Dynamic Memory Allocation in Cloud Computers using Progressive Second Price Auction

Eyal Posener

Dynamic Memory Allocation in Cloud Computers using Progressive Second Price Auction

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Systems

Eyal Posener

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tammuz 5773 Haifa June 2013

This research was carried out under the supervision of Prof. Assaf Schuster, in the Faculty of Computer Science.

Acknowledgements

I would like to thank the followings:

My supervisor, Prof. Assaf Schuster, for giving me the opportunity to work on this interesting research, for the research guidance, and for giving me the needed support.

My adviser, Orna Agmon Ben-Yehuda, for all the help and support in getting over obstacles, and for her guidance through the game theory field.

My lovely and supporting family, for their good and reasonable advices.

My girlfriend, Maria, for her understanding of my difficulties, for giving me the necessary time and space when needed, for guarding my insanity and for her support through rough time.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Physical memory is the most expensive resource in today's cloud computing platforms. Cloud providers would like to maximize their clients' satisfaction by renting precious physical memory to those clients who value it the most. But real-world cloud clients are selfish: they will only tell their providers the truth about how much they value memory when it is in their own best interest to do so. Under these conditions, how can providers find an efficient memory allocation that maximizes client satisfaction?

This research presents Ginseng, the first market-driven framework for efficient allocation of physical memory to selfish cloud clients. Ginseng incentivizes selfish clients to bid their true value for the memory they need when they need it. Ginseng continuously collects client bids, finds an efficient memory allocation, and re-allocates physical memory to the clients that value it the most.

In this research, an approach for a new type of application for efficient memory allocation in a dynamic memory cloud computer is suggested. The efficiency of the approach is demonstrated through a special developed benchmark called Memory Consumer and a modification of a widely used caching application called Memcached. An approach for notifying the guest in advance that its memory allocation is going to change is also presented. It is shown that by letting the guest prepare for a memory change, the performance can be improved. Higher memory usage without OS interference was enabled by changing the OS default configuration. It is demonstrated that in the not-overcommitted system, higher performance was achieved.

An experimental environment was developed to test Ginseng under different workloads, simulate it under different conditions, and compare the results to determine system efficiency. Ginseng was shown to achieve a $\times6.2$–$\times31.5$ improvement in aggregate client satisfaction, or $\times1.6$ in the application performance when compared with state-of-the-art approaches for cloud memory allocation. It achieved 83%–100% of the optimal aggregate client satisfaction.

Abbreviations and Notations

P	:	**Performance**. Any way to measure the performance of an application, per unit of time.
V	:	**Valuation**. The estimated benefit from the performance for a guest. Measured in $\frac{\$}{\text{hour}}$.
U	:	**Utility**. The results from the subtraction of all costs from the valuation. Measured in $\frac{\$}{\text{hour}}$.
p	:	**Unit price**. The price of unit of memory per hour. Measured in $\frac{\$}{\text{MB}\cdot\text{hour}}$.
$[r, q]$	:	**Allowed ranges**. A series of allowed memory ranges which are sent by the bidder as part of the auction protocol.
b	:	**Bid**. A composition of unit-price and allowed ranges. $b_i = (p_i, r_{i1}, q_{i1}, \ldots, r_{i,m_i}, q_{i,m_i})$.
$q_{\max}$	:	**Memory for auction**. The amount of memory that is proposed for auction. Measured in MB.
sw	:	**Social welfare**. A basic game-theoretic measurement which determines the overall player satisfaction for an allocation. $sw(a) = \sum_i V_i(a_i)$.
sc	:	**Social cost**. A generalization of the social welfare which is defined with the affine maximizer mechanisms. A function which represents the aggregate social satisfaction an arbitrary allocation.
m_0	:	**Bare memory**. Amount of memory allocation which is guaranteed to the player without participating in the auction.
$q'(t)$	:	**Extra allocation**. The allocation obtained by the allocation rule, that maximizes the social cost function, and will be allocated to the players.
$m(t)$	:	**Allocated memory**. The total allocation for the player in round t, $m(t) = m_0 + q'(t)$.
$p'(t)$	:	**Payment**. The payment obtained by the payment rule, and expresses the exclusion compensation principle.
p_0	:	**Memory exchange penalty parameter**. Determines how much memory exchange is expansive.

OC	:	**Memory Overcommitment**. The ratio between needed memory and available physical memory.
VM	:	**Virtual Machine**. Also referred to as the guest, a software implementation of a machine (computer) that executes programs like a physical machine.
l	:	**Load**. A measure of the application load. Measured in concurrent requests.
m	:	**Memory**. Amount of memory. Measured in MB.
$(\cdot)_i$	:	**Indexing of guests**.
$(\cdot)_j$	:	**Indexing of allowed ranges**.
α	:	**Reclaim Factor**. The percent of memory the host reclaims from the guests between auctions.
base	:	**Base memory**. Used with reclaim-factor, represent time depended bare memory, according to the last auction results, $\mathrm{base}_i(t) = \alpha \cdot m_{0i} + (1 - \alpha) \cdot \mathrm{m}_i(t-1)$.

Chapter 1

Introduction

Virtual machine environments are usually adopted by cloud service providers. Memory is one among several resources provided to the clients, to whom we refer as *guests*. Examples for selling resources are given in [LS99] for network bandwidth, and in [PHS$^+$09] for CPU and disk I/O. The memory resource might be considered the most important: it is expensive, and may not be expanded beyond the machine's physical limit. Additionally, memory is allocated differently than other resources and deserves special treatment: it has a long response time and a non-linear relation to performance.

Unlike physical hardware, virtual machine environments are flexible. The virtualization manager, to which we refer as *host* or *hypervisor*, may employ different memory allocation or memory overcommitment mechanisms.

When the hypervisor allocates the guests more memory than is physically available to it, we call this *memory overcommitment*. Since the guests are processes inside the host, their memory is actually allocated only when it is written. In this case, when the guests try to write to more memory than the host actually has, the host will swap their memory to disk, resulting in degraded performance.

Nowadays, the most common overcommitment techniques are deduplication and dynamic memory allocation. Deduplication, by which similar memory pages are shared among guests, is best when the guests are similar and static. Dynamic memory allocation techniques such as memory ballooning or C-Groups are best for a setup in which the running guests are heterogeneous and dynamically loaded.

In static memory allocation systems, at a given moment, some guests might be idle and have spare unused allocated memory, while other guests might be loaded and need more memory. Modifying the guests' memory allocation can increase the portion of used memory of the whole system, thus making it more efficient and improving the aggregate performance. Since the load of the system is usually dynamic, the memory allocation mechanism must be dynamic too.

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources to their clients. As competition between providers gets tougher and prices start going down, providers will need to continuously and ruthlessly reduce expenses, primarily by improving their hardware utilization. Physical memory is the most constrained and thus precious resource in use in cloud computing platforms today [Mag08, HGS$^+$11, GHDS$^+$11, HZPW09, NKG10, Wal02]. One way for providers to significantly reduce their expenses is by using less memory to run more client guest virtual machines on the same physical hosts.

Whereas today cloud computing clients buy a supposedly-fixed amount of physical memory for the lifetime of their guests, nothing stops their provider from overcommitting this memory. Clients today have no idea and no way to discern how much physical memory they are actually getting. Clients would much prefer to have full visibility and control over the resources they receive [OZN$^+$12, ABYST12]. They would like to pay only for the physical memory they need, when they need it [GGW10, AFG$^+$10]. By granting clients this flexibility, providers can increase client satisfaction. Therefore, finding an efficient allocation of physical memory on each cloud host—an allocation that gives each guest virtual machine precisely the amount of memory it needs, when it needs it, at the price it is willing to pay—poses benefits for clients, whose satisfaction is improved, and for providers, whose hardware utilization is improved.

Previous physical memory allocation schemes assumed fully cooperative client guest virtual machines, where the host knows precisely what each guest is doing, how much benefit additional memory would bring to it, and the importance of that guest's workload to the client [HGS$^+$11, GHDS$^+$11, HZPW09, NKG10]. However, when it comes to commercial cloud providers and their paying IaaS clients, none of these assumptions are realistic. Real-world clients act *rationally* and *selfishly*. They are *black boxes* with private information such as their performance statistics, how much memory they need at the moment, and what it is worth to them. Rational, selfish black-boxes will not share this information with their provider unless it is in their own best interest to do so.

When white-box models are applied to selfish guests, the guests have an incentive to manipulate the host into granting them more memory than their fair share. For example, if the host gives memory to those guests that will benefit more from it, each guest will say it benefits from memory more than any other guest. If the host gives memory to those guests that perform poorly with their current allocation, each guest will say it performs poorly. If the host allocates memory on the basis of passive black-box or grey-box measurements [Lit11, Mag08, JADAD06, Wal02] such as page faults, guests have an incentive to bias the measurement results, e.g., by inducing unnecessary page faults. Furthermore, black-box methods compare the guests only by technical qualities such as throughput and latency, which are valued differently by different guests under different circumstances.

In this work we address the cloud provider's fundamental memory allocation problem: How should it divide the physical memory on each cloud host among selfish black-box guests? A reasonable meta-approach would be to give more memory to guests who would benefit more from it. But how can the host compare the benefits of additional memory for each guest?

1.1 Research Contributions

The contributions of this research are described below.

1.1.1 Ginseng

Ginseng is a market-driven memory allocation framework for allocating memory efficiently to selfish black-box virtual machines. Ginseng is the first cloud platform to optimize overall client satisfaction for black box guests.

1.1.2 Memory Progressive Second Price (MPSP) auction

The MPSP auction is a game-theoretic market-driven incentive compatible, Pareto efficient and fair mechanism to handle completely black boxed clients. The MPSP induces auction participants to bid (and thus express their willingness to pay) for memory according to their true economic *valuations* (how they perceive the benefit they get from the memory, stated in monetary terms). In Ginseng, the host periodically auctions memory using the MPSP auction. Guests bid for the memory they need as they need it; the host then uses these bids to compare the benefit that different guests obtain from physical memory, and to allocate it to those guests which benefit from it the most. The host is not manipulated by guests and does not require unreliable black-box measurements.

