# HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms

Vishal Gupta
VMware
vishalg@vmware.com

Min Lee
Intel
min.lee@intel.com

Karsten Schwan
Georgia Institute of Technology
schwan@cc.gatech.edu

## Abstract

This paper presents HeteroVisor, a heterogeneity-aware hypervisor, that exploits resource heterogeneity to enhance the elasticity of cloud systems. Introducing the notion of 'elasticity' (E) states, HeteroVisor permits applications to manage their changes in resource requirements as state transitions that implicitly move their execution among heterogeneous platform components. Masking the details of platform heterogeneity from virtual machines, the E-state abstraction allows applications to adapt their resource usage in a fine-grained manner via VM-specific 'elasticity drivers' encoding VM-desired policies. The approach is explored for the heterogeneous processor and memory subsystems evolving for modern server platforms, leading to mechanisms that can manage these heterogeneous resources dynamically and as required by the different VMs being run. HeteroVisor is implemented for the Xen hypervisor, with mechanisms that go beyond core scaling to also deal with memory resources, via the online detection of hot memory pages and transparent page migration. Evaluation on an emulated heterogeneous platform uses workload traces from real-world data, demonstrating the ability to provide high on-demand performance while also reducing resource usage for these workloads.

***Categories and Subject Descriptors*** D.4.0 [*Operating Systems*]: General

***General Terms*** Design, Performance

***Keywords*** Heterogeneous platforms, Cloud elasticity

## 1. Introduction

Elasticity in cloud infrastructures enables 'on-demand' scaling of the resources used by an application. Resource scaling techniques used by modern cloud platforms like Amazon's Elastic Compute Cloud (EC2), involving the use of different types of virtual machines (VMs), however, are coarse-grained, both in space and in time. This has substantial monetary implications for customers, due to the costs incurred for limited sets of fixed types of VM instances and the frequencies at which heavy-weight scaling operations can be performed. Customers could implement their VM-internal solutions to this problem, but a *truly elastic* execution environment should provide 'fine-grained' scaling capabilities able to *frequently* adjust the resource allocations of applications in an *incremental* manner. Given the competition among cloud providers for better services, fine-grained resource management may prove to be a compelling feature of future cloud platforms [1, 13].

An emerging trend shaping future systems is the presence of heterogeneity in server platforms, including their processors, memories, and storage. Processors may differ in the levels of performance offered [12, 24], like the big/little cores commonly found in today's client systems. Memory heterogeneity can arise from the combined use of high speed 3D die-stacked memory, slower off-chip DRAM, and non-volatile memory [11, 30, 40, 44]. Such heterogeneity challenges system management, but we view it as an opportunity to improve future systems' scaling capabilities, by making it possible for execution contexts to move among heterogeneous components via dynamic 'spill' operations, in a fine-grained manner and driven by application needs.

The *HeteroVisor* virtual machine monitor presented in this paper hides the underlying complexity associated with platform heterogeneity from applications, yet provides them with a highly elastic execution environment. Guest VMs see what appears to be a homogeneous, yet scalable, virtual resource, which the hypervisor maintainsby appropriately mapping the virtual resource to underlying heterogeneous platform components. Specifically,HeteroVisor presents to guests the abstraction of *elasticity (E) states*, which provides them with a channel for dynamically expressing their resource requirements, without having to understand in detail the heterogeneity present in underlying hardware. Inspired by the already existing P-state interface [38] usedto

*VEE '15*, March 14–15, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-3450-1/15/03. . . $15.00.
http://dx.doi.org/10.1145/2731186.2731191

scale the frequency and voltage of processors, E-states generalizes that concept to address with one unified abstraction the multiple types of resource heterogeneity seen in future servers, including their processors and memory. As with P-state changes, E-state transitions triggered by applications provide hints to the hypervisor on managing the resources assigned to each VM, but for E-state changes, guests can further refine those hints via system- or application-level modules called *elasticity drivers* (like the Linux CPU governor in the case of P-states) to indicate preferences concerning such management. HeteroVisor uses them to better carry out the fine-grain adjustments needed by dynamic guests.

With heterogeneous CPUs, E-states are used to provide the abstraction of a scalable virtual CPU (vCPU) to applications desiring to operate at some requested elastic speed that may differ from that of any one of the actual heterogeneous physical cores. Such fine-grained speed adjustments are achieved by dynamically mapping the vCPUs in question to appropriate cores and in addition, imposing *usage caps* on vCPUs. For heterogeneous memories, E-states provide the abstraction of performance-scalable memory, with multiple performance levels obtained by adjusting guests' allocations of fast vs. slower memory resources.

E-states are challenging to implement. Concerning CPUs, fine-grained E-state adjustments seen by the hypervisor shouldbe honored in ways that efficiently use underlying cores, e.g., without unduly high levels of core switching.The issue is addressed by novel vCPU scaling methods in HeteroVisor. Concerning memory, previous work has shown that application performance is governed not by the total amount of fast vs. slow memory allocated to an application, but instead, by the fast vs. slow memory speeds experienced by an application's current memory footprint [27]. HeteroVisor addresses this by maintaining a page access-bit history for each VM, obtained by periodically scanning the access-bits available in page tables. This history is used to detect a guest's 'hot' memory pages, i.e., the current memory footprints of the running applications. Further, by mirroring guest page tables in the hypervisor, it can manipulate guest page mappings in a guest-transparent manner, thus making possible hot page migrations (between slower vs. faster memories), by simply changing mappings in these mirror page tables, without guest involvement. A final challenge is to decide which resources should be scaled to what extent, given the potential processor- vs. memory-intensive nature of an application. HeteroVisor'ssolution is to permit guest VMs to express their scaling preferences in per-VM 'elasticity drivers'.

HeteroVisor's implementation in the Xen hypervisor [4] is evaluated with realistic applications and workloads onactual hardware, not relying on architectural simulators. CPU and memory controller throttling are used to emulate processor and memory subsystem heterogeneity. For workloads derived from traces of Google cluster usage data [18], experimental results show that by exploiting heterogeneity in the unobtrusive ways advocated by our work, it becomes possible to achieve on-demand performance boosts as well as cost savings for guest applications with diverse resource requirements. The CPU and memory scaling mechanisms provide up to 2.3x improved quality-of-service (QoS), while also reducing CPU and memory resource usage by an average 21% and 30%, respectively. Elasticity drivers are shown useful via comparison of two different guest usage policies, resulting in different trade-offs between QoS and cost.

## 2. Elasticity via Heterogeneity

### 2.1 Elasticity in Clouds

Elasticity, i.e., the ability to scale resources on-demand to minimize cost, is an attractive feature of cloud computing systems. Resources can be scaled in a 'scale out' or 'scale up' manner. Table 1 shows a comparison summary of these two approaches. Scale-out varies the number of VM instances used by an application. It is used in commercial cloud services like Amazon EC2 AutoScaleto increase capacity in the form of additional VMs of fixed instance types, where instances can be rented in the order of several minutes to a full hour, and users are charged for the whole instance even if it is only partially used. Thus, scale out is a rather heavy-weight and coarse-grained operation with high end-user cost implications.

Table 1: Elastic resource scaling in clouds

| | **Scale out** | **Scale up** |
|---|---|---|
| Scaling Method | VM Instances | Resource Shares |
| Resource Granularity | Coarse | Fine |
| Time Granularity | Slow | Fast |
| Software Changes | High | Minimal |

'Scale up' operations entail adjusting the shares of platform resources to which a VM is entitled. Such fine-grained elasticity enables a user to start a VM with some basic configuration and dynamically alterthe platform configuration it needs. Such scaling may be sufficient for and in fact, preferable to VM-level scaling, e.g., when a VM experiences sudden short bursts requiring temporarily higher levels of resources. For current cloud users, the presence of such functionality would mean shorter rent durations (on the order of seconds)and reduced costs. Another advantage is that such scaling can be transparent to the VM, not requiring sophisticated software changes or VM-level management methods to deal with varying resource needs.

