medical problems.

In this work our focus is to determine the suitability of virtualized machines when working with streaming data while placing time constraints on processing. Processing such streaming data tends to strain resources differently from static settings: packeets arrive continuously with processing being performed for milliseconds per packet by memory-resident computations that are activated when data arrives. When anomalous events are detected, packets from the near past are written to disk for further study. For these experiments, we work with both Xen [2] and kernel virtual machine (KVM) [3] hypervisors. We are using a classifier to detect anomalous heartbeats among streaming ECG data. The goal is to return a classification in real-time (e.g. before the next second of ECG data is passed to the classifier). Our experiments analyze performance across both dedicated machines and virtualized resources. As cloud computing becomes ever more pervasive, it is important to analyze the capabilities of various virtual environments for processing this class of data in real-time.

We use the MIT-BIH arrhythmia dataset [4], which has been used to train specialists to detect various types of arrhythmias. Our computation consists of an R-based classifier which determines whether a segment of ECG data contains an abnormal heartbeat or not. An ECG signal comprises the PQRST wave constructed from electrical activity reported by electrodes placed on the chest. Each complete wave represents a heartbeat. This is distinct (though related) from the heartbeat heard using a stethoscope. If a heartbeat is flagged as abnormal, the proceeding and following heartbeats should be stored to disk for further analysis.

For our communications substrate, we use the Granules [5, 6] stream processing framework. Granules runs as a daemon on a resource and can interleave multiple computations on the same resource. The framework allows us to host low-overhead computations that only activate when data is available on the streams that they consume. The Granules framework also incorporates the ability to use code written in different languages as computations – namely our R-based classifiers.

Our experiments with virtualization involve both fully virtualized and paravirtualized resources. *Full virtualization* hypervisors run guest operating systems without any code modification by creating virtual interfaces for all hardware needs. While this means less up-front work to achieve a running virtual machine (VM) instance, it also leads to a potential slowdown in accessing and utilizing resources such as RAM, CPU, and disk and network I/O. *Paravirtualization* requires some operating system modification, but offers performance near that of the physical machine. In our experiments we look at KVM, which only runs in a fully virtualized mode, as well as Xen, which has both full- and para-virtualized modes.

Virtual machines are typically thought of as resources to allow web services to scale as needed. They have also been used in conjunction with Hadoop [7] for large MapReduce [8] operations when a user does not have enough resources to perform the operation locally. While web applications typically require very fast response times, they do not usually require a large amount of processing on the back end. Alternatively, MapReduce applications tend to require a large amount of processing, yet have no bounding time requirements. Our application requires both a timely response and a large amount of processing. Each packet requires processing in the order of milliseconds, but there may be millions of packets coming in to a single machine. In this respect, we are pushing these hypervisors to their limits in our experiments.

A. Paper Contributions

To the best of our knowledge, this paper represents a first exploration of the performance of various hypervisors in a stream processing context. Stream processing requires the quick and efficient processing of relatively small computations at a very large scale. In a stream processing environment it is necessary to process data at a rate no worse than the rate at which data is being generated. If data is regularly processed too slowly, incoming buffers will eventually fill and then overflow: causing a loss of irreplaceable data. Because of this, we need to ensure the reliable and timely completion of all processing. A major goal of this work is to develop recommendations for hypervisor choice for supporting stream processing.

We gather data about hypervisor behavior not only in an idealized situation, but also under heavy loads. As we outlined earlier, the computations stress several capabilities available at a resource. By running these stress tests we get to isolate the minimum requirements of the hypervisors themselves, as well as determine how performance deteriorates in the face of interference.

In a virtualized environment, *interference* from collocated machines is a given. In most situations, users have no control or knowledge of what other computations are currently running on the same physical machine. When multiple computations that share similar resource requirements are placed on the same physical machine, performance of all VMs on the machine may be negatively affected. By understanding how quickly this falloff in performance occurs, we can develop methods to detect and recover from such performance problems.

B. Paper Structure

The rest of this paper is organized as follows; Section II discusses the technologies used here as well as related work, while Section III describes our experimental approach and introduces the dataset we use in our experiments. In Section IV we work through several baseline experiments, then scale up to stress tests in Section V. We then discuss our conclusions and future work in Section VI.

II. Background and Related Work

Granules [5, 6] is a stream processing framework which allows processes to enter a dormant state between rounds of execution. Granules further allows computations to build and maintain state over time. Because of these features, Granules is particularly suitable for sensor processing and has been deployed in domains such as EEG classification [1], epidemic modeling using discrete event simulations, and handwriting recognition [9].

Xen [2] was developed as an attempt to leverage the benefits of virtualization without needing special hardware and minimizing the changes to existing operating system code. Xen supports both full virtualization and paravirtualization modes, meaning Xen can be run with full virtualization on hardware that does not support paravirtualization. To allow operating system code to run with minimal modifications, the Xen host OS implements strong resource isolation which creates the illusion that each guest OS is running on a bare machine. Xen can support both Linux and Windows guests, provided an altered operating system kernel is made available.