Ginseng is the first full implementation of a single-resource Resource-as-a-Service (RaaS) cloud [ABYST12]. It is ready for a world of *dynamic-memory applications*—applications that can improve their performance when given more memory on-the-fly over a large range of memory quantities and can return memory to the system when needed. Dynamic-memory applications are still scarce.

1.1.3 New Approaches to Dynamic Memory Cloud Applications

We present a new programming approach to be used inside a guest in cloud computers with dynamic memory management. The *dynamic application* produces greater benefit from the allocated memory, and is capable of freeing the memory before it is taken in order to avoid thrashing. We developed a modified version of Memcached, a widely-used key-value storage cloud caching application, as well as Memory Consumer, a synthetic dynamic memory benchmark. Our experiments proved that dynamic applications perform better in our environment.

1.1.4 Explicit Hinting about Upcoming Memory Allocation Changes

We implemented and tested a mechanism which hints the application running inside the guest about an upcoming memory change. By doing so, we successfully make abrupt and large changes in the guest memory without causing thrashing, which slows the system down and degrades performance.

Our approach is similar to that proposed in [Vor13], and is by far more efficient than the change rate limitations used in [Wal02, Wan09, GHDS+11, HGS+11], in which the memory degradation is dependent on the OS memory management system, which should be aware of and respond to memory pressure.

1.2 Achievements

Ginseng achieves a $\times 6.2$ improvement in aggregate client satisfaction for Memory Consumer and $\times 31.5$ improvement for Memcached, and improvement of up to $\times 1.6$ in the performance, when compared with state-of-the-art approaches for cloud memory allocation. Overall, it achieves 83%–100% of the optimal aggregate client satisfaction.

1.3 Related Work

White-Box Memory Overcommitment Systems

Memory overcommitment can improve the performance and profits of cloud service providers. This is the reason for recent studies and projects in the area. In this section we will introduce some of the state-of-the-art systems in the field of memory overcommitment.

Nathuji et al. [NKG10] present *QClouds*, a hypervisor framework that controls resource allocation in order to improve the system QoS. Q-Clouds monitors virtualized hardware and uses QoS reports from cooperative agents running on the guests. In this way, it controls the resources in a closed loop. They consider resources such as CPU caches, memory bandwidth and I/O paths, obtained from a prescribed static pool called "headroom." All guests are first profiled on a "staging server," which uses a least mean square algorithm to produce a static linear relation between QoS and resource allocation in the interference-free environment. Guests also define Q-state, a discrete relationship between the QoS and the guest's willingness-to-pay. With this information, a relationship between allocation and revenue is constructed. The real-time resource allocation for best performance is found by solving a linear minimization problem constructed by substituting the gathered real-time information into guests' profiles and Q-states. They experimentally demonstrated the influence of headroom size on performance. The size shouldn't be too small or too large, so that enough resources will be available for good performance in a normal run, and so that resources can be added when the guest is loaded.

Hines et al. [HGS+11] present *Ginkgo*, a performance-driven MOC management system. As in Q-Clouds, the guests' performance-to-memory-allocation model is first profiled, this time as a non-linear function. This static information of the interference-free environment is used to make memory allocation decisions. Gingko monitors each guest for the running application performance and load, and for allocated memory. The collected data is processed with mathematical and heuristic algorithms, in order to determine the performance as a non-linear function of application load and allocated memory. By collecting the correlated functions and using real-time information, a linear optimization problem is created. Solving the problem with linear programming methods results in an estimation of the best memory allocation, which is then implemented with memory balloons. When compared to a non-overcommitted system, Ginkgo was shown to save up to 73% of physical memory, with no more than 7% degradation in application performance.

Palada et al. [PHS$^+$09] present a program called *Autocontrol.* The program controls the CPU and disk I/O of a guest and automatically adapts to dynamic workload changes, in order to achieve application SLOs. The system was built to control a scalable number of nodes, with two resolution levels: AppController, which controls an application within a node, and NodeController, which controls a node. At the application level, performance is controlled to reach a sufficient target value. They locally approximate the nonlinear behavior of application performance as linear. The present performance is estimated as a correlation of past performance and past and present resource allocation, with an adaptive correlation parameter. The adopted optimization model maximizes the performance. It is also minimizes the change in resource allocation, by adding it as a penalty term. The developed algorithm is general and, theoretically, has multi-input-multi-output (MIMO) support. In practice, AutoControl was tested only on controlling the CPU and disk I/O, and optimizing application throughput or average response time.

Heo et al. [HZPW09] studied dynamic memory allocation in Xen virtual machines, in addition to a development of memory and CPU controller. The controller controls the memory according to the memory utilization in the VM and the CPU according to the application performance. CPU sharing and memory usage were measured by sampling the `/proc` file system and were controlled by the Xen credit scheduler and Xen balloon driver respectively. They found a correlation of memory allocation and performance by testings in a sterile environment. The response time of a guest running a memory access application was measured while changing the workload and the memory allocation of the guest. It was shown that the response time was constant with memory utilization below 90%, but when memory utilization crossed the 90% mark, the response time rose significantly. Thus, the memory allocation control system was built such that the guest will utilize approximately 90% of the memory. The authors' demonstrated that all the hosted applications achieved their SLOs without creating CPU or memory bottlenecks. The performance was found to be satisfying, but the improvement of aggregate performance was not evaluated.

Vorontsov [Vor13] proposes the *mempressure control group* (CGroup), a control-group-based subsystem that notifies the application running in the CGroup about the memory pressure it is under. Its first API was three notification levels ("low", "medium", "OOM") that described the memory pressure of the CGroup. In [Vor13], a new API is proposed, in order to enable the kernel to request that the CGroup's user space applications free specific amounts of memory (by using an event file descriptor, `eventfd()`) before it reclaims the memory. The applications are able to tell the kernel how much memory they couldn't free. This mechanism allows both the kernel and the CGroup's applications avoid swapping.

Litke [Lit11] developed and tested *Memory Overcommitment Manager* (MOM), an application that manages memory overcommitment on KVM hosts, using `libvirt`. Data about the host and active guests is gathered, organized and evaluated. After evaluation in a configurable policy, guests' memory is controlled using memory ballooning and KSM, according to one of two policies proposed by the author. The first is to control memory balloons by checking swapping of host and guests. Initially, the guests are given the highest possible memory allocation, as defined in their `libvirt`'s `maxmem` parameter (which controls the balloon driver). The guests and host are periodically monitored, and when memory pressure is detected in the host, memory is taken from the guests back to the host. Now the host has enough memory, and the memory pressure is experienced instead by the guests, who try to alleviate it. The assumption is that the guests will be more efficient than the host in dealing with memory pressure, since they know better which memory should be swapped. The second policy is to control the KSM daemon, which merges identical duplicated pages. Since running the KSM daemon does incur some overhead, it is disabled by default. Since the KSM daemon increases the amount of free memory, it is invoked only when the free memory is below a defined threshold, and disabled again when it is above another defined higher threshold. MOM is partially effective: in [Lit11], two workloads were examined, one (Memknobs) showing 20% improvement in the aggregate throughput, while in the other (Cloudy), the mechanism caused increased disk utilization and had no effect on the overall throughput. MOM was evaluated with its first balloon control policy in this research, and its performance is presented as a reference to the developed system.

Gray-Box Overcommitment systems

Gray-box methods make decisions according to samples from the guest environment but can be fooled by a selfish guest, and like white-box methods, ignore the client's valuation of performance.

Magenheimer [Mag08] used the guests' own performance statistics to guide overcommitment. Jones, Arpaci-Dusseau, and Arpaci-Dusseau [JADAD06] inferred information about the unified buffer cache and virtual memory by monitoring IO and inferring major page faults. Zhao and Wang [Wan09] monitored use of physical pages. Waldspurger [Wal02] randomly sampled pages to find unused pages to reclaim, and introduced the "idle memory tax", which resembles our reclaim factor, to be described in section 2.1.2.

Black-Box Techniques

Gupta et al. [GLV+08] did not require any guest cooperation for their content based page sharing. Wood et al. [WTLS+09] allocated guests to physical hosts according to their memory contents. Gong, Gu and Wilkes [GGW10] and Shen et al. [SSGW11] used learning algorithms to predict guest resource requirements. Sekar and Maniatis [SM11] argued that all resource use must be accurately attributed to the guests who use it so that it can be billed.

Guest Hint Techniques

Schwidefsky et al. [SFM+06] used guest hints to improve host swapping. Milos et al. [MMHF09] incentivized guests to supply sharing hints by counting a shared page as a fraction of a non-shared page.

General Resource Allocation for Monotonically Rising, Concave Valuation Functions

Kelly [Kel97] used a proportionally fair allocation: clients bid prices, pay them, and get bandwidth in proportion to their prices. His allocation is optimal for *price taking* clients (who do not anticipate their impact on the price they pay). Popa et al. [PKRS11] traded off proportional fairness with starvation prevention.

Lazar and Semret [LS99] introduced the divisible good progressive second price (PSP) auction, and found an ε-Nash equilibrium for the requested resources under complete information.

Maillé and Tuffin [MT04a] extended the PSP to multi-bids, thus saving the auction rounds needed to reach equilibrium. Their guests disclosed a sampling of their resource valuation function to the host, which computed the optimal allocation according to these approximated valuation functions. One such single auction has the complexity of a single PSP auction, times the number of sampling points. They also showed that the PSP's social welfare converges to theirs [MT04b]. Non-concave or non-monotonically rising functions require more sampling points to express them with the same accuracy, thus increasing the multi-bid auction's complexity. Though a multi-bid auction is more efficient for static problems, it loses its appeal in dynamic problems, which require repeated auction rounds anyhow.