### 2.2 Resource Heterogeneity

HeteroVisor enhances the scaling capabilities of future cloud computing systems by exploiting the increasing levels of resource heterogeneity seen in server platforms. Evidence of such heterogeneity abounds. Heterogeneous processors, i.e., CPU cores that differ in their performance/power capabilities (as shown in Figure 1), are known to be energy-efficient alternatives to homogeneous configurations [12, 24, 48], underlined by commercial implementations from CPU vendors [16, 35]and encouraged by research demonstrating the utility of low-powered cores for datacenter applications [3, 20] and by methods that efficiently utilize brawny cores [5, 26]. Schedulers for heterogeneous cores have seen extensive exploration [23, 24, 39, 41, 45].

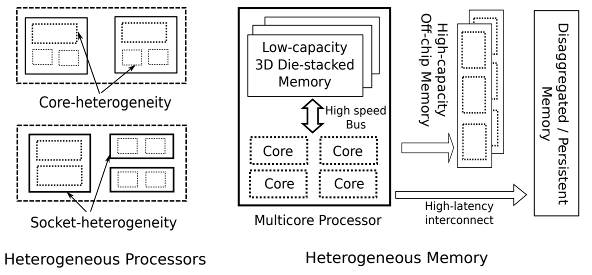


Figure 1: Platforms with heterogeneous resources

Heterogeneity in memory technologies has gone beyond the NUMA properties seen in high-end server platforms.New memory technologies like die-stacked 3D memories and non-volatile memories, in addition to traditional DRAM, can result in a hierarchy of heterogeneous memory organization, as shown in Figure 1. 3D stacked memories can provide lower latency and higher bandwidth, in comparison to traditional off-chip memories [29]. But since the capacity of such memories is limited [30], future servers expect to have a combination of both fast on-chip memory and additional slower off-chip memory. Moreover, inclusion of disaggregated memory or persistent memory technologies will further extend memory heterogeneity [11, 28, 40, 44].

Heterogeneity is already present in storage subsystems when using local vs. remote storage, SSDs vs. hard drives, andfuture persistent memory. HeteroVisor is concerned with heterogeneity in platforms' CPU and memory subsystems, but its general approach is applicable to other resources, as well.

## 2.3 Exploiting Heterogeneity

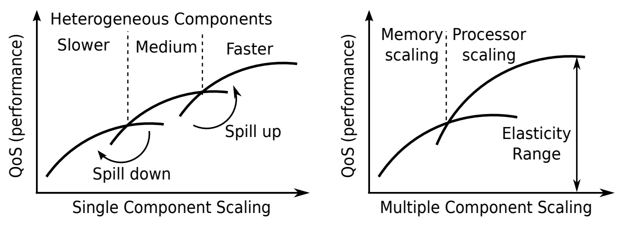


Figure 2: Using heterogeneity to enable resource scaling

HeteroVisor enhances a server's elastic resource scaling capabilities with an approach in which 'spill' operations change the heterogeneity of VMs' resource allocations. Consider a resource like memory with heterogeneous components with three different performance characteristics, i.e., die-stacked DRAM as the fast resource, off-chip DRAM as the medium-performance resource, and non-volatile memory as the slow resource. With each of these components supporting a different performance range, the performance of the memory subsystem seen by each guest VM can be adjusted across a wide range, by varying the allocation mix given to the application. As a higher share of a VM's resources are allocated in faster memory (e.g., by moving application data to on-chip memory from off-chip DRAM), its performance increases. This is denoted as a 'spill up' operation, as shown in Figure 2. Similarly, by 'spilling down' the application resource (e.g., ballooning out VM pages to persistent memory), performance can be lowered, perhaps in response to an application-level decrease in the memory intensity of its activities. In this manner, the hypervisor provides to guest VMs the abstraction of scalable memory, internally using spill operations over the underlying heterogeneous components. Further, by appropriately allocating various amounts of slower vs. faster memory to applications, memory scaling can extend beyond the three different physical speeds present in physical hardware (i.e., 3D die stacked RAM, off-chip DRAM, NVRAM) to offer what appear to be finer grain scaled memory speeds andbandwidths.

The scaling mechanisms outlined for memory above can be applied to other platform resources, including processor andstorage components, to provide an overall extended elasticity range to an application, as shown in Figure 2. Furthermore, this elasticity extends across multiple resource types,so that HeteroVisor can offer guest VMs slow processors with rapidly accessible memory for data-intensiveapplications, while a CPU-intensive guest with good cache behavior may be well-served with slower memory components. When doing so, the different components' use is governed by spill operations: (i) processor scaling is achieved by appropriate scheduling of vCPUs to heterogeneous cores and capping their usage of these cores to achieve a target speed, and (ii) memory spill operations manage memory usage. Note that considerable complexity for the latter arises from the facts that page migrations may incur large overheads and more importantly, because the hypervisor does not have direct visibility into a VM's memory access pattern (to the different memory allocated to it) determining its performance. HeteroVisor addresses this issue by developing efficient mechanismsto detect a guest VM's 'hot' pages, i.e., frequently accessed pages, and then moving those between different memories without guest involvement. The next section describes the various mechanisms incorporated into HeteroVisor to implement resource scaling.

## 3. Design

Using heterogeneous platform resources, HeteroVisor provides fine-grained elasticity for cloud platforms. To incorporate heterogeneity into the scaling methods, there are several principles that we follow in our design.

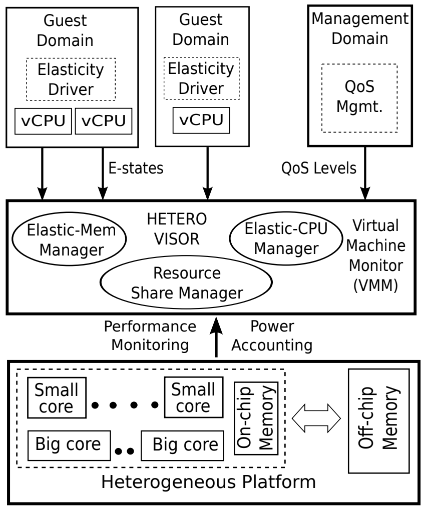


Figure 3: System architecture for HeteroVisor

- Adhering to the philosophy that cloud platforms should sell resources and not performance, VMs should explicitly request resources from the cloud provider. This design requiring application VMs to specify their resource requirements is common to IaaS platforms where users select different types of VM instances.
- Typically special software support is required for managing heterogeneity. Diversity across vendors and rapidly changing hardware make it difficult for operating systems to incorporate explicit mechanisms for managing these components. Thus, the complexity of managing heterogeneous components should be hidden from the users.
- The resource scaling interface should be generic and extensible to allow its use on various platforms with different heterogeneous configurations. It should allow scaling of resources in incremental ways and should be lightweight in nature for frequent reconfiguration. It should also work with multiple resources.

Figure 3 depicts HeteroVisor, its various components, and their interactions. The underlying server platform consists of heterogeneous CPUs and memory, and it provides capabilities for online performance and power monitoring. The platform is shared by multiple guest virtual machines, where each VM communicates with the hypervisor about its resource requirements through the *elasticity (E) state* interface (detailed in Section 3.1). E-states are controlled by an *E-state driver* module, allowing the guest VM to communicate its changing resource needs and usage as state transitions. The hypervisor contains heterogeneity-aware elastic resource managers including a CPU scheduler and, memory manager. It also contains a resource share manager which is the higher-level resource allocator that takes into account various E-state inputs from the VMs and QoS related policy constraints from the management domain, to partition resources across all VMs, whereas the CPU and memory managers enforce these partitions and manage them efficiently for each VM. These components are described in more detail next.