KVM [3] is a fully virtualized solution to virtual machine provisioning. It is built into the Linux kernel, and does not require any special hardware – this makes it a simple and effective solution to virtualization, particularly in a Linux environment. One weakness of KVM is in how it handles I/O operations. Since it is fully virtualized, virtual machines (VMs) do not have direct access to I/O interfaces and need to make calls out to the host OS in order to read and write from network and disk interfaces, leading to a high overhead for these types of operations.

While other works [10, 11] have performed various benchmarks with both Xen and KVM implementations, it has been several years since comprehensive benchmarks were last performed, on Xen 3.1 and KVM-60 in [10] and Xen 3.2.1 and KVM-83 for [11]. Furthermore, these benchmarks were designed to explore generic OS functionality, and do not reflect the requirements inherent in stream processing. Stream processing differs from generic processing in several ways. Most notably are the strict turn-around times required for stream processing: if we cannot process streaming data at least as quickly as it is being generated we risk losing vital information as buffers overflow.

Eucalyptus [12] is an Infrastructure as a Service (IaaS) provider, and essentially manages groups of VMs hosted across a physical cluster. It operates on a higher level than the Xen and KVM hypervisors, and will handle the scheduling of jobs across a pool of VMs. In this work we focus on exploring the pros and cons of various hypervisors in the context of

distributed stream processing, so Eucalyptus is outside the scope of our work.

Hadoop [7] is an open source implementation of MapReduce [8], and often the defacto standard for cloud processing benchmarks. Hadoop is not, however, designed to handle streaming data and instead is designed to perform distributed computations on a large amount of data stored on disk in the form of files staged using the Hadoop Distributed File System (HDFS) [13]. Instead, we are using Granules, which has been designed expressly for performing arbitrary computations on streaming data.

Works exploring virtual machines in the context of stream processing systems have focused on the specific problem of more efficiently sharing communications between collocated VMs [14, 15]. These works assume a set hypervisor and developed patches which help to speed up processing in certain situations. KVM and Xen guest performance is neither directly compared nor explored. These expansions have since been incorporated into their respective code bases.

The MIT-BIH arrhythmia dataset has been used primarily as a dataset in the area of machine learning [16-18]. Our work moves in a different direction as the goal of this paper is not to develop a highly trained ECG classifier, but instead explore the behavior of VMs under different hypervisors in a streaming environment. While we use our own trained neural networks for ECG classification, Granules can schedule the processing of arbitrary computations in a stream pipeline. There is nothing to preclude the incorporation of a more rigorously trained and tested classification system into our pipeline. A different classifier would potentially involve a different processing footprint, yet another ANN such as that described in [16] would have an identical processing footprint as our approach proposed here.

III. Experimental approach

For our experiments we use quad-core machines with 12GB RAM and 50GB disk space. Each core is capable of hyperthreading, so we effectively have 8 cores available on each machine. For the virtualized machines, each is given 2.5GB RAM, 8GB of disk space, and 2 cores.

For both hosts and guest operating systems, we are using Fedora 16 (Verne). For Xen experiments we use Xen 3.4.2, and for KVM we are using the version bundled with kernel 3.1.0. When configuring Granules resources, we are allocating the JVM 2 GB RAM and 2 threads for the VM runs, and 8 GB RAM with 8 threads for the pure physical machine runs – effectively four times the resources available to a single VM.

A. Dataset

The MIT-BIH data set [4] is currently hosted by PhysioNet [19]. This dataset has been accumulated from 1975-79, and is the arrhythmia dataset used to train doctors to recognize various arrhythmias. The data is gathered at a sampling rate of 360Hz from a pair of sensors. All patients had an upper and lower lead sensor. The upper sensor is better at detecting the full QRS complex, while the lower sensors are useful for the detection of ectopic beats.

This dataset contains an annotation for each heartbeat, marking each beat as normal, abnormal (in which case the exact type is noted), or a decrease in signal quality which makes classification difficult. We are specifically interested in classifying each beat as either normal or abnormal, allowing us to isolate abnormal beats and store them to disk for further analysis.

Our ECG processing computation has 2 parts: First, we use a windowing mechanism to store the last 10 seconds of ECG signals in raw data format. Second, the raw data for each heartbeat is passed to an Artifical Neural Network (ANN) [20] for classification. Our neural network contains a single hidden layer and that contains 10 hidden units. We perform training offline, and then load a trained neural network into a computation when we are ready to run classification tests. The neural network is trained to classify one second bursts of ECG data as either normal or abnormal.

