A Comparative Look at Adaptive Memory Management in Virtual Machines

José Simão
INESC-ID Lisboa
ISEL, Portugal
Email: jsimao@cc.isel.ipl.pt

Jeremy Singer
University of Glasgow, UK
Email: jeremy.singer@glasgow.ac.uk

Luís Veiga
INESC-ID Lisboa
Instituto Superior Técnico, Portugal
Email: luis.veiga@inesc-id.pt

***Abstract*—Memory management plays a vital role in modern virtual machines. Both system- and language-level VMs manage memory to give the illusion of a unbounded allocation space although the underlying physical resources are limited. One of the main challenges for memory management is the range of dynamic characteristics of the workloads. Researchers have developed a large body of work using different mechanisms and dynamic decision making to specialize the memory management system to specific workloads. This design can be considered as a *control loop* where sensors are monitored, decisions are made and actions are performed by actuators. Nevertheless as is common in systems research, improvement in one property is accomplished at the expense of some other property. In this work we survey different techniques for adaptive memory management expressed as a control loop. We propose to analyse memory management in virtual machines using three seemingly orthogonal characteristics: responsiveness (R), comprehensiveness (C) and intricateness (I). We then present the details of an extensible classification framework which emphasizes the tradeoffs of different approaches. Using this framework, some representative state of the art systems are evaluated showing inherent tensions between R, C and I.**

***Index Terms*—Memory Virtualization, Adaptability, Quantitative analysis**

I. Introduction

Regardless of the target environment, designing memory management strategies is a demanding task. Virtual machines (VMs), either virtual machine monitors (Sys-VMs) or high-level language runtimes (HLL-VMs), add an extra level of complexity.

Sys-VMs (e.g. Xen) host multiple isolated guest instances of an operating system (OS) on multi-core architectures, sharing computational resources in a secure way. From an abstract perspective, managing memory in a Sys-VM is a generalization of operations performed by a classical OS [1]. However, in practice, many Sys-VMs gain performance advantages by dynamically adapting their memory management strategies.

HLL-VMs (e.g. JVM) have a single guest application, even when there is more than one address space, i.e. *domains* in the Common Language Runtime (CLR). Again, HLL-VMs adapt memory management decisions based on the dynamics of a given workload. Actions include heap resizing or, in more extreme scenarios, changing the garbage collection algorithm to one that saves memory at the expense of some performance [2].

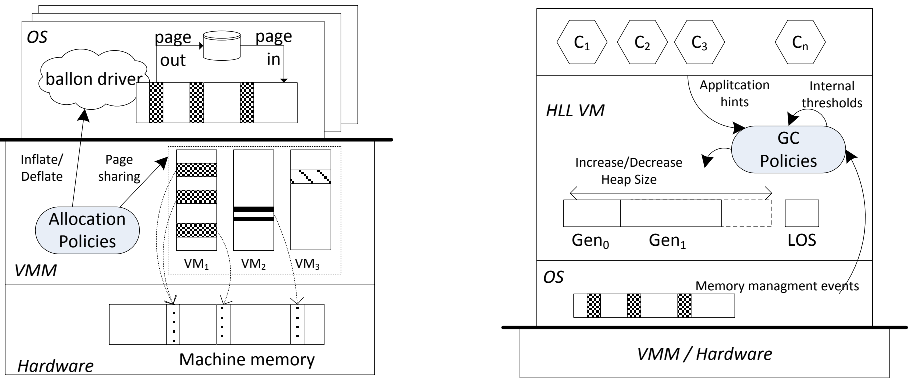
Conceptually, Sys-VMs and HLL-VMs perform the same memory management task, i.e. they mediate hosted application access to an underlying potentially scarce, address space. We argue that there are potential overlaps and unexploited synergies between Sys-VM and HLL-VM memory manager activity and adaptation. It might be the case that techniques from one domain could be transferred profitably to the other. Alternatively, in a system stack it might be possible for cross-layer exchange of information between these two levels of VMs to enable co-operative adaptation. Recent works (e.g. [3], [4]), where memory management of both domains cooperate to achieve better performance, show that this synergy is indeed important and worth exploring.