Other drawbacks of the multi-bid auction are that the guest needs to know the memory valuation function for the full range; that frequent guest updates pose a burden to the host; and that the guest cannot directly explore working points which currently seem less than optimal. (It can do so indirectly by faking its valuation function.)

Chase et al. [CAT+01] allocated CPU time assuming client valuations of the resource are fully known, concave, and monotonically increasing.

Google's GSP auction uses a limited bidding language and is not a VCG auction [EOS07].

Urgaonkar, Shenoy, and Roscoe [USR09] overbooked bandwidth and CPU cycles given full profiling information but did not address memory.

Ghodsi et al. [GZH+11], Dolev et al. [DFH+12] and Gutman and Nisan [GN12] considered allocating multiple resources to strategic guests whose private information is the relative quantities they require of the resources.

Auctions With Non-concave Valuations

Bae et al. [BBB+08] observed that non-concave valuations are common in the wireless industry and supported a single bidder with a non-concave valuation function.

Dobzinski and Nisan [DN10] presented truthful polynomial time approximation algorithms for multi-unit auctions with k-minded valuations. They only assumed that the valuations are non-decreasing (because they allow *free disposal*—shedding of unneeded goods), and did not require them to be concave, but allowed the guests to make queries before bidding.

Chapter 2

System Description

2.1 The MPSP Auction

Ginseng auctions memory on cloud computer platforms. It uses the MPSP mechanism, which is considered as an *affine maximizer*, and has the important property of *incentive compatibility.* This property determines that the player's best interest is to bid the true value of the good. This mechanism is a modification of Lazar and Semret's bandwidth auction [LS99], and we call it MPSP. The modification can handle non-concave and non-increasing valuation functions of the bidders, and it reduces the costly memory exchange.

Bare Memory

Each guest i is set up permanently with the *bare* minimal physical memory it requires to operate, denoted as m_{0i} (see section 2.3.1). This memory is charged for separately by a constant hourly fee. All the auctioning is done for memory on top of the bare allocation, and the final memory which guest i obtains is defined as:

$$m_i(t) = m_{0i} + q'_i(t) \quad ,$$

where $q' = \{q'_i\}$ is the set of extra allocations obtained through the auction.

VCG

VCG [Vic61, Cla71, Gro73] auctions optimize social welfare by incentivizing even selfish participants with conflicting economic interests to inform the auctioneer of their true valuation of the auctioned items. They do so by the *exclusion compensation* principle, which means that each participant is charged for the damage it inflicts on other participants' social welfare, rather than directly for the items it wins. VCG auctions are used in various settings, including Facebook's repeated auctions [LPLT12, Heg10].

Various auction mechanisms, some of which resemble the VCG family, have been proposed for *divisible* resources, in particular for *bandwidth sharing* [LS99, MT04a, Kel97]. For practical reasons, bidders in those auctions do not communicate their valuation for the full range of auctioned goods. One of these VCG-like auctions is Lazar and Semret's Progressive Second Price (PSP) auction [LS99]. None of the auctions proposed so far for divisible goods, including the PSP auction, are suitable for auctioning memory, because memory has two characteristics that set it apart from other divisible resources: first, the participants' valuation functions may be non-concave; second, transferring memory too quickly between two participants leads to waste.

2.1.1 An Affine Maximizer

An affine maximizer is a type of general social choice function that is a generalization of the VCG mechanism. The general form of an affine maximizer mechanism is defined by an allocation rule, and a payment rule.

The allocation rule guarantees that the resulting allocation, $q' = \{q'_i\}$, will maximize the *social cost* function. Social cost is a game-theoretic term which describes a function that expresses the social satisfaction from an allocation, as follows:

$$q' = \underset{a \in A'}{\operatorname{argmax}} \{ \sum_i \omega_i V_i(a) + c_a \} \quad , \tag{2.1}$$

where $a = \{a_i\}$ is a set of allocations for the set of corresponding bids, $A' \subseteq A$ is subset of all possible allocations, A, ω_i is a constant weight of bid i, V_i is the valuation of bidder i, and c_a is arbitrary constant that is dependent only on the allocation.

We would like the allocation rule to find the *Pareto efficient* allocation—there is no other allocation in which no player benefits less, and at least one player benefits more:

$$\mathrm{sc}_{\max} = \{ \mathrm{sc}_a \mid \exists i : a \in A,\ u \in A \setminus \{a\}, \begin{cases} V_{ja} > V_{ju} & j = i \\ V_{ja} \geq V_{ju} & \forall j \neq i \end{cases} \}. \tag{2.2}$$

This property is a necessary condition for maximizing the social cost function.

Additionally, we would like the allocation rule to be *fair*, not preferring one guest over another [WJC$^+$10]. An *ex-post fair* auction—fair even after the allocation was made—is better than an *ex-ante fair* auction, which is fair by expectation value, but may be unfair once a random choice is made in the auction.

The payment rule defines how the payment unit-price, p', is calculated, conforming with the *exclusion compensation* principle, by which the player compensates society for its existence. A mechanism will be considered an affine maximizer only if its payment rule is of the general form:

$$p'_i(a) = \frac{1}{q'_i} \{ h_i(V_{-i}) - \sum_{j \neq i} \frac{\omega_j}{\omega_i} V_j(a) + \frac{1}{\omega_i} c_a \} \quad , \tag{2.3}$$

where $h_i(V_{-i})$ is an arbitrary function that does not depend on V_i, commonly extracted using the Clarke pivot rule, which is solving the same optimization problem without bid i. The result condition on this type of payment is that the payment unit-price is limited by the the player's bid unit-price $0 \leq p'_i \leq p_i$, and thus the player is guaranteed not to pay a higher unit-price than its proposal.

The player's *utility* is defined by the difference between its valuation of the memory and the amount it is charged:

$$U_i = V_i(q'_i) - p'_i q'_i \quad . \tag{2.4}$$

2.1.2 Memory Waste

Since guest valuations change over time, auctions must expire and allow resources to be put up for auction again. Repeated bandwidth auctions (*rounds*) can be analyzed as stand-alone auctions because the benefit from increased bandwidth is immediate. In contrast, the benefit from winning more memory is not immediate.

Memory is often used for caching. To utilize increased cache sizes, guests need to retain the memory used for caches for long-term use, to increase the likelihood of cache hits.

Rapid exchanges are repeating allocation patterns involving guest and host behavior [BCI+07], where resources are transferred back and forth between guests. If subsequent memory auctions result in rapid exchanges, then increasing auction frequency will yield less benefit for guests; memory they rented but did not yet have time to use is *wasted*. Hence, unlike in bandwidth auctions, memory auctions should not be analyzed separately. Instead the auctioneer should control the amount of memory exchanging hands in each auction round to balance memory waste with the time required to respond to changing guest valuations.

We would like the MPSP mechanism to be able to control the amount of exchanged memory in the system, preventing changes that would not have significant influence on the social cost and allowing necessary ones.

2.1.3 The Linear Maximizer

In the conventional bandwidth auction the bidder bids with a (p, q) tuple, p is the unit price it is willing to pay, and q is the quantity it is willing to buy. This bidding language results in a linear valuation function: $V_i(a) = p_i \cdot a_i$.

The subset A' defines the valid allocations, by which the allocation of bid i must agree with the quantity it is willing to by: $a_i \leq q_i$, and the sum of all allocations must be limited by the amount of memory being proposed for auction, $q_{\max}$: $\sum_i a_i \leq q_{\max}$.

The affine maximizer equation (2.1) can be seen as an optimization problem, and can be simplified to the form:

$$\begin{aligned} q' = &\operatorname*{argmax}_a \sum_i p_i a_i \\ \text{s.t.} \quad & a_i \le q_i \quad \forall i \\ & \sum_i a_i \le q_{\max} \quad . \end{aligned} \tag{2.5}$$

We refer to this optimization problem as the *linear maximizer.* The payment formulation for the linear maximizer described above is:

$$p'_i(a) = \sum_{j \neq i} p_j (q''_{ij} - q'_j) \quad , \tag{2.6}$$

where q''_i is the solution of the linear maximizer problem without bid i.

Incentive Compatibility of the Linear Maximizer

The linear maximizer bidding language limitations prevent players from revealing their whole valuation function. Thus, in the linear maximizer, as in Lazar and Semret's PSP auction [LS99] and in Maillé and Tuffin's multi bid auction [MT04a], the bids cannot be defined as truthful, in the classic game-theoretical sense.

Lazar and Semret claim that the bids in the PSP auction are truthful and the mechanism is incentive compatible, despite the linear bidding language. Furthermore, Maillé and Tuffin define that in their limited "multi-bid" bidding language , a list of (p, q) tuples, a truthful bid is when the bid prices equal to the player's marginal valuations. Accordingly, we define that a truthful bid is when the player reveals a point on its valuation function, as follows:

$$(p_i, q_i) \text{ is truthful} \quad \Leftrightarrow \quad p_i \cdot q_i = V_i(q_i) \quad . \tag{2.7}$$

As described in [LS99], due to the exclusion compensation principle, by which the payment is calculated, the player's best interest is to submit a bid with values representing its true valuation. The player would not be interested in increasing or decreasing the unit price for a desired amount of goods, and would not request a larger or smaller quantity of goods for a specific unit price.

The client's interest is to submit a bid that would maximize its utility, but with a high enough unit-price such that it would also have a good change to be accepted to the auction. It might consider bids with higher utility versus bids with higher unit-price.

Implementation of the Linear Maximizer

The optimization problem for the linear maximizer presented in equation (2.5), can be solved by the simple algorithm presented in algorithm 2.1.

Algorithm 2.1 Allocation for the highest social cost of the linear maximizer mechanism

```
Sort the bids by decreasing $p_i$.
$rem \leftarrow q_{\max}$
for i = 1 ... n do
    $q'_i \leftarrow \min\{q_i, rem\}$
    $rem \leftarrow rem - q'_i$
end for
```

This algorithm returns the set q' which maximizes the linear maximizer equation and satisfies the conditions. It runs in complexity of $O(n \log n)$, where n is the number of bids, because of the guests sorting.