Should the last heartbeat be abnormal, the stored 10 second window of ECG data is written to disk. Every incoming beat is then stored on disk until 10 seconds have passed without any abnormal beats. This approach allows us to store 'interesting' data to disk, allowing a healthcare professional to later analyze data leading up to a potential abnormal heartbeat, as well as any immediately following data – allowing the arrhythmia to be viewed in context.

It is particularly important to store data from the perspective of a health sensor monitoring system as data may be used at a later date to assess new diagnoses. These snippets of interesting data can also be used as a new training set for machine learning approaches; for example, a classifier based on support vector machines (SVMs) [21] could be devised. Correctly classified abnormal data gives us specific windows of interest – possibly allowing our algorithms to learn patterns which lead up to various arrhythmias. Even if the data was misclassified and normal heartbeats are saved, it becomes useful for future rounds of training as that particular portion of data was obviously difficult for the current classifier to correctly classify. While there is an argument to storing all data – as it all may be useful for doctor analysis and the training of future classifiers – it is simply infeasible to assume that all data can be stored since the storage requirements will quickly outpace available disk capacity.

IV. Baseline Experiments

For our baseline experiments, we work with three physical machines. One machine has a base Fedora 16 install, one is a KVM host, and the third is a Xen host. Each host (both Xen and KVM) manages 4 guest operating systems with minimal Fedora 16 installs. For all tests, the JVM is configured to use up to 2GB of RAM and run two threads. This means that the JVM has the exact same resources as on each VM, giving us a more fair comparison of overheads.

First, we need to determine the baseline performance of a single monitoring computation on the base physical machine well as a single guest VM in both KVM and Xen. We first looked at the computation processing times, the results of which are displayed below in TABLE I. This benchmark encompasses the amount of time spent after receiving a

message, before a response is generated and sent back to the user, but not the amount of time required to interpret the message to determine whether it is incoming data or an internal communication (notices of machine failure, state passing between replicas, etc.).

TABLE I. COMPUTATION PROCESSING TIMES FOR A SINGLE MONITOR(IN MS)

Approach	Mean	Min	Max	SD
Base	304.71	228.66	529.75	45.704
KVM	333.11	248.06	577.35	49.612
Xen PV	317.05	224.92	584.45	48.14
Xen FV	340.75	250.54	585.14	51.950

From these results, we see a best case performance in the case of the base install – this is expected since it is running on the physical machine. There is no hypervisor introducing overheads by doling out resource accesses. While the paravirtualized Xen (Xen PV hereafter) is generally outperforming KVM, it does have a worse maximum runtime. The fully virtualized Xen (Xen FV hereafter) is simply performing poorly across the board.

The amount of time interpreting the message, as well as the communications overheads are the difference between the round-trip communications times which are displayed below in TABLE II. and the computation processing time displayed above in TABLE I.

The paravirtualized Xen again performs closest to the baseline physical machine, followed by KVM and the full virtualization Xen instance. While Xen with full virtualization had a worse mean and minimum communications time, it also had a lower maximum and standard deviation than KVM. The previous results show that KVM can perform better for the CPU intensive processing, but here we see that even the full virtualization Xen has more reliable performance once we take into account networking-based operations.

TABLE II. ROUND TRIP COMMUNICATIONS FOR A SINGLE MONITOR (IN MILLISECONDS)

Approach	Mean	Min	Max	SD
Base	349.54	272.80	716.58	48.097
KVM	379.11	292.88	774.31	52.139
Xen PV	361.65	269.83	607.82	47.86
Xen FV	385.39	294.83	629.80	51.998

For this round of experiments, we enabled disk writes, meaning that the computation will store any inputs flagged as abnormal to disk. This part of the experiment stresses short, repeated writes to disk, the results of which can be seen in TABLE III. Here both the para- and full virtualization Xen approaches outperformed KVM, even achieving a lower mean than the base Fedora install. The Xen hypervisor is obviously doing a better job of streamlining writes to disk than KVM. Given KVMs added overheads for I/O operations, we expected to see this behavior.

These tests reinforced the original expectation that Xen performs best in situations with many writes and networking requirements. We also saw that Xen does not perform as well for CPU bound operations. One reason why we may be seeing this is that the Xen hypervisor is simply more strict in ensuring isolation between VMs. For these tests, only one VM was running, while the other 3 were simply idling. It is possible that KVM is allowing the one running VM to monopolize the processor, while Xen enforces a stricter scheduling policy, meaning it can't take advantage of spare cycles when other VMs are idling.

TABLE III. SHORT WRITE TIMES FOR A SINGLE MONITOR (IN MILLISECONDS)

Approach	Mean	Min	Max	SD
Base	1.55	0.88	16.16	1.372
KVM	3.23	1.94	23.67	3.06
Xen PV	1.24	0.90	16.35	1.035
Xen FV	1.52	0.91	20.75	1.861

A. Single Physical Machine Stress Tests

After analyzing the footprint of a single monitoring computation, we moved on to determine the maximum number of monitors we could support on a single physical machine (no virtualization), as well as the number supportable on a single VM (both in KVM and Xen). Knowing the limits of the single physical machine gives us a guideline as we scale up to using all 4 VMs on a single physical machine.