## 3.1 Elasticity States

Inspired by the P-state (performance-state) interface [38] defined by the ACPI standard and used to control CPU voltage and frequency (DVFS), the E-state (elasticity-state) abstraction permits VMs to provide hints to the hypervisor, governed by VM-specific E-state drivers.

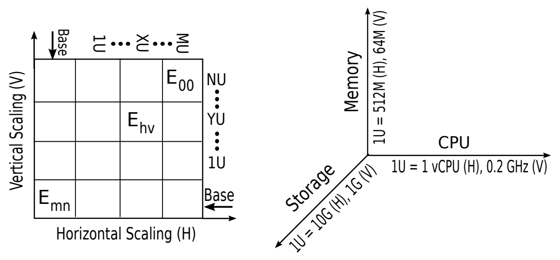


Figure 4: Elasticity state abstraction for resource scaling

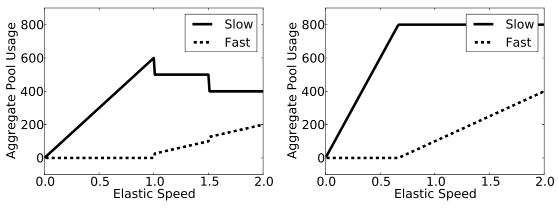
The E-state interface defines multiple states, where each state corresponds to a different resource configuration. E-states are arranged along two dimensions, corresponding to horizontal and vertical scaling as shown in Figure 4. Horizontal scaling makes it possible to add virtual resources to the application, using hot-plug based mechanisms; vertical scaling implies boosting the performance of existing platform resources. Both horizontal and vertical scaling are scale up methods, separate from the scale out methods varying the number VM instances. As in the case of P-states, a higher numbered E-state ($E_{mn}$) represents a lower resource configuration, while a lower numbered E-state ($E_{00}$) implies a higher performance mode. Further, these states are specific to each scalable component, resulting in separate state specifications for processor, memory, and storage subsystems. For all resource types, however, a change in E-state implies a request to change the allocation of resources to that VM by a certain number of resource units (U). For the CPU component, a horizontal E-state operation changes the number of vCPUs, while vertical scaling adjusts vCPU speed in units of CPU frequency. Similarly, for the memory subsystem, horizontal scaling is achieved by changing its overall memory allocation, while vertical scaling adjusts its current allocation in terms of usage of fast/slow memory (at page granularity). HeteroVisor's current policies are concerned with vertical scaling in the presence of heterogeneous resources, explainedin more detail next.

## 3.2 Elastic CPU Manager

Heterogeneous resources consisting of components with different performance levels can be used to provide a virtual, highly elastic server system. We next describe how this can be achieved for heterogeneous cores,with a formulation specialized for the case of two different types of cores (this can be generalized to multiple performance levels). Section 3.3 extends the approach to heterogeneous memory systems.

### 3.2.1 Virtual Core Scaling:

Given a platform configuration with heterogeneous cores, the objective of the elastic CPU manager is to provide to a guest VMhomogeneous virtual cores running at some desired speed that may be different from the speeds of the physical cores being used. This can be achieved by appropriate scheduling of the vCPUs on these heterogeneous cores and assigning a *usage cap* to each vCPU, limiting its usage of physical resources. For such scaling, our current approach schedules all vCPUs on slow cores initially, with fast cores kept idle, the assumption being that slow cores have lower ownership costs for the cloud usersrequesting resources for their VMs. As vCPUs are scaled up, the slow core cap of vCPUs is increased to meet the desired scaling speed.When slow core cycles are saturated, further scaling results in vCPUs being scheduled to fast cores, providing higher scaled speeds than what is possible with slow cores only.



(a) 6 vCPU VM (b) 12 vCPU VM

Figure 5: Models for vCPU scaling using heterogeneity

The expressions for the corresponding usage caps of various cores for achieving a given effective processing speed can be obtained by formulating a linear optimization problem, solvable using standard solvers. Since allocations must be computed in kernel-space, instead of relying on external solvers, we obtain a closed-form solution for the special case of two types of cores, slow (s) and fast (f), where slow cores have lower ownership cost than fast cores, thereby prioritizing allocations to use slow cores before using fast cores. We omit the formulation and derivation of these expressions due to space constraints. Instead, Figure 5 plots the resultant equations for a configuration with 8 slow cores with 1x speed and 4 fast cores with 4x speed. The figure shows the aggregate slow and fast pool usage for a VM (total percentage utilization caps are assigned collectively to all vCPUs) as we vary the elastic core speed. Two different VM configurations are plotted, by varying the number of vCPUs in the VM to 6 and 12.

In both the cases, slow pool usage first increases linearly as we increase the elastic core speed (solid lines). Once slow cores are saturated at usage values 600 for 6 vCPUs and 800 for 12 vCPUs (constrained by 8 physical slow cores), fast pool usage gradually increases (see the dotted lines) to obtain the requested elastic scaling. For example, a VM with 12 vCPUs at speed 1U exhibits 800% slow pool utilization (8 slow cores fully utilized) and 100% fast pool usage (1 fast core with speed 4x). We also see jumps in the CPU usage with $v_n$ equal to 6 at speed 1 and 1.5, which happens due to the shift of a slow pool vCPU to the fast pool.

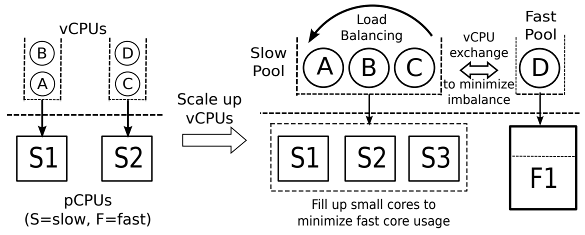


Figure 6: Virtual core scaling using heterogeneous cores

For elastic scaling, vCPUs are partitioned into two pools, one corresponding to the each type of core, i.e., slow and fast pool. Each vCPU executes within a particular pool and is load-balanced among other vCPUs belonging to that pool, as shown in Figure 6. Because of this partitioning of vCPUs into pools, there may arise performance imbalances among vCPUs. To deal with this, a rotation is performed periodically among pools to exchange some vCPU, thus giving every vCPU a chance to run on the fast cores, resulting in better balanced performance. Such migrations have very little cost if done infrequently and particularly if the cores involved share a last-level cache.

### 3.2.2 Implementation:

Virtual core scaling is implemented by augmenting Xen's CPU credit scheduler by adding two different types of credits: slow and fast. Credits represent the resource right of a VM to execute on the respective types of cores and are distributed periodically (30ms) to each running VM. A vCPU owns one type of credits during one accounting period. As the VM executes, its credits are decremented periodically (every 30ms) based upon the type of cores it uses. A vCPU can execute as far as it has positive credits available. Once it has consumed all credits, it goes offline by being placed into a separate 'parking queue' until the next allocation period. At this point, the credits are redistributed to each VM, and its vCPUs are again made available for scheduling. Further, a circular queue of vCPUs is maintained to periodically rotate vCPUs between slow and fast cores.We find it sufficient to use a granularity of 10 scheduler ticks, i.e., at a frequency of 300ms, for this purpose,for the long-running server workloads used in our experimental evaluation.

## 3.3 Elastic Memory Manager

HeteroVisor realizes performance-scalable memory by changing a VM's memory allocation across underlying heterogeneous components, i.e., use of fast memory for high-performance E-states and slow memory for slower E-states. This section describes elasticity management for heterogeneous memories involving fast die-stacked memory and slow off-chip DRAMs. Since die-stacked memoryis small in capacity in comparison to off-chip DRAM, a subset of pages from the application's memory must be chosen to be placed into stacked-DRAM. For this purpose, it is important to detect and manage the application's 'hot' pages that are critical to its performance. This requires the hypervisor to efficiently track each guest's memory accesses.