The payment for each bid, presented in equation (2.6), is calculated by solving the same problem again, for each bid, in complexity of $O(n^2 \log n)$.

The total complexity of the algorithm is $O(n^2 \log n)$.

Tie Breaking

Guests are sorted by the unit-prices they bid when they queue for memory. When two or more bids are identical, the tie must be broken, preferably fairly and Pareto-efficiently. In Lazar and Semret's bandwidth auction [LS99], PSP, bids of this kind were rejected from the auction, Leaving the players responsible for not bidding the same value. Since our auction is a *sealed bid auction*, in which a player does not see the other players' bid, we cannot allow such a rule.

A *steady state* is when the auction's personal results (a guest's won goods and payment) turn out the same in subsequent auctions in response to the same strategy. A *Nash equilibrium* is a steady state in which guests stick to their bids if they know what other guests plan to bid. Breaking ties by excluding guests prevents ties in Nash equilibrium. However, in dynamic, real-life scenarios, guest bids are not always in Nash equilibrium, especially if guests do not continuously inter-communicate. Hence, we sought alternatives to this tie-breaking method, which we find unsuitable for memory auctions.

We considered three Pareto-efficient options. The first was to divide the memory among all the tied guests [AM85], but this approach is also NP-hard, because the forbidden ranges may turn solving it into solving a knapsack problem. The second option was to prefer guests according to a random shuffle which is *ex-ante fair* before each round. The third option was to prefer the current memory holder [Wan09], which is only ex-ante fair before the tie is formed, but is the most efficient tie breaker.

We opted for combining the latter two approaches in the MPSP auction. The tie breaking procedure prefers guests whose previous allocation, $q'_i(t-1)$, was higher. This kind of tie breaking was set in order to reduce the costly memory exchange in the system. If the previous allocations were equal to each other, $q'_i(t-1) = q'_j(t-1)$, we set the preference to be random, which grants the algorithm the fairness property. In

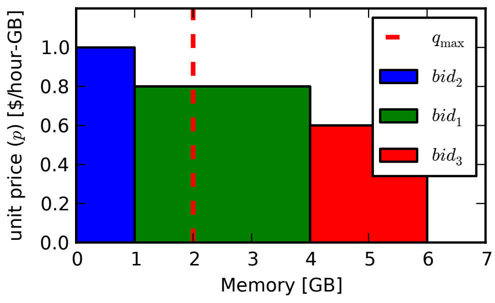
Practice, it does not matter in this case which guest will be allocated.

Example

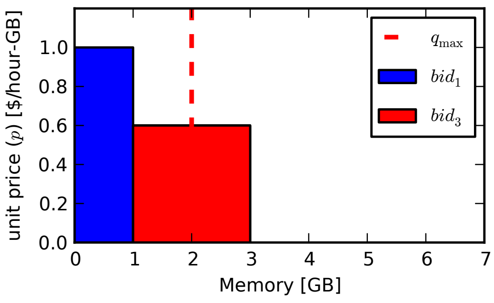
Let's consider a case, as presented in figure 2.1, in which the received bids are $b_1 = (0.8, 3)$, $b_2 = (1, 1)$, $b_3 = (0.6, 2)$, and the memory for auction is $q_{\max} = 2$. The bids will be sorted by descending p values: $\{b_2, b_1, b_3\}$, and will be allocated in order until the auction memory runs out. As can be seen in figure 2.1(a), the allocation for the bidders will be: $q' = \{1, 1, 0\}$. Let's examine the payment for bid b_1. The allocation is repeated again, this time with bids $\{b_2, b_3\}$, as can be seen in figure 2.1(b), and the result will be $q''_1 = \{1, 0, 1\}$, (setting 0 for b_1). The payment according to the payment rule will be:

$$p'_1 = \frac{1}{q'_1}\{p_2[q''_{12} - q'_2] + p_3[q''_{13} - q'_3]\} = \frac{1}{1}\{1[1-1] + 0.6[1-0]\} = 0.6 \quad .$$

It can be seen that indeed we got $0 \le p'_1 \le p_1$.



(a) Allocation



(b) Payment

Figure 2.1: Example for linear maximizer algorithm. In figure 2.1(a) the allocation rule is shown. We can see the bids sorted by descending p vaule, and allocated by order with the memory for auction, $q_{\max}$. In figure 2.1(b) the payment calculation for bid 1 is shown.

2.1.4 Handling Valuation Functions

The valuation function V_i are the player's valuation of memory, and is handled by the MPSP algorithm as an arbitrary function. Nevertheless, it is important to see the mechanism through the player's point of view, and address its concerns for different types of valuation functions.

The Player's Memory Valuation

The memory valuation function, which also describes how much the guest is willing to pay for different memory quantities, is a composition of two functions:

$$V(m, l) = V_p(P(m, l)) \quad .$$

The function $P(m, l)$ describes the performance the guest can achieve given a certain load and memory quantity. Performance is a guest-specific metric that differs between guests. It might be measured in hits per second for a web server, transactions per second for a database, trades per second for a high-frequency-trading system, or any other guest-specific metric.

The function $V_p(P)$ is the guest's valuation of performance function. It describes the benefit that the client derives from a given level of performance from a given guest. This function is different for each client and is private information of that client.

The linear maximizer provides a solution for players who are willing to bid with the presented (p_i, q_i) tuple. Let's consider the guest's utility function, presented in equation (2.4), since the algorithm allows allocation of a player with a lower allocation than its full allocation, $q_i' \leq q_i$, and since the payment unit-price can be equal to the proposed unit price, $p_i' = p_i$, the utility will be positive only when the player valuation agrees with:

$$V_i(a_i) \geq p_i a_i \ , \ \ \forall a_i \leq q_i \ \ . \tag{2.8}$$

If this condition fails, and the player is not fully allocated, the player's utility becomes negative, which means that it is loosing.

This condition restricts the valuation function to be concave and monotonically rising, a restriction that the players would not accept. Even if the performance memory function, $P(q)$, is concave and monotonically rising, which is a reasonable but not always true assumption, (for example, a fixed heap application, is reasonable case for non-concave performance function), the valuation function can easily violate this restriction (for example, when $V(P) \propto P^2$).

Allowed Ranges

Auction protocols which assume monotonically rising concave valuation functions either interpret a bid of unit price and quantity (p, q) as willingness to purchase exactly q units of memory for unit price p or as willingness to buy up to q units at price p. In the first case, the bidding language is limited to exact quantities. In the second case, if the valuation function is non-concave, the guest may get a quantity that is smaller than the one it bid for, and pay for it a unit price it is not willing to pay. If the function is not, at the very least, monotonically rising, it may even get a quantity it would be better off without.

The PSP auction [LS99] optimally allocates a divisible resource if and only if all the valuation functions are monotonically rising and concave. If at least one guest function is not monotonically rising, or not concave, bidding the true valuation of the requested memory quantity is no longer the player's best interest, and the mechanism is no longe incentive compatible. Additionally, the chosen allocation does not necessarily maximize the guests' social welfare.

We propose a generalization of the bidding language, to handle non-concave or even non-monotonically rising valuation functions, by forbidding memory ranges in the submitted bid, b_i,

$$b_i = (p_i, r_{i,1}, q_{i,1}, \ldots, r_{i,m}, q_{i,m}) \enspace , \tag{2.9}$$

where (r_{ij}, q_{ij}) is the j^{th} allowed range of player i. The original bid can be transformed to the general form as follows: $(p_i, q_i) \to (p_i, 0, q_i)$. The first allowed range, in which $j = 0$, is added to the bid that is received from the player, and represents the zero allocation acceptance range, $r_{i,0}, q_{i,0} \equiv 0$.

By using this bidding language we allow the player to forbid allocations where $V_i(a_i) < p_i a_i$, and avoid negative utility. Additionally, we keep the linear bidding language, with its advantages. This new bidding language changes only the affine maximizer subset of possible allocations, A', and thus doesn't hurt the incentive compatibility property. The subset of possible allocations becomes:

$$\begin{gathered} \exists j \mid a_i \in [r_{ij}, q_{ij}] \quad \forall i \\ \sum_i a_i \le q_{\max} \enspace . \end{gathered} \tag{2.10}$$

The payments are calculated as in the linear maximizer solution. For each player i, we determine the highest social cost allocation without the player's participation in the auction, q_i'', and use equation (2.6) to determine its payment unit-price. Since all we have done is to modify the subset of possible allocations, A', all the described properties of the payment remain.

Implementation with Allowed Ranges

The algorithm for finding the best allocation for the linear maximizer can be applied to the modified bidding language, as long as the last allocation, that might not be equal to the full allocation, $a_i < q_{i,m_i}$, falls in an allowed range, $a_i \in [r_{ij}, q_{ij}]$. The algorithm for the allocation is shown in algorithm 2.2. The q_0 variable is the preallocation vector, used by the recursive algorithm. Those values are allocated to the guests regardless of the allocation rule. If they sum up to a higher value than the auctioned memory, the procedure returns a vector of zero allocation for all bidders, and thus it will not be chosen.