TABLE IV. RESPONSE TIMES (IN MILLISECONDS) FOR A SINGLE PM

Monitors	Mean	Min	Max	SD	%Failure
12	822.22	348.01	1819.22	174.798	14.46
10	689.91	313.87	1490.86	129.312	2.04
9	621.16	333.77	1246.17	107.659	0.47
8	545.97	310.18	1088.39	83.260	0.13

We ran several tests with 12, 10, 9, and 8 concurrent monitors running on a single machine. For these tests we configured the JVM to use 8 threads and allowed it a maximum of 8GB of RAM. We chose these settings as they are four times what one of our VMs are configured to handle.

TABLE IV. shows the round-trip response times for monitors on a single physical machine. A message is deemed a failure if it takes longer than one second (1000 ms) to return to the user. At first, the failure rate drops rapidly as we move from 12 to 10 concurrent ECG monitoring computations, with limited returns as we reduce the number of concurrent monitors to eight.

To investigate this further, we looked at the probability distribution of our round-trip results. From Figure 1. we can see that as we reduce the number of ECG monitors, we are not only slowly shifting our average response to the left (quicker responses), we are also lowering the standard deviation dramatically (narrower peaks). Looking back at TABLE IV. we can corroborate the steady decrease in both standard deviation and mean runtime as we move from 12 to 8 monitors.

Based on these results from a physical machine, we know that a solution involving VMs should aim to support about 8 concurrent monitors (in total). While we expect decreased performance as we move to VMs, we still hope to maintain a less than one percent failure rate.

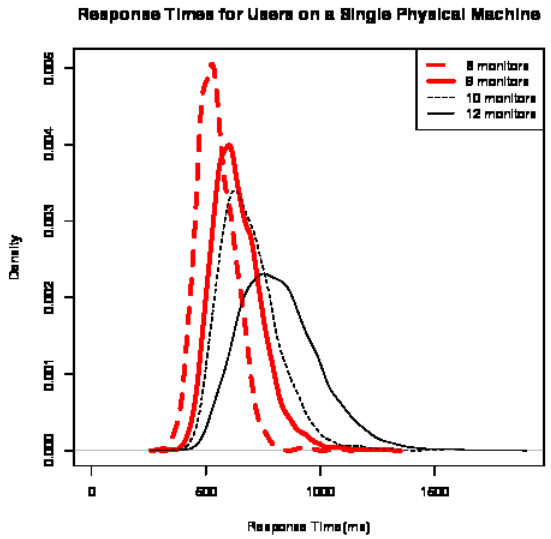

Figure 1. Density of response times for 8, 9, 10, and 12 concurrent monitors on a single physical machine.

B. *Single Virtual Machine Stress Tests*

For these experiments, we focus on stress testing a single virtual machine. Unless otherwise noted, the other 3 VMs on the same physical machine are simply idling - they have been provisioned and are running, but no computations are occurring on them.

First, we tried supporting three concurrent monitors in a single VM. While this is probably not sustainable as we move up to using all VMs on the physical machine, as the base install physical machine could not support 12 concurrent users, it is worthwhile to determine how our setup behaves when overloaded. These results can be seen below in TABLE V.

TABLE V. RESPONSE TIMES (IN MILLISECONDS) FOR A SINGLE VM WITH 3 MONITORS

Approach	Mean	Min	Max	SD	%Failure
KVM	558.77	284.25	1143.98	147.586	0.78
Xen PV	596.28	273.90	1478.85	183.12	3.81
Xen PV (paused)	552.70	295.27	1290.84	148.610	1.35
Xen PV (shutdown)	540.00	263.28	1510.87	145.86	1.17
Xen FV	553.66	291.88	1413.83	152.503	1.62
Xen FV (paused)	543.01	279.36	1302.60	144.56	1.22
Xen FV (shutdown)	524.23	282.30	1173.89	130.833	0.66

In this experiment, we were expecting to see Xen PV outperforming KVM, since Xen offers paravirtualization support instead of the fully virtualized KVM. While we do see that Xen with paravirtualization has a lower minimum runtime, the mean, maximum, standard deviation, and failure rate are higher with Xen PV than KVM. One theory is that the Xen hypervisor is again attempting to be too fair in scheduling, and is restricting the CPU usage (our bottleneck) of the single VM which is performing computations in order to give the other VMs (which are idling) a chance to submit data for processing. The KVM hypervisor, on the other hand, seems to be much more willing to allow the single running VM to monopolize the queue.