### 3.3.1 Memory Access Tracking:

Modern processors provide only limited hardware support for detecting application' memory access patterns. On the x86 architecture, each page table entry has an access bit, which is set by the hardware when the corresponding page is accessed. Software is responsible for clearing/using this bit. We use this single-bit information to build an access bit history to determine a VM's memory access pattern. Specifically, we periodically scan and collect the access bits, forming a bitmap, called an 'A-bit history' (access-bit history), shown in Figure 7. A 32-bit word and a 100ms time interval is used for scanning, implying 3.2 seconds of virtual time corresponding to one word. If the A-bit history has many ones, i.e., it is a dense A-bit history, this indicates that the page is hot and frequently accessed. A threshold of 22, obtained experimentally, is used in our work for marking a page as hot.

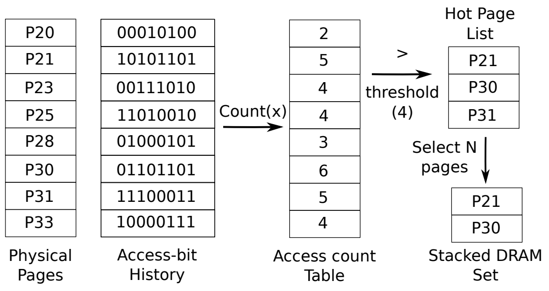


Figure 7: Tracking hot pages using access-bits

Since application processes within a guest VM run in virtual time, page tables are scanned over virtual time – every 100ms –rather than wall-clock time, for an accurate A-bit history.For accurate accounting, events like timer tick (TIMER), address space switches (NEW_CR3), and vCPU switches (SCHEDULE) are also taken into account.Our implementation collects and maintains an A-bit history for all machine frames for all guest VMs, including the management domain.

### 3.3.2 Hot Page Management:

Detected with the A-bit history, hot pages are actively managed by moving them in and out of fast/slow memories. There are four categories of pages and associated actions, depending on their residencies and status, as shown in Table 2. Active hot pages should be migrated to or maintained in fast memory, and inactive cold pages should be discarded. While Cases 2 & 3 are relatively easy tasks, Actions 1 and 4 are the primary determinants of the overhead of page migrations, handled as described below.

Table 2: Hot page management actions

| | Residency | Status | Action |
|---|---|---|---|
| 1 | Off-Chip | Active | Migrate to on-chip DRAM |
| 2 | Off-Chip | Inactive | Drop from the list |
| 3 | On-Chip | Active | Keep in on-chip DRAM |
| 4 | On-Chip | Inactive | Migrate to off-chip DRAM |

Hot pages are managed to form a linked list (see Figure 8). Since this list can be quite long, its inspectioncan cause substantial overheads for scanning and migrating such pages. To efficiently manage this list, only parts of the list are considered at one time, where MAX_SCAN (currently 512) determines the number of pages that are scanned in a time window (every 100ms). Further, the removal of inactive pages may incur page migrations to off-chip memory, causing potentialperturbation seen by co-running applications. In addition, since pages freed by the guest are likely to be used again by the next job(since memory allocators in guest VMs often reuses previously freed pages), it is beneficial to employ 'lazy' page migrations, that is, to delay the eviction of selected pages from stacked DRAM. We do so by migrating only MAX_MFNS pages from the hot page list every time the list is scanned. Finally, TIME_WINDOW macro (3000ms) defines when a page in the list becomes inactive. Thus, if a page in the list is not accessed for 3000ms, it is considered inactive and eventually discarded.

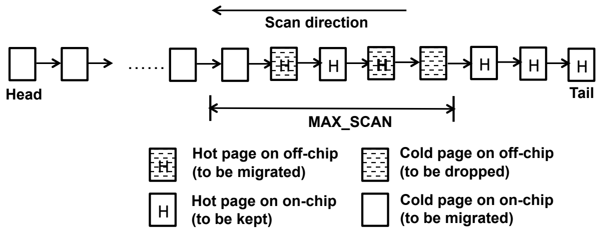

Scan direction
Head
Tail
MAX_SCAN
Hot page on off-chip (to be migrated)
Cold page on off-chip (to be dropped)
Hot page on on-chip (to be kept)
Cold page on on-chip (to be migrated)

Figure 8: Hot page management and associated actions

A final note concerning hot page list management is that scanning happens in the reverse direction, as new pages are added to the front of the list, and the tail typically contains the oldest pages. This further reduces overhead, since it avoids unnecessary page migrations.

### 3.3.3 Transparent Page Migration:

Memory spill operations are performed by migrating pages between different memories. Such migrations require remapping guest page tables, which are hidden from the hypervisor. In order to do this in a guest-transparent way, Hetero-Visor*mirrors* guest page tables.For para-virtualized guests, page tables are write-protected and require the hypervisor's involvement in updating page table entries through a hypercall. We simply intercept these calls and re-create a mirror version of the page tables as shown in Figure 9 and install them in the CR3 hardware register, forcing the guest to use these mirrors. This allows us to freely change virtual-to-physical mappings, without any changes to the guest OS. For fully virtualized guests, these mechanisms may be simplified by hardware supports such as NPT (nested page table) or EPT (Extended page table). These architectural extensions implement an extra level of address translation so this extra layer can be used to implement transparent page migrations.

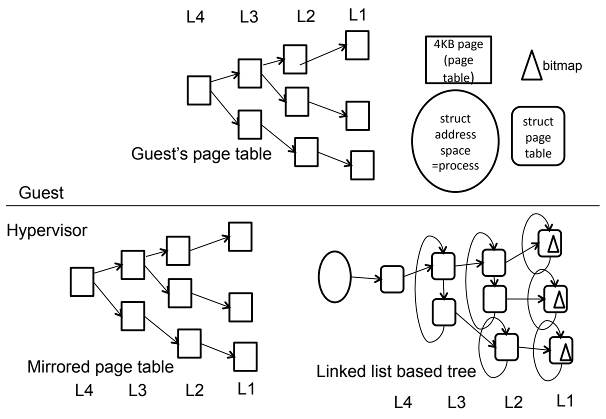


Figure 9: Mirror page tables for page migrations

Additional optimizations serve to minimize negative impacts on cache behavior. Rather than scanning page tables completely, separate metadata structures using linked lists and pointers (see Figure 9) are used for easy iteration over page table entries, optimizing page table scanning for access bits. Without this optimization, the whole 4KB of each page table would be scanned, thus trashing cache real estate (i.e., 4KB-worth cache lines). This is particularly important for interior (L2, L3, L4) page tables. Further, only existing mappings are managed in this list, thereby effectively eliminatingunnecessary scans. Finally, L1 page tables (leaf node) use a bitmap to quickly detect present pages. This again eliminates unnecessary scans on the L1 page table and prevents valuable cache lines from being evicted.

### 3.3.4 Handling Shared Pages:

Since any machine frame can be shared between multiple processes/guests, all of the corresponding page table entries must be updated when migrating such a page. To do this efficiently, we employ *reverse maps* (Rmaps) that store this reverse mapping information,i.e., from a page in physical memory to entries in various page tables. We can iterate over this Rmap list to find all of the mappings to a given page, thus enabling efficient remapping for any given page. Each machine page (mfn) is associated with one Rmap_list that contains pointers to page table and page table index.

## 3.4 Elasticity Driver

Elasticity drivers are the guest-specific components of the HeteroVisor stack, allowing guest VMs to guide resource allocation by triggering E-state transitions, in a manner similar to the CPU governor making P-state (performance-state) changes in the context of DVFS (dynamic voltage and frequency scaling) [38]. Interesting resource management policies can be implemented by using different implementations of the driver, thus permitting each application (i.e., guest VM orsome set of guest VMs – a VM ensemble) to choose a specific driver catering to its requirements. Various solutions (e.g., RightScale) are already available to implement resource scaling controllers for applications,and by making it easy to employ such solutions, the E-state driver is a step forward in giving applications fine-grained controlover how their resources are managed. HeteroVisor does not require guests to specify E-state drivers, of course, thus also able to support traditional VMs with static configurations, but such VMs will likely experience cost/performance penalties due to over/under-provisioning oftheir resources.