The allocation algorithm either returns a valid allocation, or information about an invalid allocation. In the case of a valid allocation, no farther search is required, and the node becomes a leaf in the search tree. The allocation is invalid when the last invalid allocation, a_i, of player i, who refer to as the *borderline guest*, falls in the forbidden region: $a_i \in [q_{i,j-1}, r_{i,j}]$. In this case the node becomes a branch, and the search for the highest social cost function is forwarded to two children. Those children can be seen as splitting the bid i at the memory value of r_{ij}, as follows (see also figure 2.2):

1. **Preallocation**. The preallocation of bid i, $q_{0,i}$, is increased by r_{ij} (it is initiated

Algorithm 2.2 Allocation for the highest social cost of the linear maximizer with allowed ranges mechanism

```
function ALLOC($p, r, q, q_{\max}, q_0$)
    Sort the bids by decreasing $p_i$.
    $r \leftarrow \{0\} + r$                                        ▷ Add the zero range
    $q \leftarrow \{0\} + q$
    $rem \leftarrow q_{\max} - \sum_i q_{0,i}$
    if $rem < 0$ then
        return $\{0\}$
    end if
    for i = 1 ... n do
        $a_i \leftarrow \min\{q_{i,n}, rem\}$
        if $\nexists j \mid a_i \in [r_{ij}, q_{ij}]$ then
            return $(i, \text{argmin}_{j \in [1,m_i], r_{ij} \geq rem}\{r_{ij}\})$
        end if
        $rem \leftarrow rem - a_i$
        $a_i \leftarrow a_i + q_{0,i}$
    end for
    return $a$
end function
```

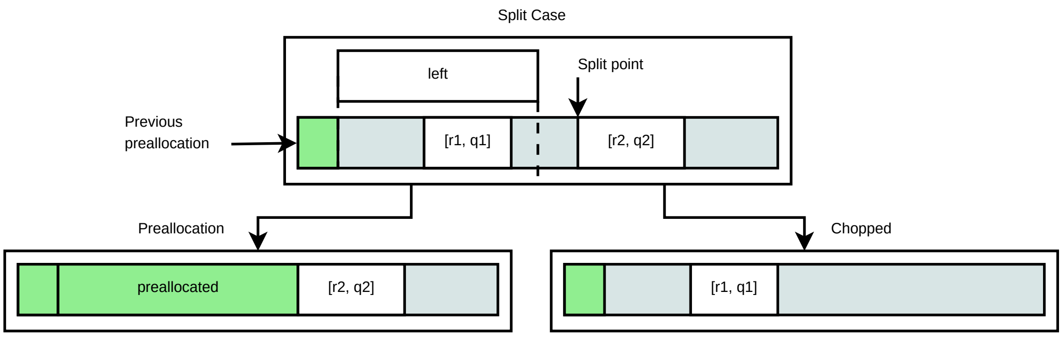


Figure 2.2: A diagram showing the 2-tree split of the recursive algorithm. If the remaining memory to be allocated is between two allowed ranges, two options are considered. One is to preallocate the minimal value of the allowed range above, and the other is to discard all above ranges.

with 0 in the root node). The bid is then converted to have only the ranges above and with respect to r_{ij}: $b_i \leftarrow (p_i, r_{i,j} - r_{i,j}, q_{i,j} - r_{i,j}, \ldots, r_{i,m_i} - r_{i,j}, q_{i,m_i} - r_{i,j})$.

2. **Trimming**. The bid remains only with the memory ranges below the split point, r_{ij}: $b_i \leftarrow (p_i, r_{i1}, q_{i1}, \ldots, r_{i,j-1}, q_{i,j-1})$.

This branching changes the way we treat the borderline guest's j^{th} forbidden range. Before the branching, we first computed the allocation, and only then verified this forbidden range was not violated. After the branching, this forbidden range is an explicit constraint. Each branched node returns the allocation with the higher social welfare value among its children. The recursion algorithm is described in algorithm 2.3.

Algorithm 2.3 A recursive algorithm that finds the allocation of highest social cost in a 2-tree, in which invalid allocation split the search in preallocated bid and in trimmed bid. The first call for the function is with $q_{0,i} = 0$ and $\text{sc}_{\min} = 0$

```
function REC_ALLOC($p, r, q, q_{\max}, q_0, \text{sc}_{\min}$)
    $results \leftarrow alloc(bids, q_{\max}, q_0)$
    if $results$ is not a split then
        $a \leftarrow results$
        $\text{sc} \leftarrow \sum_i p_i a_i$
        return $a, \text{sc}$
    end if
    $i, j \leftarrow results$
    $R_i \leftarrow r_i, Q_i \leftarrow q_i, Q_0 \leftarrow q_0$
                                                        ▷ Preallocation
    $r_i \leftarrow R_{ij} - R_{ij}, \ldots, R_{im} - R_{ij}$
    $q_i \leftarrow Q_{ij} - R_{ij}, \ldots, Q_{im} - R_{ij}$
    $q_{0i} \leftarrow Q_{0i} + R_{ij}$
    $a_p, \text{sc}_p \leftarrow rec\_alloc(p, r, q, q_0, q_{\max}, \text{sc}_{\min})$
                                                        ▷ Trimming
    $r_i \leftarrow R_{i,1}, \ldots, R_{i,j-1}$
    $q_i \leftarrow Q_{i,1}, \ldots, Q_{i,j-1}$
    $q_{0i} \leftarrow Q_{0i}$
    $a_t, \text{sc}_t \leftarrow rec\_alloc(p, r, q, q_0, q_{\max}, \max\{\text{sc}_p, \text{sc}_{\min}\})$
                                                        ▷ Return the better allocation
    if $\text{sc}_p \geq \text{sc}_t$ then
        return $a_p, \text{sc}_p$
    else
        return $a_t, \text{sc}_t$
    end if
end function
```

Correctness

Since the MPSP auction with the forbidden ranges belongs to the affine maximizer's social cost functions, it is has the property of incentive compatibility. All that remains to show is that the recursive algorithm finds the allocation which maximizes the social cost function, given in equation (2.5), subject to the conditions in equation (2.10).

The algorithm builds a 2-tree: every node in the tree is either a leaf or a branch with two children. We will show that scanning the tree is sufficient to find a globally optimal allocation. Every branch node divides the optimization space to two disjoint cases of potentially valid allocations. Together with the invalid case, in which a guest gets a quantity within its forbidden range, the unification of these cases is exactly the full range of values the borderline guest can have. If a globally optimal allocation belongs in this node, it must be in at least one of its children. Hence, scanning the full tree will find the optimal allocation.

We can additionally prove that it is unnecessary to branch in an invalid node with a sc value that is lower than a known valid sc value. We define a *pseudo-divisible allocation* as an allocation of the entire memory quantity Q to guests G according to their order of valuations (unit price), regardless of forbidden ranges. This allocation sc value is at least as high as that of any other more constrained allocation of Q to G (constraints can only lower the sc value). Hence, if the pseudo-divisible allocation is valid, it can be set as a leaf, because more constrained allocations cannot yield a higher sc value. If it is invalid, and its sc value is lower than the value of the best previous allocation, it can be dismissed.

Complexity

The MPSP algorithm with the forbidden ranges solves an NP-hard problem. Its time complexity is $O(n^2 \cdot 2^m)$, where n is the number of guests and m is the number of all the forbidden ranges in all the bids. To find an optimal allocation, at most 2^m divisible allocations are attempted, each taking $O(n)$ to compute. The sorting is $O(n \log n)$, but it is done only once, hence does not influence the overall complexity. For the payment calculation, $O(n)$ allocations need to be computed.

However, for real life performance functions, a few forbidden ranges are enough to cover the non-concave regions, additionally this number could be limited by the auction rules. Given the small number of guests on a physical machine, the algorithm's run-time is reasonable (less than one second using a single hardware thread in our experiments). For concave functions, the complexity is reduced to $O(n^2)$, as in the PSP auction [LS99]. Therefore, the MPSP auction enjoys almost the same computational efficiency and exactly the same Pareto-efficiency of results as the PSP algorithm, while incentivizing quests to bid their true valuations for the memory quantities they bid for, even for guests with forbidden ranges.

Example

Let's consider a case, as presented in figure 2.3, in which the received bids are $b_1 = (1, 0, 2)$, $b_2 = (0.6, 0, 3, 5, 6)$, $b_3 = (0.4, 0, 2)$, and the memory for auction is $q_{\max} = 6$. The bids will be sorted by descending p and remain in the same order, and will be allocated in that order until the auction memory runs out. As can be seen in figure 2.3(a), bid 1 will get a full allocation of 2 GB, and the remaining 4 GB fall in the forbidden range of bid 2: $[3, 5]$. In this case we split b_2, with split information of $(i, j) = (2, 2)$, and search within the two children:

- **Preallocation**. As can be seen in figure 2.3(b), bid 2 gets a preallocation of 5 GB. We can see that the bid with the highest p, bid 1, is only allocated after the preallocation of bid 2. We can also see that bid 2 is now reduced to $b_2 = (0.6, 5 - 5, 6 - 5) = (0.6, 0, 1)$. In this case, the allocation, $a = \{1, 5, 0\}$, is valid, and no further search is required. The social cost of that allocation is $\mathrm{sc}_p = 1 \cdot 1 + 5 \cdot 0.6 = 4$

- **Trimming**. As can be seen in figure 2.3(c), bid 2 is reduced to the ranges below the split location: $b_2 = (0.6, 0, 3)$. In this case, the allocation, $a = \{2, 3, 1\}$, is valid, and no further search is required. The social cost of that allocation is $\mathrm{sc}_t = 2 \cdot 1 + 3 \cdot 0.6 + 1 \cdot 0.4 = 4.2$

The highest social cost of the 2-tree search is the trimming allocation, and thus this would be the chosen one.

2.1.5 Memory Exchange Penalty

Memory, unlike other resources, cannot be time shared between guests in a virtual machine, and exchanging it rapidly has its price (see section 2.1.2). We exploit the general definition of the affine maximizer, as presented in equation (2.1), in order to reduce the exchange of memory.