Existing surveys of virtualization technologies (e.g. [5]) tend to focus on a wide variety of approaches which sometimes results only in an extensive catalog. In this paper we present our current results of using a classification framework to characterize the adaptation techniques in memory management of Sys-VMs and HLL-VMs. Furthermore, several existing systems are analyzed and compared using this framework. In Section II we detail the rationale of an extensive classification framework aiming to characterize VM memory management. Section III discusses the techniques used in each type of VM memory management and presents them as part of a control loop. Section IV surveys several state of the art approaches from the literature. Using the framework previously presented, we show the inherent tension between responsiveness, comprehensiveness and intricateness. The final conclusions are presented in Section V.

II. Comparing the Adaptive Loop of Memory Management

In this work we aim to find a systematic way to compare different systems that use adaptive memory management techniques. An illustration of the control loop in each of these domains is depicted in Figure 1.

We have constructed a framework to quantitatively analyze and classify adaptive memory management techniques and, as a result, the overall systems that use them. The framework is named *RCI* because it is based on three criteria: *Responsiveness*, which represents how fast the system is able to adapt, *Comprehensiveness*, which takes into account the breadth and scope of the adaptation process, and *Intricateness*, which



(a) (b)

Fig. 1: The control loop for adaptive memory management in (a) system-level and (b) high-level language virtual machines.

considers the complexity of the adaptation process. Each of these criteria can be further detailed for each of the adaptability loop. For example, when classifying the monitoring phase, *Responsiveness* decreases with the overhead introduced by the techniques used, *Comprehensiveness* depends on the quantity of the monitored sensors and *Intricateness* is the amount of interference and complexity of the monitoring sensors.

We argue that these criteria form a fundamental tension: *A given adaptation technique aiming at achieving improvements on two of these aspects, can only do so at the cost of the remaining one.* As a result, systems build around these techniques inherit this tension and must compromise one or two criteria to favor another.

The RCI framework is indeed a generalization of common trade offs in systems research, applied in the context of the adaptive sub-systems of virtual machines. A prominent example of such trade offs, but in the context of distributed systems, is the CAP conjecture [6]. CAP argues that there is a tension in large-scale distributed systems among (C)onsistency, (A)vailability, and tolerance to (P)artitions. Although our research work is still undergoing, we think that a precise characterization of our framework classification metrics will help to show the major tensions in adaptive systems, and in particular, those present in adaptive memory management.

A. Quantitative analysis

In order to quantitatively compare different systems (e.g. more responsive or more comprehensive), each of the previously discussed metrics must be assigned with a quantitative value, which depend on the analyzed adaptation technique. Table I presents the nature of these metrics.

	Responsiveness	*Comprehensiveness*	*Intricateness*
Monitor	ISL	Q	SL
Decision	PT	Q	IC
Action	ISL	Q	SL

TABLE I: Quantitative units of the classification metrics

Table II shows the meaning of each metric for each of the quantitative values that the framework allows techniques to be classified (i.e. 1, 2 or 3). Quantitative (Q) intervals, Intrinsic Complexity (IC), and Processing Times (PT) used in the framework are presented. Also, two notes are worth noting. First, *System level* (SL) represents the natural organization of a computer system, assigning 1 to hardware, 2 to OS and hypervisor and 3 to applications. *Inverse system level* (ISL) uses this scale in reverse order so that the term Responsiveness can be understood as described previously. Second, for the decision step of the control we adapt the criteria of Maggio et al. [7].

Level	1	2	3
Q	[1..2]	[3..4]	[4..N]
SL	hardware	hypervisor/OS	application
ISL	application	hypervisor/OS	hardware
PT	milliseconds	seconds	minutes
IC	simple	medium	complex

TABLE II: Relation between classification levels (on top) and classification metrics

To better understand how the framework is used, we consider some hypothetical techniques, $T_a..T_f$. After having a classification of each technique according to Table II the framework builds the RCI of a system by aggregating each criteria value. For a given system, S_α, the three criteria of the framework, responsiveness, comprehensiveness and intricateness, are represented by $R(S_\alpha)$, $C(S_\alpha)$, $I(S_\alpha)$, respectively. The corresponding criteria of each technique (e.g. T_a) used by S_α is summed (e.g. $R(S_\alpha) = \sum_t responsiveness(t)$). Using these mock techniques, Table III presents, in the bottom row, the resulting RCI of S_α.