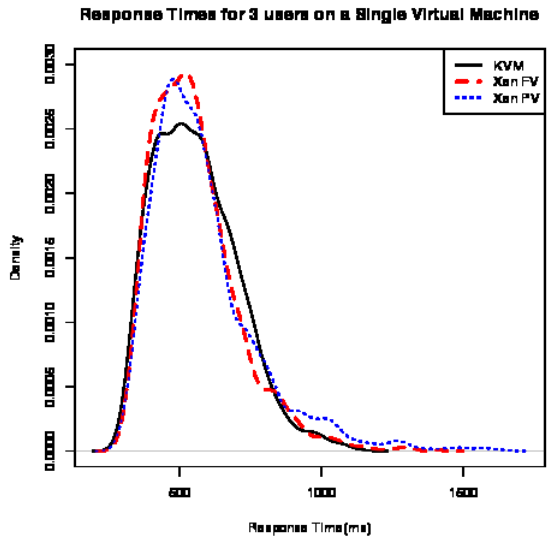

Figure 2. Density of response times for 3 users on a single Virtual Machine hosted by KVM, Xen with full virtualization, and Xen with paravirtualization

Of interest in these results is the fact that the full virtualization Xen install is actually outperforming the paravirtualized version. From the results in TABLE V. it's not entirely clear why this is happening. By looking at the density of the response times in Figure 2. we can get more insight into this behavior. All implementations appear to have an almost bimodal distribution, which can skew our mean and standard deviation calculations. From the figure, the paravirtualized Xen guest is actually returning results most often just below the 500ms mark, clearly outperforming the KVM implementation with results very near the Xen guest with full virtualization.

As we discussed previously, one possibility as to why KVM can outperform Xen is due to Xen adhering to a strict CPU scheduling algorithm. In order to test this hypothesis, we ran the Xen tests again, attempting to force the hypervisor to reduce the amount of time reserved for the idle VMs. We tried both pausing and shutting down the idle VMs for both of these tests.

With paravirtualized Xen (Figure 3.), we saw an all-around increase in performance, though even with three shutdown VMs (our best case scenario) the percentage of failures is larger than we saw with KVM. In Figure 4. we show the results from running the same experiment with Xen running full virtualization. With the other three VMs stopped, we finally achieve a lower failure rate than we see with KVM. We found it interesting that the full virtualization Xen approach actually managed to perform better than the paravirtualized Xen. Obviously the full virtualization approaches are taking advantage of something that the paravirtualized approach cannot.

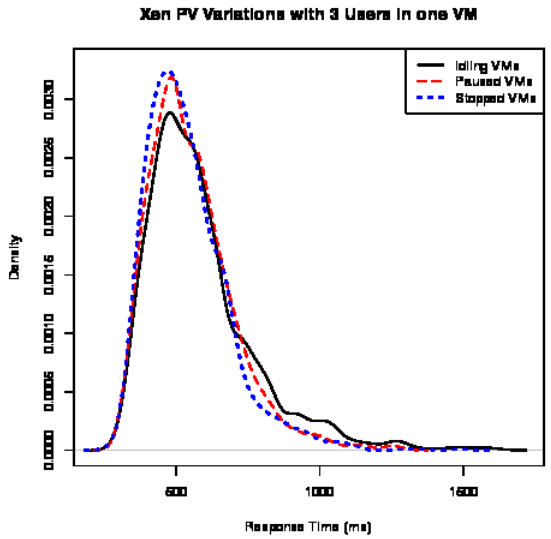

Figure 3. Density of response times for 3 users on one VM with paravirtualized Xen variations

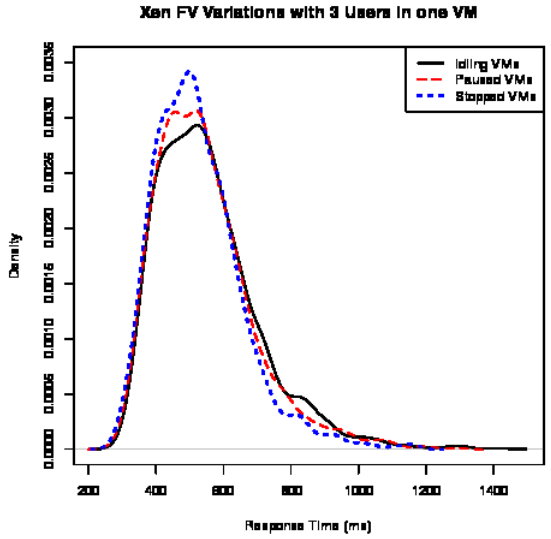

Figure 4. Density of response times for 3 users on one VM with full virtualization Xen variations

TABLE VI. RESPONSE TIMES (IN MILLISECONDS) FOR A SINGLE VM WITH 2 MONITORS

Approach	Mean	Min	Max	SD
KVM	397.51	283.61	903.50	58.077
Xen PV	412.13	270.29	882.83	90.532
Xen PV (paused)	399.30	249.35	809.42	75.98
Xen PV (shutdown)	396.66	277.32	813.85	78.649
Xen FV	412.49	279.63	830.87	72.011
Xen FV (paused)	398.39	272.37	930.90	66.027
Xen FV (shutdown)	387.13	263.35	919.79	60.833

Next, we looked at the performance of KVM and Xen with only 2 computations on each VM. For these tests, no failures occurred, so we removed the failure percentage column from the results in TABLE VI. As before, we also tried the Xen experiments with the three idle VMs both paused and shutdown.