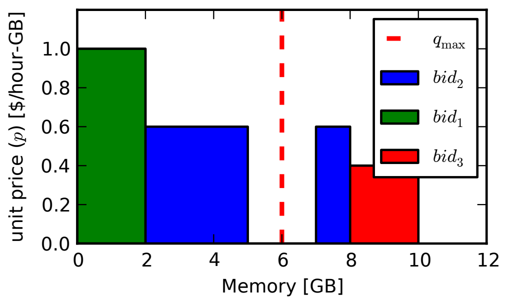
We suggest to define the arbitrary constant, c_a, as a penalty term which is proportional to the amount of memory exchange from the last allocation, and reduces the social cost for higher memory exchange values:

$$c_a = -p_0 \cdot \sum_i \max\{0, a_i - q'_i(t-1)\} \ . \tag{2.11}$$

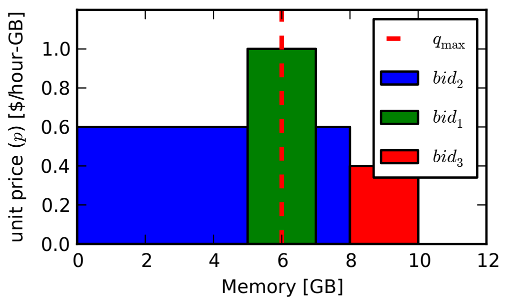
When inserting equation (2.11) to the affine maximizer formula presented in equation (2.1), we obtain:

$$q' = \operatorname*{argmax}_a \{\sum_i p_i a_i - p_0 \cdot \sum_i \max[0, a_i - q'_i(t-1)]\} \ . \tag{2.12}$$

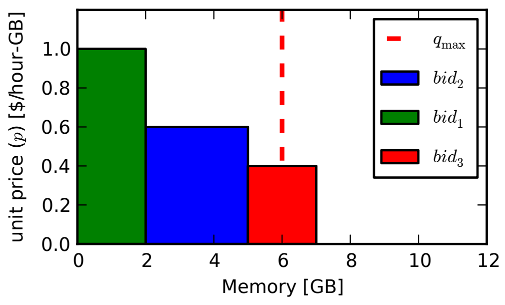
We can manipulate this equation, such that the penalty term will be seen as splitting the bid i, in $q'_i(t-1)$, where the part with ranges above the split point will have a unit



(a) Branch



(b) Preallocation



(c) Trimmed

Figure 2.3: Example for MPSP with allowed ranges. In figure 2.3(a) the bids are shown, arranged by descending p value, where bid 2 has one forbidden range. In figure 2.3(b) we can see how bid 2 gets preallocated with 5 GB, only after the preallocation, the bid with the highest p value is placed. In figure 2.3(c) we can see how bid 2 gets trimmed, as it only has its lower allowed range.

price which is reduced by p_0, as follows:

$$\begin{aligned} q' = \underset{a}{\operatorname{argmax}} \sum_i \{ & p_i \cdot \min[a_i, q'_i(t-1)] + \\ & (p_i - p_0) \cdot \max[0, a_i - q'_i(t-1)]\} \ . \end{aligned} \tag{2.13}$$

The summation is done over i; each guest contributes to the social cost function the expression which is inside the summation. We have shown in equation (2.13) that the expression can be split into two arguments. Let's consider each one of them and show how to convert the bid into two — one parent and one child — bids.

1. **Parent**. The first argument, $p_i \cdot \min[a_i, q'_i(t-1)]$, corresponds to a bid with a unit price of p_i and allocation that is limited by $q'_i(t-1)$. This limitation is equivalent to the case that the bid contained memory ranges up to the $q'_i(t-1)$. Thus, the argument can be replaced with $p_i \cdot a_i^-$, if the bid was defined by:

$$\begin{aligned} r_i^- &= \{r_i \mid r_i \le q'_i(t-1)\} \\ q_i^- &= \{q_i \mid q_i \le q'_i(t-1)\} \cup \{q'_i(t-1)\} \\ b_i^- &= (p_i, r_{i1}^-, q_{i1}^-, \ldots, r_{i,m_i^-}^-, q_{i,m_i^-}^-) \ . \end{aligned} \tag{2.14}$$

   The maximal allocation for b_i^- is $q_{i,m_i^-}^- \le q'_i(t-1)$. Thus, we ensure that $a_i^- = \min[a_i, q'_i(t-1)]$, and have the following equivalence: $p_i \cdot \min[a_i, q'_i(t-1)] \equiv p_i^- \cdot a_i^-$.

2. **Child**. The second argument, $(p_i - p_0) \cdot \max[0, a_i - q'_i(t-1)]$, corresponds to a bid with a unit price of $p_i - p0$ and allocations that is not negative, and additionally, has allocations that are lowered by $q'_i(t-1)$. If we convert bid i, to a bid with a unit price lowered by p_0, and memory ranges decreased by $q'_i(t-1)$, and trim the decreased above zero, as follows:

$$\begin{aligned} r_i^+ &= \{q'_i(t-1)\} \cup \{r_i \mid r_i \ge q'_i(t-1)\} \\ q_i^+ &= \{q_i \mid q_i \ge q'_i(t-1)\} \\ b_i^+ &= (p_i - p_0, r_{i1}^+, q_{i1}^+, \ldots, r_{i,m_i^+}^+, q_{i,m_i^+}^+) \ , \end{aligned} \tag{2.15}$$

   we will achieve the following equivalence: $(p_i - p_0) \cdot \max[0, a_i - q'_i(t-1)] \equiv p_i^+ \cdot a_i^+$.

Implementation of Memory Exchange Penalty

In order to use the memory exchange penalty term, and still employ the MPSP 2-tree search algorithm, as presented before, we employ a method of converting bid i's contribution to the social cost function, $p_i \cdot \min[a_i, q'_i(t-1)] + (p_i - p_0) \cdot \max[0, a_i - q'_i(t-1)]$, into a social cost of two bids, as presented in equations (2.14) and (2.15).

By splitting all the bids, and taking the set of all bids, $b = b^- \cup b^+$, we can see that equation (2.13) with the conditions of the allowed ranges in equation (2.10), and an additional condition to keep the parent-child relation, is reduced to:

$$\begin{aligned}
& q' = \underset{a}{\operatorname{argmax}} \sum_i p_i a_i \\
& \text{s.t.} \\
& \exists j \mid a_i \in [r_{ij}, q_{ij}] \quad \forall i \\
& \sum_i a_i \leq q_{\max} \\
& a_i^- \neq q^-_{m_i^-} \Rightarrow a_i^+ = 0 \ .
\end{aligned} \tag{2.16}$$

Note, from the last condition, that the parent-child relation must be maintained, since a child can be allocated only if the parent was fully allocated.

When encountering an invalid allocation, the search is split into a preallocation of a bid and trimming of a bid. Then, we must reconsider the branching rules:

1. **Preallocation**. The preallocation rule is valid both for a parent and a child, and we treat them as regular bids.

2. **Trimming**. Trimming a child bid is valid. Nevertheless, when a parent is being trimmed, its child must be removed (or, more easily, converted to an empty bid), to validate the additional last condition in equation (2.16).

Determining p_0

The penalty parameter, p_0, is very important, It determines how restrictive the system will be, and it can influence the social welfare or even freeze the memory state. Making it a constant number is not practical, because players can change their valuation during the game, and the order of magnitude of the currently played bids can be changed. Additionally, it cannot depend on the current valuations, since it will exclude the social cost function from the affine maximizer's family.

We proposed to make the penalty parameter proportional to the minimal accepted bid of the last auction. This value represents the bids that were relevant in the last round. It is limited by the maximal accepted bid from above and zero from below.

Correctness and Complexity

The only difference between the new problem and the one solved in the allowed ranges implementation is the additional condition of the parent-child relation. We have shown in section 2.1.2 that the 2-tree recursive algorithm searches within all possible allocations of m bids that can maximize the social cost function.

It is clear that without the parent-child condition, the same algorithm searches for all possible allocations of the new $2 \cdot n$ bids. Adding the new condition might invalidate some of the allocations that are found in the 2-tree recursive algorithm. The ranges do not change, and thus the algorithm will search for all valid possible allocations. Additional search options will be invalid because of the addition of the parent-child relation rule, as presented in equation (2.16). When the search branches, the new trimming rule will ensure that trimming a parent will produce a valid allocation.

The complexity of the new algorithm is the same, since the number of allowed ranges, m, remained the same, and the number of players was multiplied by two, $2 \cdot n$.

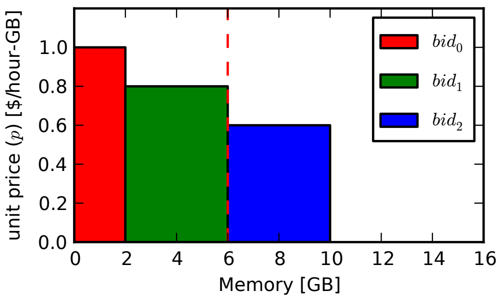
Example

Let's consider a case, as presented in figure 2.4, in which the received bids are $b_1 = (1, 0, 2)$, $b_2 = (0.8, 0, 4)$, $b_3 = (0.6, 0, 4)$, the memory for auction is $q_{\max} = 6$, the allocation for the last round is $q'(t-1) = \{1, 2, 3\}$, and the penalty constant is $p_0 = 0.3$.

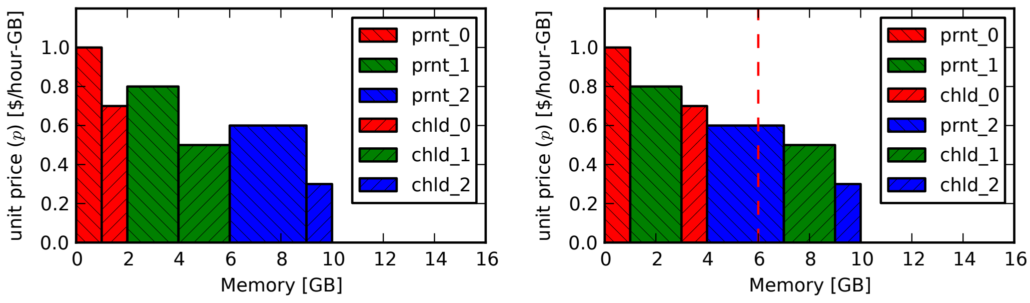
We can see in figure 2.4(a) that without the penalty term, as presented in the linear maximizer algorithm, the resulting allocation would have been: $q' = \{2, 4, 0\}$.

In figure 2.4(b) we see the result of splitting the bids according to the last allocated memory and the penalty constant. Each bid has a parent bid, the one with the higher p value, and a child bid. The sorted new set of bids is shown in figure 2.4(c), and it can be seen that the allocation this time is: $q' = \{2, 2, 2\}$.