System	*Phase*	*R*	*C*	*I*
Sa	Monitor	Ta(1)	Ta(2)	Ta(3)
	Decision	Tc(3)	Tc(2)	Tc(3)
	Action	Tf(1)	Tf(2)	Tf(1)
RCI		5	6	7

TABLE III: RCI of hypothetical system S_α

III. Adaptation Techniques

In this section we survey relevant adaptation techniques of the adaptive control loop, found in the literature of Sys-VMs and HLL-VMs. We conclude the section with the RCI analysis of each technique.

A. Page management in Sys-VMs

The Virtual Machine Monitor (VMM) can regulate the access of multiple guest OS (each running on a different VM) to the underlying physical memory. In order to do so, the VMM adds a translation step, keeping the relation between what the OS believes is the *real* memory to what is in fact the *physical* hardware page. Nevertheless, using shadow pages, regular memory management operations made by the guest OS bypass the VMM so that the performance impact can be minimized. When page tables need to be changed the guest OS depends again on the VMM strategy to distribute the available pages among the running guests. In doing so, the VMM acts as an adaptive system, monitoring, deciding and acting, based only on information he directly accesses or hints collected outside the VMM, namely the application performance.

Regarding the monitoring phase, the VMM need to determined how pages are being used by each VM. To do so he must collect information regarding *i)* page utilization [8], [9] and *ii)* page contents equality or similarity [8], [10]. Some systems also propose to monitor application performance, either by instrumentation or external monitoring, in order to collected information closer to the application's semantics [3], [4]. To determine which guest OS must relinquish pages in favor of the current request, decisions are made using *i)* shares [8], *ii)* feedback control [11], LRU histogram [9] or *iii)* linear programming [3]. Finally, regarding the acting phase, the VMM can enforce *i)* page sharing [8] using the page tables controlled by the VMM or *ii)* page transfer between VMs using the guest OS balloon driver.

B. Garbage collection in HLL-VMs

Traditional GC algorithms are not fully adaptive in the sense that the allocation strategy for new objects, the organization of spaces used to do so and the way garbage is detected does not change during program execution. Nevertheless, most research and commercial runtimes incorporate some form of parameterized strategy regarding memory management e.g. [5]. To accomplish adaptation, the following metrics are tracked: *i)* memory structure dimensions (e.g. total heap size, nursery size) [12], [13], *ii)* the program behavior (e.g. allocation rate, stack height, key objects) [2] and, *iii)* relevant events in the operating systems (e.g. page faults, allocation stalls) [14], [15].

Decisions regarding the adaptation of heap related structures are taken either *i)* offline or *ii)* inline with execution. Offline analysis takes in consideration the result of executing different programs to see which parameter or algorithm has the best performance for a given application. Inline decisions must be taken either based on a mathematic model or on some kind of heuristic. Some authors have elaborated mathematical models of objects' lifetime, e.g. using thermodynamics [16] or radioactivity [17]. These models are mostly used to give a rationale of the GC behavior, rather than being used in a decision process [16]. So, most systems have a decision process based on some kind of heuristics. The decision process include *i)* machine learning *ii)* control theory and *iii)* microeconomic theories such as the elasticity of demand curves.

Adaptability regarding memory management aims to improve overall system performance. Classic GC algorithms provide base memory virtualization. Recent work has focused on optimizing memory usage and execution time, taking in consideration not only the program dynamics and but also its execution environment. Some work also adapts GC to avoid memory exhaustion in environments where memory is constrained. To accomplish this, actions regarding GC adaptability are related to changing: *i)* heap size [12], *ii)* GC parameters [13] *iii)* GC algorithm [2].

C. Summary of adaptation loop techniques

Figure 2 present a summary of techniques used in the adaptation loop of Sys-VMs and HLL-VMs. The two have related approaches to *monitoring* by relying on information provided by the operating system, mostly the underlying host OS, to assess memory utilization and application performance or behavior. There is a range of approaches to *decision*. HLL-VMs rely on higher-level information about memory structures (objects). Generally a summary of this data feeds into simple heuristic models, although recent work replaces specialized heuristics with machine learning, control theory and microeconomics. In Sys-VMs, the input data is less high-level. Again, specialized heuristics are common but sometimes linear optimization is applied. It is in the *action* domain that the approaches differ the most, with Sys-VMs performing adaptations at the page level and HLL-VMs making more refined adjustments to the object heap and to GC parameters, or switching algorithms altogether.