Even after pausing and shutting down the idle VMs, KVM is outperforming both Xen approaches in these experiments with a much lower standard deviation. With the idle VMs shutdown, both Xen approaches do manage to achieve a lower mean runtime, but still maintain a higher standard deviation.

To further analyze our results, we graphed the density of response times for KVM and Xen. This can be seen in Figure 5. Interestingly, several of these density curves appear to be almost bimodal, most clearly seen in the KVM approach, but also apparent in the Xen with full virtualization approach. Interestingly, the paravirtualized Xen results do not show this bimodal behavior, just a slightly wider curve than we have seen previously.

These bimodal ridges are not present in the base physical machine experiments, so must be an artifact of the interference of the hypervisor in the Xen FV and KVM experiments. There seems to be some extra layer of interference found in the fully virtualized approaches that is not in the base approach and at least is much less apparent in the paravirtualized approach.

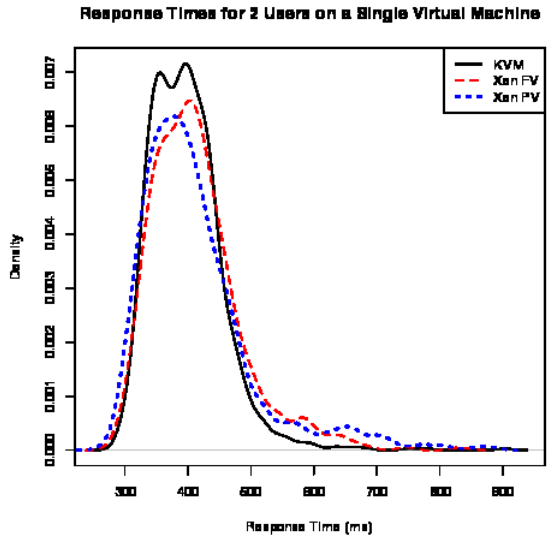

Figure 5. Density of response times for 2 monitors run in VMs controlled by KVM and Xen

In Figure 6. the probability densities of responses for the paravirtualized Xen are displayed side by side. Even as the other VMs are paused and shutdown, this approach still maintains a normal distribution. This further gives credence to the theory that something is occurring in the full virtualization hypervisors which is not being seen in either the bare machine or paravirtualized instances.

Figure 7. shows the detailed results of variations on the full virtualization Xen. While the paused and stopped VM approaches are not as obviously bimodal as the original results, there is still the hint of it given the uneven falloff of the density ridges. We are also seeing a steady procession of improvements as we pause and then stop the other VMs –

clearly the Xen hypervisor does reserve CPU cycles for VMs even when idling and paused.

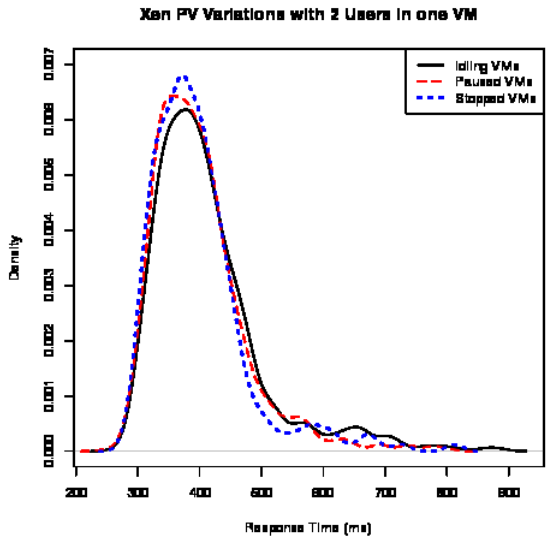

Figure 6. Density of response times for 2 users on one VM with paravirtualized Xen variations

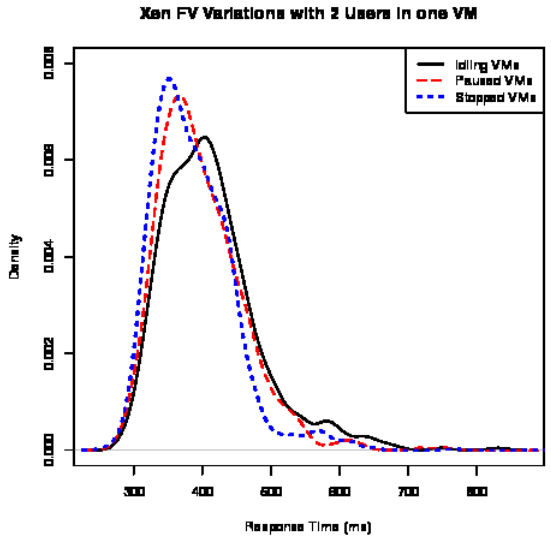

Figure 7. Density of response times for 2 users on one VM with full virtualization Xen variations

V. Full Machine Experiments

After determining baseline performance in the previous section, we next looked at scaling up to full machine experiments. As we add more VMs and computations, each begins to compete for limited resources, and interesting behavior emerges. Here we examine where the bottlenecks are occurring and why.