Without this algorithm, 3 GB are moved from player 3, and with it only 1 GB are moved, thus demonstrating the algorithm's effectiveness.



(a) Original



(b) Parent-child split (c) Sort and allocation

Figure 2.4: Example for MPSP with memory exchange penalty. In figure 2.4(a) the original bids are shown. In figure 2.4(b) we can see the bids, after splitting each bid into a parent and a child. In figure 2.4(c) we can see the new bids sorted to be allocated, the result allocation is different from the allocation of the original bids, thus memory exchange reduction is demonstrated.

2.1.6 Alternative Researched Mechanisms

In additional to the memory exchange penalty term, two other techniques for reducing the memory exchange were researched. The first is the reclaim factor, and the second is conditional allocation.

Reclaim Factor

In this mechanism, the guest initial state in each round depends on its won extra memory in the previous round, and depends on the decay constant, $0 < \alpha \leq 1$, we refer to as *reclaim-factor*. In each round, the auctioneer reclaims α of each guest's extra memory for a new auction,

$$\text{base}_i(t) = \alpha \cdot m_{0i} + (1 - \alpha) \cdot m_i(t-1) \ \ ,$$

and the memory allocation is the won auction memory on top of the base memory and not on top of the bare memory, as presented in the MPSP auction,

$$m_i(t) = \text{base}_i + q'_i(t) \ \ .$$

The guest continues to rent the rest of the extra memory it won in previous auctions at the prices for which it won it. The host can change the reclaim factor between auctions. It can increase it to improve the system's responsiveness when the memory pressure rises or is expected to rise (e.g., a new guest is launched), or when guests change bids fast, indicating fast valuation changes. Otherwise, it can decrease it to decrease the potential memory waste.

Accounting becomes more complicated when using the reclaim factor. In each round, a guest may win a memory *chunk*: a memory quantity with an attached rental unit-price. Over time, guests come to hold memory chunks of different sizes with different unit prices. The host holds this information as a list, sorted by unit price. The list is updated at the end of the auction round in two stages: first, α of the guest's extra memory is released (the cheapest chunks or parts thereof). Then, if the guest won memory quantity q'_i in the auction, a memory chunk of size q'_i, with a unit price of p'_i is added to the list.

The purpose behind of this mechanism is to maintain a steady state in the system. In practice, a lot of noise was created, and the system behavior was not clear due to its influence. Eventually we preferred to shelve the idea.

Conditional Allocation

Conditional allocation is a rough and quick patch that was examined before the use of the penalty term, in order to delay change of memory allocation. In this technique, two constants were defined; C_{sc}, which is the percentage by which the current social cost must be higher than the previous social cost, and C_p is the maximal change in the guest unit price values below which a preservation of the previous allocation is allowed. With those constant, we defined the three following conditions:

$$\begin{cases} \mathrm{sc}(t-1) > \mathrm{sc}(t) \cdot (1 - C_{\mathrm{sc}}) \\ |p_i(t-1) - p_i(t)| < C_p \ \ \forall i \\ q_{\max}(t-1) = q_{\max}(t) \end{cases},$$

where t is the current auction round, $t-1$ is the previous auction round. The previous social welfare is calculated according to an allocation that agrees with the current allowed ranges and according to the current unit price. If the all of the conditions pass, the previous allocation remains according to the current allowed memory range, and the bills are recalculated according to the current unit prices.

This technique gave good results when truthful players were playing in the auction. Nevertheless, using it discards the incentive compatibility property of the mechanism, and truthful bids can no longer be assumed.

2.2 System Overview

The developed system, described in figure 2.5, is composed of a cloud server, running a number of virtual machines (VMs) that can be referred to as guests. The server OS, referred to as the host or hypervisor, also runs a process called Ginseng. This process manages memory allocation to the guests according to the results of periodic auctions in which the guests can participate in order to win extra memory, on top of their basic predefined allocation.

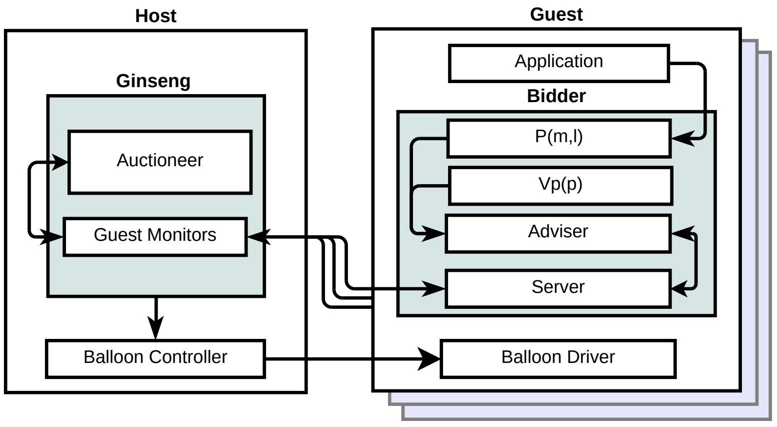


Figure 2.5: Ginseng system architecture diagram. Ginseng runs on the host, and communicates through TCP/IP with an agent running inside the guests, and controlling the balloon driver inside the guest. This agent controls the guest's memory. The agent monitors the tested application, in order to serve bids which will maximize its utility.

2.3 Host Side Design

Ginseng runs as a process on the host OS and auctions memory between guests by periodically running an MSPS auction algorithm. It uses the MPSP *incentive compatibility* property, by which the guest's best interest is to bid the true valuation of the memory. and calculate the allocation for highest social cost accordingly. Ginseng announce a new auction, collects the bids from the guests, and notifies the guests of the auction's results. It communicates with the guests via the TCP/IP communication protocol described in section section 2.3.2. The auction flow and rules are described in sections 2.1 and 2.3.1 respectively.

Ginseng uses balloons [Lit11] to change the memory allocation of guests according to the auction's results. However, it does not specifically depend on balloons, but only requires that the host support some underlying mechanism for memory borrowing. Ginseng was implemented for cloud hosts running the KVM hypervisor [KKL+07].

2.3.1 The Auction Flow

In the MPSP auction, memory allocations change every round. The guest rents the memory for the full duration of one round. Here we describe one MPSP auction round, indexed t.

Time constants

Three time scales are involved in the usability of memory borrowing and therefore the limits to the experiments we conducted: the typical time that passes before the change in physical memory begins to affect performance, T_{mem}; the time between auction rounds, (or other decision making), T_{auction}; a typical time scale in which conditions (e.g., load) change, T_{load}. Useful memory borrowing requires $T_{\text{load}} >> T_{\text{mem}}$. This condition is also necessary for on-line learning of memory valuation. To evaluate T_{mem}, we performed large step tests, making abrupt sizable changes in the physical memory and measuring the time it took the performance to stabilize.

Each auction period is a predefined value, T_{auction}, measured in seconds. In realistic setups providers should set $T_{\text{auction}} < T_{\text{load}}$. Therefore, we set T_{auction} to 12 seconds. In each 12-second auction round the host waited 3 seconds for guest bids and then spent 1 second computing the auction's result and notifying the guests. The guests were then allowed 8 seconds to prepare in case they lost memory.

Initialization

Not all of the guest's memory is achieved by winning an auction: each guest is guaranteed to be allocated with a minimal, predefined, memory amount denoted as m_0. This approach is similar to QClouds' [NKG10] Q_0, which is the minimal QoS a guest is ensured to have, and that Ginkgo [HGS$^+$11] ensures minimal SLA. The difference is that Ginseng provide basic resources and does not interfere with the guests' performance.

In Ginseng, this m_0 memory is used by the guest as the basic memory for the OS and minimal memory for the application. The host must demand high prices on this memory, so the guest will ask for the real minimum memory that it requires. In section 5.1.4 we discuss the host revenue from the m_0 memory, and from the *extra memory*, which is the amount of memory on top of the m_0 memory.

Auction Announcement

Each round, Ginseng announces a new auction. But first it needs to calculate the initial conditions of the auction round. Ginseng computes the *auction memory*, $q_{\max}$, the total memory that will be proposed for auction, and is the maximal amount of memory each guest can bid for:

$$q_{\max} = m_{\text{total}} - \sum_i m_{0i} - m_{\text{host}} \quad ,$$

where m_{total} is the total amount of physical memory in the system, and m_{host} is the amount of memory the host reserves for its own use.

Ginseng informs each guest i on the upcoming auction. It sends the m_{0i} memory, the auction memory, $q_{\max}$, the auction's *closing time*, after which bids are ignored, and the auction round, which is used as a unique ID to clarify the context of corresponding bid and notification messages.

Bid Collection

Interested guests bid for memory. Agent i's *bid* is composed of a *unit price* p_i—price per MB per hour (billing is still done per second according to exact rental duration) and a list of *desired ranges*: mutually exclusive, closed ranges of desired memory quantities $[r_{ij}, q_{ij}]$ for $j = 1 \dots m_i$, sorted in ascending order, such that $r_{i,1} \leq q_{i,1} \leq \dots \leq r_{i,m_i} \leq q_{i,m_i})$. The bid means that the guest is willing to rent any memory quantity within the desired range list, in addition to its current basic holdings $\text{base}_i(t)$, for a unit price p_i.

We refer to the memory ranges between the desired ranges as *forbidden ranges*: $[q_{i,j-1}, r_{ij}]$ for $j = 1 \dots m_i$. Desired and forbidden ranges are shown in figure 2.6.

Collection The host asynchronously collects guest bids. It considers the most recent bid from each guest. Since the guests are committed to send the bid with the corresponding auction ID, only those with the correct one are taken into account in the following bid processing stage. Guests that did not bid lose the auction automatically. A guest that persists in not bidding gradually loses its extra memory, until it is left with its m_0 memory.