D. RCI classification of techniques

Figure 3 uses a triangular chart to represent techniques presented in this section. In each figures, techniques are further categorized among the three steps of the adaptability loop: Monitoring, Decision, and Action.

In the next section, we analyze relevant works regarding monitoring and adaptability in virtual machines, both at system as well as managed languages level. The RCI framework is used to compare different systems and better understand how virtual machine researchers have explored the tension between responsiveness, comprehensiveness and intricateness.

IV. Systems

In this section we survey some of the most relevant works regarding the use of the techniques identified in Section III. We conclude presenting the RCI overall analysis of these systems and some preliminary observations.

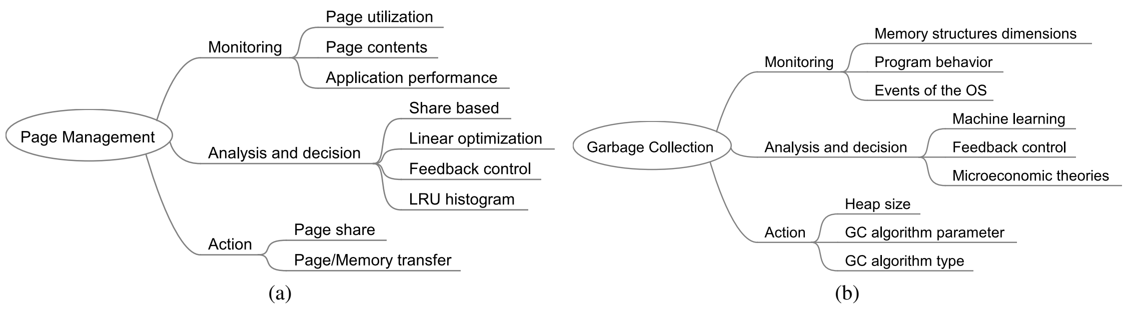
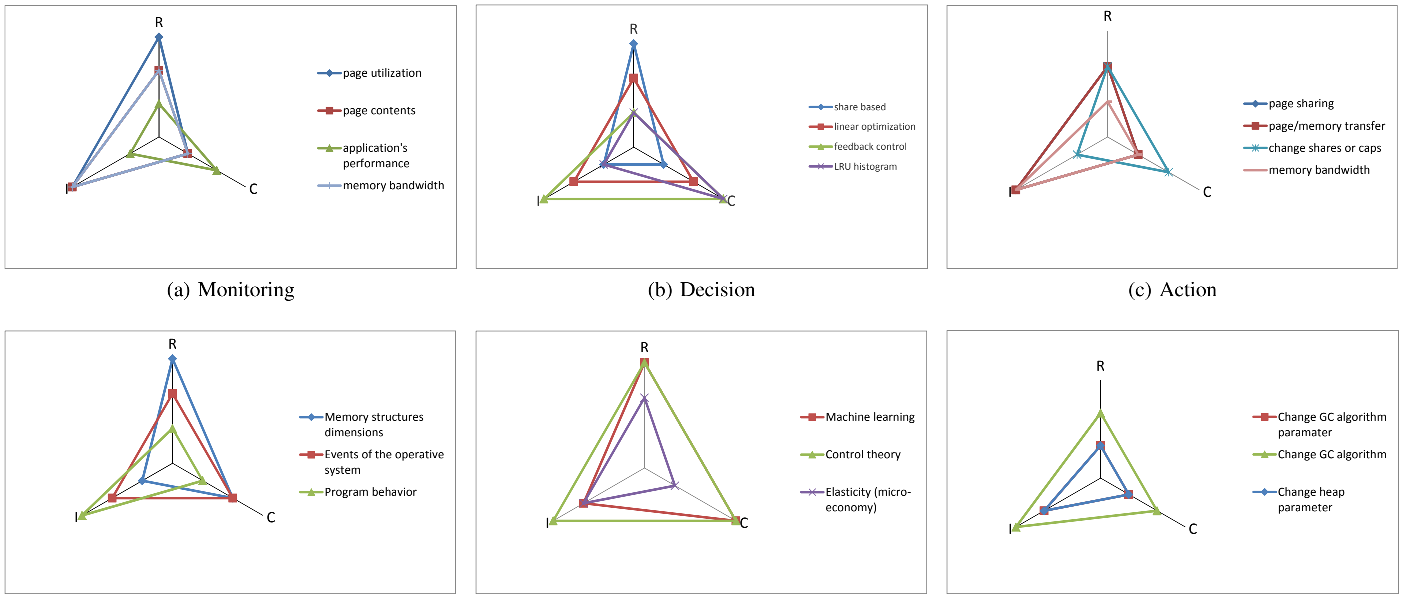