First, we attempted to support 8 concurrent users with one physical machine hosting 4 VMs. This means 2 computations running concurrently on each VM. From the bare-bones approach, we could only support up to 9 concurrent users with a less than one percent failure rate; and from the single VM experiments, we can support 2 concurrent users without any failures. The results of these tests are displayed in TABLE VII.

TABLE VII. Response Times (in milliseconds) for 8 monitors across 4 VMs

Approach	Mean	Min	Max	SD	%Failure
KVM	754.29	322.45	1584.89	127.954	3.64
Xen PV	781.63	344.98	1400.88	131.174	5.33
Xen FV	785.37	370.47	1529.54	133.802	5.84

Again, we are seeing that KVM is outperforming the Xen approaches with our CPU intensive task. We again believe that this is a problem with the Xen hypervisor using a different strategy to apportion CPU times. In this particular case, we believe that Xen is being outperformed by KVM because the hypervisor does not have enough processing available to handle the context switching. A detailed view of the response times is shown below in Figure 8.

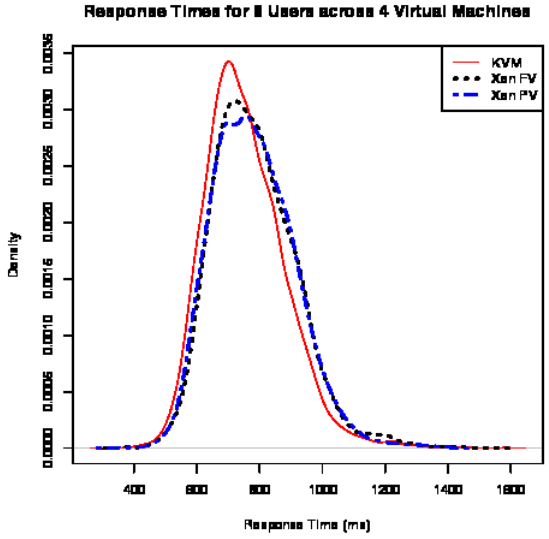

Figure 8. Density of response times for 8 users across 4 virtual machines

To test our hypothesis that Xen is being outperformed by not leaving enough cycles free for the hypervisor, we attempted to support 6 users across 3 VMs, leaving the fourth VM idle on each machine. This will leave more free cycles available to the hypervisor, so we should see that Xen is outperforming KVM. These results are shown below in TABLE VIII.

TABLE VIII. Response Times (in milliseconds) for 6 monitors across 3 VMs

Approach	Mean	Min	Max	SD	%Failure
KVM	574.94	287.94	1500.92	164.615	1.02
Xen PV	588.93	286.21	1079.87	111.509	0.12
Xen FV	596.73	315.34	1234.93	118.826	0.47

Now we see Xen outperforming KVM overall, though it still has a worse minimum and mean runtime than KVM. To better understand these results, we graphed the density of responses in Figure 9. Here, we see that KVM has a distinct bimodal appearance, with a rather large standard deviation. As this behavior doesn't appear in the Xen with full virtualization,

it seems to be an artifact of KVM instead of due to full virtualization. A major takeaway from these results is that Xen behaves much more reliably when the hypervisor has access to a dedicated CPU, as evidenced by the drastic decrease in failures (a drop from 5.33% and 5.84% to less than 1%) as we move from overloading all 4 VMs to only using three.