We note that it should be clear from these descriptions that guest VMs can use custom and potentially, quite sophisticated controllers in their E-state drivers, including prediction-based mechanisms [42] that model application behavior to determine the resultant E-state changes. The E-state driver used in our current work implements a simple reactive heuristic for illustration. The driver performs the scaling in two steps:

**Step 1:** First pick a resource for scaling (CPU, memory) for the current epoch

**Step 2:** Request scaling operation (up, down, or no change) for the selected resource

To select a resource for scaling, it needs to consider two factors in making this decision: the application's sensitivity to the resource and the cost of the resource to obtain best performance for minimum cost. The current driver uses IPC (instructions-per-cycle) as the metric to determine an application's sensitivity to CPU or memory. If IPC is high, scaling is performed along the CPU axis for that epoch; otherwise, it picks the memory resource for scaling. It currently assigns equal cost to both types of resources (CPU and memory), but any cost values can be incorporated in the heuristic to obtain the desired cost/performance trade-off.

The scaling heuristic used employs a combination of utility factor (*util*) and application performance (*qos*) to form the E-state transition logic shown in Algorithm 1. The utility factor is analogous to CPU/memory utilization, i.e., the percentage of resources (CPU usage cap or memory pages) consumed by a VM against its assigned usage. Similarly, the QoS metrics, such as response time or response rate, can be obtained from the application. Specifically, for these metrics, the E-state driver defines four thresholds: $qos_{hi}$, $qos_{lo}$, $util_{hi}$, and $util_{lo}$. If $qos$ is lower than the minimum required performance $qos_{lo}$ or if the utility factor is higher than $util_{hi}$ mark, an E-state scaleup operation is requested. Scale down logic requires $qos$ to be higher than $qos_{hi}$ and $util$ to be lower than the $util_{lo}$ threshold.

Intuitively, if the application performance is lower than its SLA or if the utility factor is too high, which may cause SLA violations, a scale up operation is issued to request more resources. On the other hand, if application performance is higher than its desired SLA, a scale down operation can be issued, given that the utility factor is low to avoid violations after scaling. In order to avoid oscillations due to transitory application behavior, history counters are used to dampen switching frequency. Specifically, a switch is requested only after a fixed number of consecutive identical E-state change requests are received. The history counter is a simple integer counter, which is incremented whenever consecutive intervals generate the same requests and reset otherwise.

**Algorithm 1:** Elasticity-Driver Scaling Heuristic

```
if $util > util_{hi}$ OR $qos < qos_{lo}$ then
    $E_{next} \leftarrow E_{cur-1}$ ;                // Scale up
else if $util < util_{lo}$ AND $qos > qos_{hi}$ then
    $E_{next} \leftarrow E_{cur+1}$ ;                // Scale down
else
    $E_{next} \leftarrow E_{cur}$ ;                  // No change
```

The E-state driver is implemented as a Linux kernel module that periodically changes E-states by issuing a hypercall to Xen. The driver uses a QoS interface in the form of a proc file to which the application periodically writes its QoS metric. In addition, it reads the VM utility factor from the hypervisor through a hypercall interface. The E-state driver runs once every second, with a value of three for the history counter and 1.25 as the IPC cut-off for resource selection.

### 3.5 Resource Share Manager

HeteroVisor uses a tiered-service model where different clients can receive different levels of quality of service. Clients requesting higher QoS and thus paying higher cost, obtain better guaranties in terms of resource allocation, by prioritizing allocations to their VMs. In comparison, clients with lower QoS requirements obtain resources as they become available. Such clients with different QoS levels can be co-run, to minimize resource waste while maintaining QoS. An example of such a service is Amazon EC2 Spot instances which run along with standard EC2 instances and receive resources when unused by standard instances. In such scenarios, each VM is assigned a QoS-level (similar to Q-states [34]) by the administrator that states its willingness to pay. The allocation is then performed for each resource to determine the share of each VM, taking into account any E-state and Q-state changes, as shown in Algorithm 2. The allocation happens periodically and proceeds in multiple phases, as described below.

**Phase 1:** to ensure fairness in allocation, a minimum share value is assigned to each application (if given), so that no high QoS application can starve lower QoS applications. **Phase 2:** the allocation is done in sorted order of VM Q-states. For each VM, it increases or decreases its resource allocation by a fixed fraction ($\delta$) depending on the desired E-state change by that VM as 'up' or 'down', respectively. Otherwise, the allocation is kept constant. **Phase 3:** After allocating shares to all the applications, any remaining resource shares are assigned to low QoS instances which are willing to accept as many resource as available.

**Algorithm 2:** Resource Share Allocation Algorithm

```
Input: $estate(vm_i), qstate(vm_i)$
Output: $share(vm_i)$
$share_{avail} = share_{total}$
foreach VM $vm_i$ (in order $qstate(vm_i)$ high $\rightarrow$ low) do
    if $share_{avail} > 0$ then
        if $estate(vm_i) == estate_{up}$ then
            // Assign more shares
            $share(vm_i) = share(vm_i) + \delta$
        end
        else if $estate(vm_i) == estate_{down}$ then
            // Reduce resource shares
            $share(vm_i) = share(vm_i) - \delta$
        end
        $share_{avail} = share_{avail} - share(vm_i)$
    end
end
```

The paper's current experimental evaluation considers only single-VM scenarios. The allocation problem across competing VMs can be solved using various statistical techniques including bidding mechanisms and priority management [2].We do not experiment with such methods because this paper's focus is on ways to manage heterogeneity in cloud environmentsrather than on allocation and scheduling techniques.

## 4. Experimental Evaluation

### 4.1 Setup

Our experimental platform consists of a dual-socket 12 core Intel Westmere server with 12GB DDR3 memory, withheterogeneity emulated as follows. Processor heterogeneity is emulated using CPU throttling, by writing to CPU MSRs, which allows changing the duty cycle of each core independently. Memory throttling is used to emulate heterogeneous memory. It is performed by writing to the PCI registers of the memory controller, thus slowing it down. This allows us to experiment with memory configurations of various speeds, such as M1 and M2, which are approximately 2x and 5x slower than the original M0 configuration with no throttling.

In all experiments, response time is chosen as the QoS metric (lower is better), implying that an inverse value of latency is used in the QoS thresholds for the driver. A latency value of 10ms is chosen as the SLA, corresponding to which two policies are evaluated by using different thresholds for the scaling algorithm as shown in Table 3. These thresholds are obtained after experimenting with several different values. The first QoS-driven policy (ES-Q) is performance-sensitive, while the second resource-driven policy (ES-R) favorslower speeds, and thus, higher resource savings.

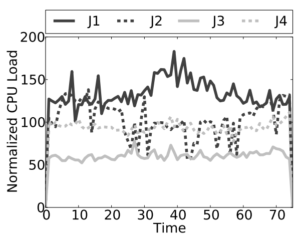
Table 3: Thresholds for scaling policies

| | $qos_{hi}$ | $util_{lo}$ | $qos_{lo}$ | $util_{hi}$ |
|---|---|---|---|---|
| ES-Q | 1/5 | 40 | 1/10 | 90 |
| ES-R | 1/5 | 50 | 1/15 | 95 |

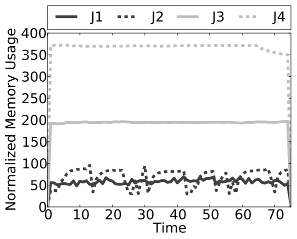
For CPU experiments, a platform configuration consisting of eight slow cores and four fast cores is considered, where slow and fast cores are distributed uniformly on each socket to minimize migration overheads. The performance ratio between fast and slow cores is kept at 4x. Experiments are conducted using a VM with 12 vCPUs, providing an elasticity range up to 2U. Having an E-state step of 0.2U gives us 10 CPU E-states from E0 (2U) to E9 (0.2U), which are exported by the E-state interface. Similarly, memory evaluation is done using a platform configuration with 512MB of fast memory and an E-state step of 64MB, resulting in 8 memory E-states. Elastic scaling mechanisms are compared against a base case configuration with a static allocation of 1U CPU resources (E5 CPU state) and 256MB stacked memory (E4 memory state).