Fig. 2: Typical techniques used by Sys-VM (a) and HLL-VM (b) to monitor, control and enforce



(a) Monitoring (b) Decision (c) Action

(d) Monitoring (e) Decision (f) Action

Fig. 3: Relation of *responsiveness*, *comprehensiveness* and *intricateness* for the different techniques used in the control loop of Sys-VM (a-c) and HLL-VM (d-f)

A. *VMM Memory Management*

a) Overbooking and Consolidation: In [11], Heo et al. use a feedback control mechanisms to dynamically allocate memory in an environment where multiple virtual machines share the same host. They show that allocating memory in such an overcommitted environment without taking also in account the CPU results in significant service level violations. Their sensors measure memory allocation and usage and also the application performance (i.e. response time of an Apache Web server). Memory allocations are collected from the balloon driver along with page fault rates from the `/proc` file system.

b) VMMB: In [18], Min et al. presents VMMB, a Virtual Machine Memory Balancer for Unmodified Operating Systems. It uses the LRU histogram to estimate memory demand to periodically re-balance the memory allocated to each VM. Their algorithm determines the memory allocation size of each VM while it strives to globally minimize the page miss ratio. Their sensors are looking at nested page faults and to the guest swapping, using a pseudo swap device for monitoring. They act of the balloon driver to enforce each VM new memory size. When the balloon cannot collect enough memory, VMMB uses a VMM-level swapping to select a set of victim pages and immediately allocate memory to a beneficiary VM.

c) Ginkgo: Ginkgo [3] takes into account application performance hints to determine the page management strategy of the VMM. Doing so, it allows cloud providers to run more System VMs with the same memory. Ginkgo uses a profiling phase where it collects samples of the application performance, the submitted load and the memory used to process each request. During operation, Ginkgo uses the previously built model and, using a linear program, determines the memory allocation that, for the current load, maximizes the application performance (e.g. throughput). Ginko's performance is also due to the use of JVM-level balloon that is able to reclaim memory faster from Java-base applications (e.g. WebSphere Application Server). A similar approach is followed by Salomie et al. [4].

d) Difference engine: Gupta et al. [19] is an extension to the Xen VMM which supports sub-page sharing using a novel approach that builds patches to pages by using the difference relative to a reference page. Similar pages are identified by comparing hash value of randomly selected parts of different pages. In-memory compression of infrequently accessed pages is made using multiple algorithms.

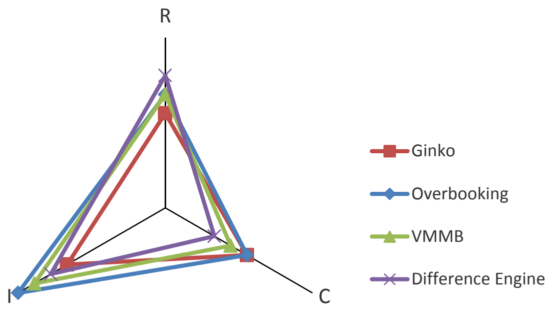


Fig. 4: Relationship of *R*, *C* and *I* for different memory management works in system-level VMs

Analysis: When looking at Figure 4, most works in Sys-VM exchange *comprehensiveness* (e.g. low quantity of monitored sensors and actuators) for the use of more *responsiveness* (e.g. monitor page utilization) and *intricate* techniques (e.g. feedback control techniques).