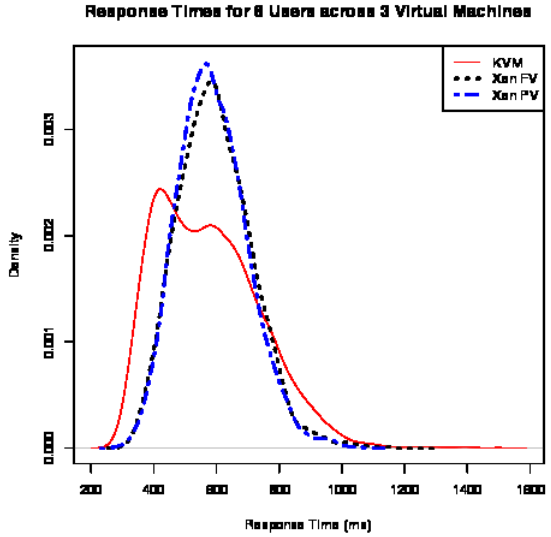

Figure 9. Density of response times for 6 users across 4 virtual machines

VI. Conclusions and Future Work

With this work we have taken a first step in evaluating virtual machines in the context of stream processing. Our computation relating to classifying ECG data in real time involves high CPU overheads, a comparably small (100s of KB) memory footprint, moderate networking requirements, and steady bursts of short disk writes. Stream processing computations typically have a moderate to large networking footprint – data is regularly sent to the node for processing, and results need to then be pushed out to either the user or another node for further processing. Xen generally performs better with network communications, and the paravirtualized approach performs on par with a bare machine implementation for sustained, short writes.

We also saw that KVM performed better with CPU intensive tasks – often outperforming even the paravirtualized Xen in many experiments. As we moved on to the virtual machine stress tests, we found that KVM generally outperforms both para- and fullvirtualization Xen implementations when the physical machine is overloaded.

Based on our experiments, we have two recommendations for the selection of hypervisor in the context of stream processing. In situations where there is control over the placement of VMs and collocated computations, the Xen hypervisor provides very good response times. Once the hypervisor begins to become overloaded, however, KVM becomes the better choice.

The work described here provides a basis to identify types of computations that are suitable for collocation and move towards intelligent computation placement. By defining different types of computations based on their networking, disk, RAM, and CPU requirements we can determine which types are most likely to interfere with each other and take measures to minimize collocation. This should help to reduce variance in response times, and reduce failure rates across the cluster by minimizing buffer overflow.

References

[1] K. Ericson, et al., "Analyzing Electroencephalograms Using Cloud Computing Techniques," in IEEE Conference on Cloud Computing Technology and Science, Indianopolis, USA, 2010.

[2] P. Barham, et al., "Xen and the art of virtualization," presented at the Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA, 2003.

[3] A. Kivity, et al., "kvm: the Linux Virtual Machine Monitor," in Linux Symposium, Ottawa, Canada, 2007, pp. 225-330.

[4] M. GB and M. RG, "The impact of the MIT-BIH Arrhythmia Database," IEEE Eng in Med and Biol, vol. 20, pp. 45-50, May-June 2001 2001.

[5] S. Pallickara, et al., "Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce," in IEEE International Conference on Cluster Computing, New Orleans, LA., 2009.

[6] S. Pallickara, et al., "An Overview of the Granules Runtime for Cloud Computing," in IEEE International Conference on e-Science, Indianapolis, 2008.

[7] T. White, Hadoop: The Definitive Guide, 1 ed.: O'Reilly Media, 2009.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ACM Commun., vol. 51, pp. 107-113, Jan. 2008 2008.

[9] K. Ericson, et al., "Handwriting Recognition using a Cloud Runtime," in Colorado Celebration of Women in Computing, Golden, 2010.

[10] A. Chierici and R. Veraldi, "A quantitative comparison between xen and kvm," Journal of Physics: Conference Series, vol. 219, p. 042005, 2010.

[11] C. Jianhua, et al., "Performance Measuring and Comparing of Virtual Machine Monitors," in Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on, 2008, pp. 381-386.

[12] D. Nurmi, et al., "The Eucalyptus Open-Source Cloud-Computing System," in Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on, 2009, pp. 124-131.

[13] D. Borthakur. (2007). The Hadoop Distributed File System: Architecture and Design. Available: http://hadoop.apache.org/common/docs/r0.18.0/hdfs_design.pdf

[14] J. Liu and B. Abali, "Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization," presented at the Proceedings of the 23rd international conference on Supercomputing, Yorktown Heights, NY, USA, 2009.

[15] X. Zhang, et al., "XenSocket: a high-throughput interdomain transport for virtual machines," presented at the Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Newport Beach, California, 2007.

[16] S. G. Artis, et al., "Detection of atrial fibrillation using artificial neural networks," in Computers in Cardiology 1991, Proceedings., 1991, pp. 173-176.

[17] C. Philip de, et al., "Automatic classification of heartbeats using ECG morphology and heartbeat interval features," Biomedical Engineering, IEEE Transactions on, vol. 51, pp. 1196-1206, 2004.

[18] M. G. Tsipouras, et al., "An arrhythmia classification system based on the RR-interval signal," Artificial Intelligence in Medicine, vol. 33, pp. 237-250, 2005.

[19] A. L. Goldberger, et al., "PhysioBank, PhysioToolkit, and PhysioNet : Components of a New Research Resource for Complex Physiologic Signals," Circulation, vol. 101, pp. e215-e220, June 13, 2000 2000.

[20] B. Yegnanarayana, Artificial Neural Networks: Prentice-Hall of India, 2004.

[21] C. Cortes and V. Vapnik, "Support-vector networks," Machine Learning, vol. 20, pp. 273-297, 1995.