## 4.2 Workloads

Experimental runs use a web server and an in-house memcached-like (memstore) application, which service a stream of incoming requests from a client machine. The web server launches a CPU-intensive computation kernel, while memstore performs a memory lookup operation in response to each request. The memstore application allows us to load the memory subsystem to its peak capacity, avoiding CPU and network bottlenecks associated with standard memcached implementation. In addition, several other benchmarks, including SPEC CPU2006 and modern datacenter applications, are also included in the analysis.



(a) CPU



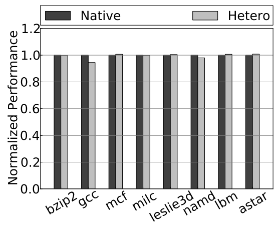
(b) Memory

Figure 10: Traces based on Google cluster data [18]

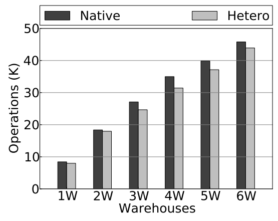
In order to simulate dynamically varying resource usage behavior, workload profiles based on data from Google cluster traces are used [18]. Specifically, this data provides the normalized CPU utilization of a set of jobs over several hours from one of Google's production clusters. The dataset consists of four types of jobs from which we obtain the average CPU load and memory usage of each type of job, with resultant data shown in Figure 10. As seen from the figure, workload J1 has constant high CPU usage, while J2 has varying behavior, with phases of high and low usage. In comparison, workloads J3 and J4 have uniform CPU usage, with J3 having significant idle components. Similarly, the memory usage behavior of these jobs shows J1 and J2 having low memory footprints, while J3 and J4 have higher usage profiles. These traces are replayed by varying the input request rate in proportion to the CPU load and changing the data-store size of the memstore workload in proportion to the memory usage of each trace, with each data point maintained for 20 seconds. It is to be noted that the data presented in the graphs is averaged across the entire cluster rather than being retrieved from a single server instance, because the dataset does not provide the machine mapping. We believe, however, that these jobs offer a good mix to test different dynamic workload scenarios present in server systems.

## 4.3 Experimental Results

Evaluations analyze the gains in performance and resource cost attainable from using fine-grained elastic scaling, compared to static allocation schemes.



(a) SPECCPU



(b) SPECJBB

Figure 11: Performance comparison of heterogeneous configurations with the native platform

Figure 11 evaluates the overheads associated with scaling operations. Specifically, Figure 11a compares the performance of several SPEC CPU2006 benchmarks with composed virtual platforms using heterogeneous cores (8S+4F) against standard homogeneous configurations (12S). Both configurations operate at an elastic core speed of 1U and memory is allocated completely from off-chip DRAM. The data shows comparable performance for both the configurations, implying that the overhead associated with its scaling operations are small. In order to evaluate multi-threaded execution scenarios, Figure 11b shows the performance score for SPECjbb2005, a Java multi-tier warehouse benchmark, at different configurations, by increasing the number of warehouses. As seen from the figure, performance results for the both cases closely follow each other with increasing threads, showing its applicability to multi-threaded applications as well.

E-state scaling is first evaluated by running the web server application with increasing load and withdynamic scaling of

E-states (see Figure 12). Figures 12a and 12b show the response rate and response time for this workload. As apparent in the figures, throughput rises gradually as load is increased. The corresponding latency curve is relatively flat, as the E-state driver scales E-states to maintain latency within the SLA (10ms). We also notice a few spikes in the latency graph; these occurin response to an increase in the input load, whereupon the E-state is scaled up to reduce latency. The corresponding E-state graph is shown in Figure 12c, where E-states are scaled from from E9 to E4 in multiple steps.

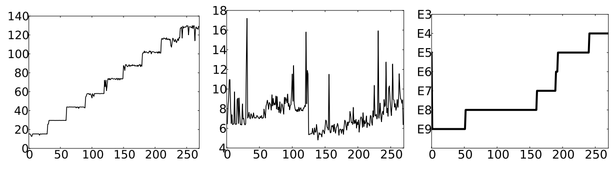

(a) Response rate (b) Resp. time (ms) (c) E-state

Figure 12: Elastic scaling experiment using the webserver workload (x-axis = time (s))

Evaluating the impact of memory heterogeneity, Figure 13 compares the performance of several SPEC CPU2006 applications (see Figure 13a) and various modern cloud application benchmarks, including graph database, graph search, key-value store, Lucene search engine, Tomcat server, kmeans, page-rank, and streamcluster algorithms (see Figure 13b) on different memory configurations. Specifically, it shows normalized performance at the base M0 configuration (without throttling) and for the M1 and M2 memory configurations (by applying different amounts of memory controller throttling). As evident from the figure, several applications experience severe performance degradation due to low memory performance, including 14x (5x) and 7x (4x) performance loss for the mcf and kvstore (key-value store) applications for the two memory configurations: M2 and M1. Other applications, like bzip and page-rank, exhibit less sensitivity. Overall, these results suggest that memory performance is critical to realistic applications which can therefore, benefit from the elastic management of heterogeneous memory resources.

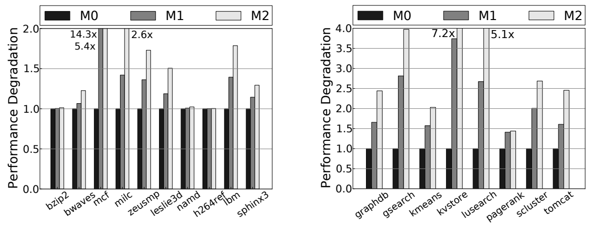

(a) SPECCPU (b) DATACENTER

Figure 13: Impact of memory performance

Showing the use of the A-bit history based mechanisms to obtain the working set sizes (WSS) of applications, Figure 14 plots WSS graphs as a function of time for several SPEC CPU2006 workloads. As seen in the figure, working set size varies across applications from ~10MB for omnetpp to a much larger value of ~200MB for memory-intensive mcf. Further, WSS dynamically changes over time for these applications, thereby showing the need for runtime memory elasticity.

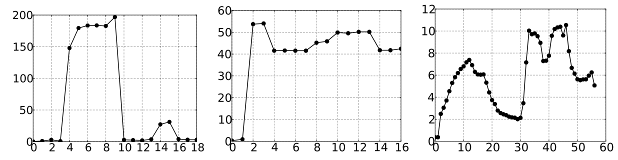

(a) mcf (b) milc (c) omnetpp

Figure 14: Working set size detection using access-bit history (x-axis = time (s), y-axis = WSS (MB))

Figure 15 shows the performance impact of memory E-state scaling on the memstore application, by gradually scaling E-states from E7 to E0, i.e., increasing the size of the fast memory allocation, where each state is maintained for five seconds before scaling to the next state. The non-scaling scenario (NS) shows a flat latency graph at 34ms and 42ms for the M1 and M2 configurations, respectively. In comparison, when E-states are scaled up from E7 (left) to E0 (right) in Figure 15a, the average latency for each memory operation decreases gradually to 8ms. The reduced access times with elastic scaling causes a 4.3x increase in application throughput (from 0.28M to 1.2M) (see Figure 15b). Also, the performance of the NS and ES configurations are comparable when no fast memory is used, signifying negligible overheads due to management operations like page table scans, mirroring, and maintaining other data structures. These results demonstrate that resource scaling on heterogeneous memory systems can be applied to obtain desired QoS for memory-sensitive applications.