B. Garbage Collection

Garbage collection is known to have different performance characteristics with different applications [2], [20]. The remainder of this section analyzes recent works belonging to one of the following categories: *i)* adjust heap related parameters (e.g. nursery size, total heap size) [14], [12]; *ii)* algorithms that take execution environment events into account [21]; *iii)* VMs that switch the GC algorithm at runtime [2]. Common to all these solutions is the goal to decrease application's total execution time or, in some scenarios, to continue operation despite memory exhaustion.

e) Isla Vista and the allocation stalls: Grzegorczyk et al. [14] take into account *allocation stalls*. In Linux, a process will be stalled during the request of a new page if the system has very few free memory pages. If this happens, a resident page must be evicted to disk. This operation is done synchronously during page allocation. They have implemented an algorithm that grows the heap linearly when there are no allocation stalls. Otherwise, the heap shrinks and the growth factor for successive heap growth decisions is reduced, in an attempt to converge to a heap size that balances the tradeoff between paging and GC cost. This heap sizing behavior is inspired by the exponential backoff model for TCP congestion control, where transmission rate relates to heap size, and packet loss relates to page faults.

f) Resource-based GC: Hertz et al. [15] observe that the same application operating with different heap sizes can perform differently if the heap size is under or over dimensioned, resulting in many collections or many page faults, respectively. Based on this observation they have devised the time-memory curve, that is, the shortest running time of a program independently of his heap size for a given amount of physical memory. Their approach allows that the heaps of multiple applications remain small enough to avoid the negative impacts of paging, while still taking advantage of any memory that is available within the system. They have modified the slow path of the GC (i.e. code path that can result in tracing alive objects) to also take in account two conditions: if the resident set has decreased or if the number of page faults have increased. If any of this conditions is true, GC is triggered. They call it a *resource-driven* garbage collection.

g) GC economics: In [12], Singer et al. discuss the economics of GC, relating heap size and number of collections with the price and demand law of micro-economics - with bigger heaps there will be less collections. This relation extends to the notion of elasticity to measure the sensitivity of the heap size to the size of the number of GCs. They devise an heuristic based on elasticity to find a tradeoff between heap size and execution time. The user of the VM provides a target elasticity. During execution, the VM will take into account this target to grow, shrink or keep the heap size. Doing so, the user can supply a value that will determine the growth ratio of the heap, independently of the application specific behavior.

h) Intelligent Selection of Application-specific Garbage Collectors: Singer et al. [22] show that Java applications have differing execution characteristics in different GC regimes. They use an offline machine learning algorithm based on decision trees to generate a classifier that, given a small profile run of a benchmark, can predict an appropriate GC algorithm for efficient execution. In a related work, this approach is used in order to improve the performance of a MapReduce's Java implementation for multi-core hardware. For each relevant benchmark, machine learning techniques are used to find the best execution time for each combination of input size, heap size and number of threads in relation to a given GC algorithm (i.e. serial, parallel or concurrent). The goal is to decide about a GC policy when a new MapReduce application arrives. The decision is made locally to an instance of the JVM.

i) GC switch: Soman et al. [2] add to the memory management system the capacity of changing the GC algorithm during program execution. The system considers program annotations (if available), application behavior, and resource availability to decide when to switch dynamically, and to which GC it should switch. The modified runtime incorporates all the available GCs into a single VM image. At load time all possible virtual memory resources are reserved. The layout of each space (i.e. nursery, Mark-Sweep, High Semispace, Low Semispace) is designed to avoid a full garbage collection for as many different switches as possible. For example, a switch from Semi-Space to Generational Semi-Space determines that the allocation site will be done at a nursery space, but the two half-spaces are shared. Switching can be triggered by points statically determined by previous profiling the application execution or by dynamically evaluating the GC load versus the application threads. If the load is high they switch from a

Semi-Space (which performs better when memory is available) to a Generational Mark-Sweep collector (which performs better when memory is constrained).

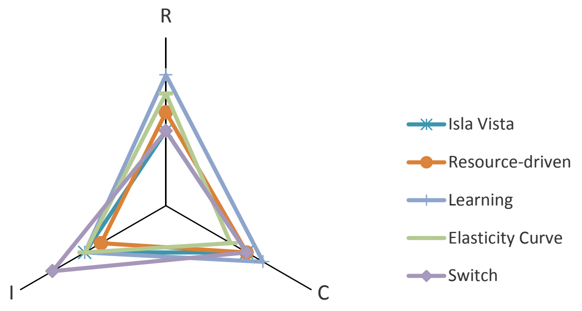


Fig. 5: Relationship of *R*, *C* and *I* for different memory management works in high-level language VMs

Analysis: When looking at Figure 5, the system with greater *intricateness*, GC Switch [2], which is also reasonably *comprehensive* in comparison to others, is the least *responsive*. Our current observation of the others is that they exchange *intricateness*, for *comprehensiveness* and *responsiveness*.