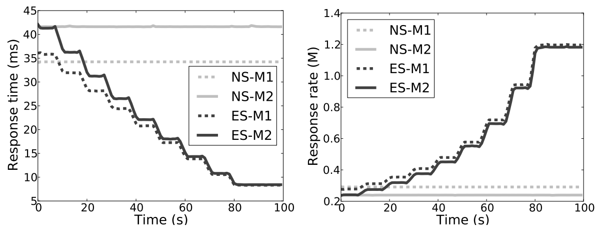
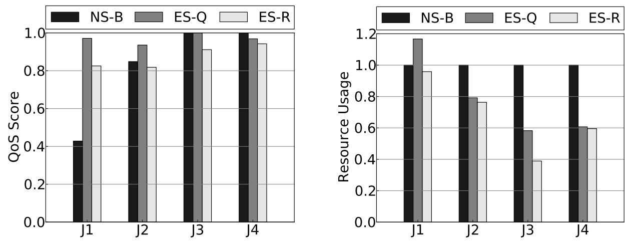

(a) Latency (b) Throughput

Figure 15: Impact of elastic memory scaling on the performance of memstore application

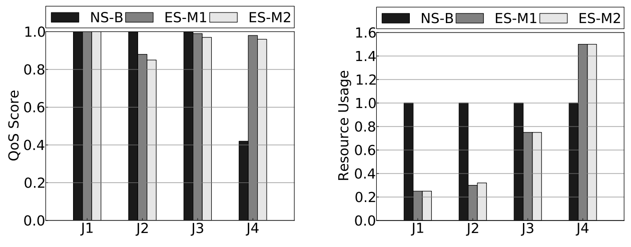
We next evaluate the four workloads based on Google cluster traces shown in Figure 10. The results in Figure 16 compare the QoS and resource usage for the base configuration without any elastic scaling (NS-B) with elastic CPU scaling for the two policies ES-Q and ES-R given in Table 3. The base platform configuration consists of 12 slow cores, each with an elastic speed of 1U. The QoS score graph shows the fraction of queries for which service latency falls within the SLA (10ms). Similarly, the resource usage graphs compare the relative usage of various configurations, assuming a linear relationship between E-states and resource usage.



(a) QoS Score (b) Resource Usage

Figure 16: Experimental results for CPU E-state scaling

As the results show, both policies provide much higher QoS than the base system for workload J1. Specifically, the QoS-sensitive policy ES-Q results in a 97% QoS score, with a 17% resource usage penalty, while the resource-driven policy ES-R provides lower QoS (83%), with lower usage (0.96x). In comparison, the base platform can only sustain a 43% QoS level. It is clear, therefore, that HeteroVisor can scale up resources to provide better performance when system load is high. For workload J2, ES-Q exhibits 9% higher and ES-R results in 3% lower QoS, while also reducing resource usage by 21% and 24%, respectively. Thus, resources are scaled up and down to meet the desired performance requirement. For J3 with low input load, HeteroVisor yields resource savings while also maintaining QoS, i.e., it generates 100% and 91% QoS scores with 42% and 61% lower resource usage for the two policies. In this manner, scaling down resources during low load periods produces savings for these jobs. Finally, the uniformly behaving workload J4 also shows comparable performance with significant resource savings across these configurations (~40%). In summary, E-states enable dynamic scaling of resources providing high-performance when required (as for J1) and resource savings for low activity workloads like J3 and J4.

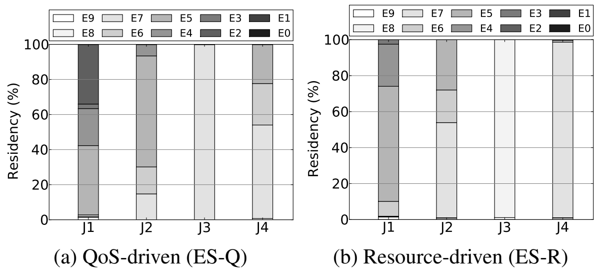


(a) QoS Score (b) Resource Usage

Figure 17: Results for memory E-state scaling

Concerning memory elasticity, Figure 17 shows the memstore application, using the load traces depicted in Figure 10b to vary the datastore size. The figure compares the QoS score and resource usage for the base configuration (NS-B) with the QoS-driven policy (ES-Q) under the M1 and M2 memory configurations. Additional experimentswith the resource-driven ES-R policy shows only minor variation for the memory scaling experiments. With a base case configuration consisting of 256MB of stacked DRAM (state E4), as the data in Figure 17a suggests,ES provides a 2.3x better QoS score for job J4, while performance is comparable for J1 and J3. J2 shows a 15% performance loss with scaling due to its varying memory usage, causing frequent scaling operations. Comparable behavior is seen across the two memory configurations (M1 and M2). The resource usage results in Figure 17b illustrate that ES policies significantly reduce the use of fast memory of jobs J1, J2, and J3 (75%, 70%, and 25%, respectively). In comparison, J4 observes a 50% increase in its resource usage due to its large memory footprints. Overall, elastic resource scaling using HeteroVisor provides a 30% lower stacked memory usage while maintaining performance.

Figure 18 shows the residency distribution (%) in each E-state for each of the four jobs, for the CPU scaling experiments. The states are color coded by their gray-scale intensity, meaning that a high-performance E-state is depicted by a darker color in comparison to a low-performing E-state. The graphs in Figures 18a and 18b correspond to the ES-Q and ES-R policies. As seen in the figures, different E-states dominate different workloads. J1 has large shares of E-states E2, E4, and E5 due to its high activity profile. For the low-CPU workload J3, the slower states E7 and E8 are dominant under the ES-Q and ES-R policies respectively. Similarly, J4 spends the majority of its execution time in states E7 and E5,while J2 makes mixed use of the E8, E7, E6, and E5 states. The results show the rich use of E-states, differing across andwithin workloads.

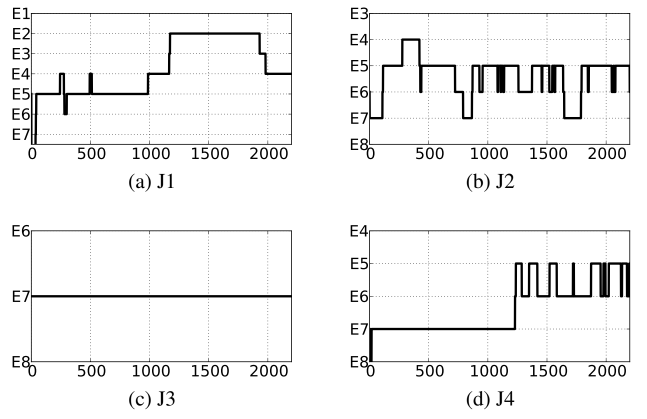


(a) QoS-driven (ES-Q) (b) Resource-driven (ES-R)

Figure 18: E-state residencies for two scaling policies

The corresponding E-state switch profiles for the ES-Q policy are shown in Figure 19. Both J1 and J4 stay in lower E-states initially and scale up when demand increases. J3 stays in a single E-state, while J2 has several E-state transitions due to its variable load. In summary, results make clear that HeteroVisor successfullyand dynamically scales resources to match the varying input load requirements seen by guest VMs.

Interesting about these results is that HeteroVisor exploits platform heterogeneity for dynamically scaling the resources neededby guests to meet desired application perfor-



(a) J1 (b) J2 (c) J3 (d) J4

Figure 19: E-state switch profiles showing usage of various states (x-axis = time (s), y-axis = E-states)

mance/cost trade-offs. As shown by the experimental data, the approach not only better services load peaks in comparison to homogeneous platforms (up to 2.3x), but it also provides savings (an average 21% for CPU and 30% for memory) to applications by scaling down their resources during idle periods. For all of these activities, E-state drivers can be customized to meet different user requirements, which we demonstrate experimentally with policies thateither meet high QoS requirement using an aggressive policy or that reduce resource usage while maintaining performance by using a conservative policy.

## 5. Related Work

**Resource Management in Clouds:** There has been substantial prior work on elastic resource scaling for server systems. In comparison to cluster-level scaling solutions [14, 19], HeteroVisor focuses on platform-level, fine-grained resource scaling. RaaS [1] and Kaleidoscope [6] argue in favor of fine-grained resource management for future cloud platforms, as also explored in our work. Q-Clouds, VirtualPower, AutoPilot, and CloudScale propose hypervisor-level mechanisms for elastic scaling of cloud resources [32, 34, 37, 42]. However, none of these address the effective use of platform-level heterogeneity in multiple platform resources. Several techniques have been developed for fair sharing of resources in cloud environments [15, 49]. Similarly, market-based allocation methods for datacenter applications have also been analyzed [17, 43, 47]. Such methods can be incorporated into HeteroVisor to ensure efficient operation and provide fairness.

**Heterogeneous Processor Scheduling:** Earlier work has demonstrated the need for compute heterogeneity advocating wimpy and brawny cores to efficiently support a wide variety of applications [3, 5, 20, 26, 48], as well as shown its presence in datacenters [36]. Several implementations of heterogeneous processor architectures have been released by various CPU vendors [16, 35]. In order to manage these platforms, appropriate OS-level [7, 23, 24, 39, 41, 45] and VMM-level [22, 25] techniques have been developed to efficiently run applications on heterogeneous cores. HeteroVisor adopts an alternative approach that hides heterogeneity from the OS scheduler, exposing a homogeneous scalable interface. Finally, several heterogeneity-aware cloud schedulers have also been proposed [8, 33] which are complementary to HeteroVisor that works at the platform level.

**Heterogeneous Memory Management:** The detection of memory usage behavior of virtual machines has been exploredin previous work [21, 31, 46]. In comparison, we usepage-table access bits to detect not only the working set sizes but also provide 'hotness' informationabout each page to guide page placement. Similarly, various methods for balancing memory allocation among competing VMs also exist which can be incorporated into our design for improving efficiency [2, 50]. Concerning heterogeneous memory, several architectural solutions have been proposed for page placement strategies in such systems involving NVRAM, DRAM caches, and disaggregated memory [11, 28, 40, 44]. In comparison, our work focuses on software-controlled memory management to more efficiently utilize stacked DRAM. There is also increasingly more emphasis on memory voltage scaling efforts [9, 10]. HeteroVisor approach goes beyond voltage scaling to support heterogeneous resources for efficient operation.

## 6. Conclusions & Future Work

This paper presents the HeteroVisor system for managing heterogeneous resources in elastic cloud platforms, providing applicationswith fine-grained scaling capabilities. To manage heterogeneity, it provides the abstraction of elasticity (E) states to the guestvirtual machine, which an E-state driver can use to elastically request resources on-demand. The proposed abstractions generalizeto managing multiple resource types and levels of resource heterogeneity. Demonstrating its application to CPU and memory, we present techniques to manage these heterogeneous resources in an elastic manner. The HeteroVisor solution is implemented in the Xen hypervisor along with a simple E-state driver realizing two scaling policies, QoS-driven and resource-driven. Experimental evaluations are carried out using real-world traces on an emulated heterogeneous platform. They show that HeteroVisor can provide VMs with the capabilities to quickly obtain resources for handling load spikes and/or to minimize cost during low load periods.

There are multiple possible directions for future work. Investigating challenges with the design of fine-grain resource management policies for requesting and allocating resources in the presence of multiple competing users is one future direction. Market based allocation mechanisms based on game theory become relevant in this context. In addition, using elasticity-states with multiple platform resources and multiple levels of heterogeneity is also interesting.

## References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (RaaS) cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[2] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 41–52, New York, NY, USA, 2014. ACM. .

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14. ACM, 2009. .

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. .

[5] L. A. Barroso. Brawny cores still beat wimpy cores, most of the time. *Micro, IEEE*, 30(4):20 –24, july-aug. 2010. ISSN 0272-1732. .

[6] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 273–286, New York, NY, USA, 2011. ACM. .

[7] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 225–236, Washington, DC, USA, 2012. IEEE Computer Society.

[8] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM. .

[9] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: active low-power modes for main memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 225–238. ACM, 2011. .

[10] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. CoScale: Coordinating CPU and memory system DVFS in server systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 143–154. IEEE, 2012. .

[11] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10. IEEE, 2010. .

[12] A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Commun. ACM*, 52(12):48–57, Dec. 2009. .

[13] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 263–270. IEEE Computer Society, 2012. .

[14] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch. SOFTScale: stealing opportunistically for transient scaling. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 142–163, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11. USENIX Association, 2011.

[16] P. Greenhalgh. Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7. White paper, ARM, Sept 2011.

[17] M. Guevara, B. Lubin, and B. C. Lee. Navigating heterogeneous processors with market mechanisms. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 95–106, 2013. .

[18] J. L. Hellerstein. Google cluster data. Google research blog, Jan. 2010. Posted at /urlhttp://googleresearch.blogspot.com/2010/01/google-cluster-data.html.

[19] Y.-J. Hong, J. Xue, and M. Thottethodi. Dynamic server provisioning to minimize cost in an IaaS cloud. In *Proceedings of the international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 147–148, New York, NY, USA, 2011. ACM. .

[20] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM. .

[21] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 14–24. ACM, 2006. .

[22] V. Kazempour, A. Kamali, and A. Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 85–96, New York, NY, USA, 2010. ACM. .

[23] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[24] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36. IEEE, 2003.

[25] Y. Kwon, C. Kim, S. Maeng, and J. Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 45–56, New York, NY, USA, 2011. ACM. .

[26] W. Lang, J. M. Patel, and S. Shankar. Wimpy node clusters: what about non-wimpy workloads? In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 47–55. ACM, 2010. .

[27] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 451–462. ACM, 2012. .

[28] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12. IEEE, 2012. .

[29] G. H. Loh. 3D-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464. IEEE Computer Society, 2008. .

[30] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung. Challenges in heterogeneous die-stacked and off-chip memory systems. In *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*, Feb 2012.

[31] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 3:1–3:15, Berkeley, CA, USA, 2007. USENIX Association.

[32] R. Nathuji and K. Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 265–278. ACM, 2007. .

[33] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 5–. IEEE Computer Society, 2007. .

[34] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250. ACM, 2010. .

[35] Nvidia. Variable SMP: A multi-core CPU architecture for low power and high performance. White paper, 2011.

[36] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[37] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. .

[38] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present and future. *Linux Symposium*, 2:223–238, 2006.

[39] S. Panneerselvam and M. M. Swift. Chameleon: operating system support for dynamic processors. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 99–110, New York, NY, USA, 2012. ACM. .

[40] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM. .

[41] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *5th EuroSys*, pages 139–152, New York, NY, USA, 2010. .

[42] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM. .

[43] T. Somu Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *19th Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 161–176. ACM, 2014. .

[44] K. Sudan, K. Rajamani, W. Huang, and J. Carter. Tiered memory: An iso-power memory architecture to address the memory power wall. *Computers, IEEE Transactions on*, 61 (12):1697–1710, Dec 2012. ISSN 0018-9340. .

[45] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[46] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX conference on Operating systems design and implementation*, OSDI'02, Berkeley, CA, USA, 2002. USENIX Association. .

[47] W. Wang, B. Liang, and B. Li. Revenue maximization with dynamic auctions in IaaS cloud markets. In *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pages 1–6, 2013. .

[48] D. Wong and M. Annavaram. KnightShift: Scaling the energy proportionality wall through server-level heterogeneity. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 119–130. IEEE Computer Society, 2012. .

[49] S. M. Zahedi and B. C. Lee. REF: Resource elasticity fairness with sharing incentives for multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 145–160. ACM, 2014. .

[50] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 21–30. ACM, 2009. .