V. Conclusions

Currently, data centers in the context of cloud infrastructures make extensive use of virtualization to achieve workload isolation and efficient resource management. This is carried out primarily by means of virtualization technology. Virtualization mechanisms to enforce resource management are present both at hypervisors (e.g. Xen, ESX) and high level virtual machines (e.g. CLR, Java). Although the services offered by each of these two software layers are used or extended in several works in the literature, the community lacks an organized and integrated perspective of the mechanisms and strategies used at each virtualization layer regarding resource management and focusing on adaptation.

We have further detailed a classification framework which aims to understand the trade offs that underpin adaptive subsystems of virtual machines, particularly regarding memory virtualization. Future work needs to be done regarding the analysis of more fundamental memory adaptive techniques and the systems that use them. Doing so, we can further understand the boundaries of each technique and then eventually argue for the need of information exchange between these two layers.

Acknowledgments: This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, projects PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/102250/2008, PEst-OE/EEI/LA0021/2013, the PROTEC program of the Polytechnic Institute of Lisbon (IPL) and UK EPSRC grant EP/L000725/1.

References

[1] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[2] S. Soman and C. Krintz, "Application-specific garbage collection," *Journal of Systems and Software*, vol. 80, pp. 1037–1056, July 2007.

[3] M. Hines, A. Gordon, M. Silva, D. D. Silva, K. D. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with ginkgo," in *CloudCom '11: 3rd IEEE International Conference on Cloud Computing Technology and Science*, 2011.

[4] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 337–350.

[5] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," in *Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, ans Adaptation*, 2005.

[6] E. A. Brewer, "A certain freedom: thoughts on the cap theorem," in *PODC*, A. W. Richa and R. Guerraoui, Eds. ACM, 2010, p. 335.

[7] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, "Comparison of decision-making strategies for self-optimization in autonomic computing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 4, pp. 36:1–36:32, Dec. 2012.

[8] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, December 2002.

[9] Z. Weiming and W. Zhenlin, "Dynamic memory balancing for virtual machines," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09, 2009, pp. 21–30.

[10] 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," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, October 2003.

[11] J. Heo, X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of xen virtual machines in consolidated environments," in *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, ser. IM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 630–637.

[12] J. Singer, R. E. Jones, G. Brown, and M. Luján, "The economics of garbage collection," *SIGPLAN Not.*, vol. 45, pp. 103–112, June 2010.

[13] J. Singer, G. Kovoor, G. Brown, and M. Luján, "Garbage collection auto-tuning for java mapreduce on multi-cores," in *Proceedings of the international symposium on Memory management*, ser. ISMM '11. New York, NY, USA: ACM, 2011, pp. 109–118.

[14] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski, "Isla vista heap sizing: Using feedback to avoid paging," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 325–340.

[15] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard, "Waste not, want not: resource-based garbage collection in a shared environment," in *Proceedings of the international symposium on Memory management*, ser. ISMM '11. New York, NY, USA: ACM, 2011, pp. 65–76.

[16] H. G. Baker, "Thermodynamics and garbage collection," *SIGPLAN Not.*, vol. 29, pp. 58–63, April 1994.

[17] W. D. Clinger and L. T. Hansen, "Generational garbage collection and the radioactive decay model," in *Proceedings of the 1997 ACM Conference on Programming Language Design and Implementation*, 1997, pp. 97–108.

[18] C. Min, I. Kim, T. Kim, and Y. I. Eom, "Vmmb: Virtual machine memory balancing for unmodified operating systems," *J. Grid Comput.*, vol. 10, no. 1, pp. 69–84, Mar. 2012.

[19] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: harnessing memory redundancy in virtual machines," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322.

[20] F. Mao, E. Z. Zhang, and X. Shen, "Influence of program inputs on the selection of garbage collectors," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[21] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," *SIGPLAN Not.*, vol. 40, pp. 143–153, June 2005.

[22] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proceedings of the 6th international symposium on Memory management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 91–102.