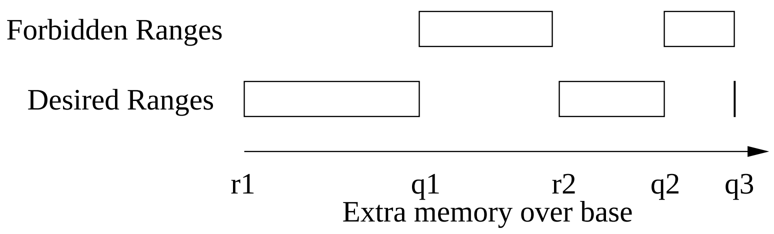


Figure 2.6: A bidding language example for strategy $(p, [r_1, q_1], [r_2, q_2], [r_3, q_3])$. The first forbidden range does not exist $(r_1 = 0)$. The last desired range is only a point $(r_3 = q_3)$.

Processing

Allocation and Payments The host computes the allocation and payments according to the MPSP auction protocol described in section 2.1. For each guest i, it computes how much memory it won, q'_i, and at what unit price, p'_i.

Notification

The host informs each guest i of its personal results p'_i, q'_i. To improve the performance of guest learning algorithms, to be described in section 2.4.4, the host also announces information that guests can work out anyhow, about *borderline bids*: the lowest accepted bid's unit-price and the highest rejected bid's unit-price. It also notifies the guest if another guest submitted a bid with the same unit-price. By doing so, Ginseng helps the guest to avoid tie situations. The last data in the notification is the time of the upcoming memory change.

Ginseng gives the guests an *adjustment period* before it changes their memory allocation. The purpose of this period is to allow each guest's agent to notify its applications of the upcoming memory changes, and then allow the applications time to gracefully reduce their use of memory, if necessary. The applications are free to choose when to start reducing their memory consumption, according to their memory-release agility. This early notification approach makes it possible for the guest operating systems to gracefully tolerate sudden large memory changes and spares applications the need to monitor second-hand information on memory pressure.

Adjusting and Moving Memory

After an adjustment period following the announcement, the host actually takes memory from those who lost it and gives it to those who won.

2.3.2 Auction API

As part of Ginseng, a client-server protocol is implemented. Since Ginseng was based on MOM (see section 2.5.1), in which the host is the client and the guests run the server, Ginseng is the client and constantly tries to connect to a port in the guest. If a connection is created, the auction messages are transferred:

1. **Welcome message**

   When a guest starts running on a Ginseng host, Ginseng tries to communicate with it by sending it a welcome message to a defined port. We discovered that the balloon driver causes an *allocation difference*, ΔM, in which the guest sees a lower memory allocation (the MemTotal field in `/proc/meminfo` system file) than the host allocated it. We also discovered that this allocation difference is constant as long as the guest lives. Moreover, If the maximal memory amounts vary, so will the differences in allocation: larger maximal memory results in a larger allocation difference—around 200 to 300 MB for a maximal memory of 10000 MB and around 50 MB for maximal memory of 2000 MB.

   The guest must deal with this kind of problem. Thus, in the welcome message, the host sends the guest the current memory allocation, the guest uses this data to calculate the allocation difference and respond to the host with the new desired bare memory. From this point on, the guest knows it has to request ΔM more memory than it needs, and when the host notifies the guests of an upcoming memory allocation it will get ΔM less.

2. **Announcement**

   Ginseng sends an announce message to the guest, announcing a new memory auction:

   - Auction round: the auction round number, also used as the auction ID.
   - Auction memory, $q_{\max}(t)$: the amount of memory for auction in the following round.
   - Bare memory, $m_{0i}(t)$: per-guest information that reminds the guest of its bare memory.
   - Closing time: the time that the auction will be closed for new or updated bids.

   The guests should respond to this message with a reply containing their bidding information:

   - Auction round: the round number, which is used as an ID for the auction that the bid is related to, in order to prevent any misunderstanding between the host and the guest.
   - Unit price: the amount of money that the guest is willing to pay for one MB of extra memory per hour.

- Bid ranges: a list of tuples representing the desired extra memory values in ranges that the guest is willing to get.

3. **Notification**

   After all the bids are received and the results calculated, Ginseng notifies the guests of the results of the auction. The notification message contains the following fields:

   - Auction round: the auction ID that the notification is related to.
   - Bill: the amount of money that the guest was billed.
   - Memory, $m_i(t)$: the total memory that will be allocated to the guest,

   $$m_i(t) = m_{0i} + q'_i(t) \quad .$$

   - Unit price: the unit price that the guest paid for the extra memory it won.
   - Tie: a boolean that indicates whether the guest asked for the same unit price as another participating guest.
   - Actuation time: when Ginseng will actuate the memory controller and apply the memory change; gives the guest an opportunity to prepare for a reduction in memory.
   - Minimal accepted unit price: the minimal unit price bid in the auction for which any extra memory was won.
   - Maximal rejected unit price: the maximal unit price that did not win any extra memory.

   The last three fields are only used to help the guest improve its bid, and are not necessary.

2.4 Guest Side Design

Each guest runs an application that is supposed to produce the maximal utility. This application is monitored and controlled with an application tier, which we call the *bidder*(see section 2.4.1). The bidder also sends bids to and receives information from Ginseng according to the auction API (see section 2.3.2).

The bidder has several entities that help it bid, as shown in figure 2.7: the *adviser* (see section 2.4.2), which calculates the best bid according to the given state; the *profiler* (see section 2.4.3), which predicts the program performance according to the desired memory (the function $P(l, m)$); and the *estimator* (see section 2.4.4), which estimates the unit price that the won memory will cost ($p'(q')$).

2.4.1 Bidder

The bidder queries the application for its current state. In our case it asks for the load. It also queries the OS for the memory state. It then passes to the adviser the information of the application, information from Ginseng on the upcoming auction and the last auction results. After the adviser calculates the bid, the bidder sends the bid back to Ginseng's auctioneer. A scheme of the bidder's interactions can be seen in figure 2.7.

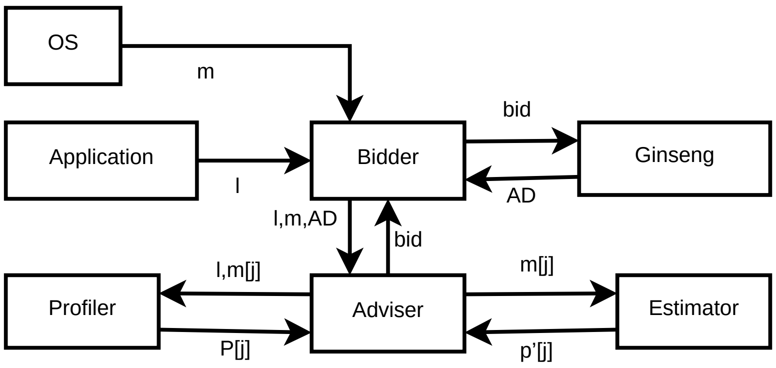


Figure 2.7: A scheme of the Bidder's interactions, with the OS, application and Adviser. l denotes load state; m is memory state; $m[i]$ is an array of desired memory states; $P[i]$ is an array of predicted performance correlated to the memory states; $p'[i]$ is an array of estimated unit prices; AD is auction data

2.4.2 Adviser

Our implementation is a simple adviser that bids a truthful bid with the highest chances to win the auction. Since the adviser was not the main concern of our research, it is relatively simple, but can be made more sophisticated if need be, by incorporating elements of machine learning and signal processing to make better bids and improve the system performance.

Performance and Valuation Function

The guest must decide on a performance-valuation relation, similar to the implementation on the host side in Qclouds [NKG10] where the guests define "Q-states", a discrete relationship between the QoS and the guest's willingness-to-pay. For example, if the performance P is measured by transactions per second, and any transaction might be evaluated as 20\$, then the valuation function is $V(P) = 20\$ \cdot P$

The Adviser Algorithm

The adviser calculates a bid according to a given guest state, which contains a prediction for the average load, l, the base memory, m, and the amount of memory offered for auction $q_{\max}$ (see section 2.3.1).

The adviser first creates a vector of q values, representing memories above the base memory, as follows:

$$q_i = (i+1) \cdot \Delta q \ , \quad \max_i\{q_i\} \leq q_{\max} \ ,$$

where Δq is predefined value determines the advising resolution.

The valuation V_i for each q_i can be calculated with the profiler performance interpolation function $P(l,m)$, and the valuation as function of performance $V(P)$:

$$V_i = V(P(l, m+q_i)) \ .$$

The unit price p_i can be calculated using the no bid valuation $V_0 = V(P(l,m))$ as the slope between the $q=0$ point to the bidding point on the valuation function (see example in figure 2.8):

$$p_i = \frac{V_i - V_0}{q_i} \ .$$

The utility U_i is calculated using the estimated unit price from the estimator, p'_i, as follows:

$$U_i = \max\{V_i - p'_i \cdot q_i, 0\} \ .$$

A target unit price p_{tar} is calculated by the average of the last minimal accepted unit prices in the last 10 auctions. Those values are part of the auction API. They are stored by the adviser when received from the bidder as part of the auction data. The adviser then calculates the best possible unit price, p^*, by the following three steps:

1. Keep only points where $p_i \geq p_{tar}$, because those have the best chances to win the next auctions. If none of the points agree with that condition, the adviser chooses the highest value available:

$$p^* = \max_i \{p_i\} \ .$$

2. Choose the highest utility points the guest can have:

$$U^* = \{j \mid U_j = \max_i\{U_i\}\} \ .$$

3. If there is more than one point left after the previous filtering, the adviser chooses the highest unit price that was left, in order to give the guest the best chance to win the next auction. The utilities of all the points are the same and the unit price will, in any case, not be defined by the chosen unit price of the guest, but by the unit prices that were rejected in the auction:

$$p^* = \max_{i \in U^*}\{p_i\} \quad .$$

The chosen unit price p^* is then used to calculate the allowed ranges. The allowed ranges are all the q values where $p_i \geq p^*$, and they are calculated by interpolating between adjusted p_i values to create a list of tuples representing the ranges. The adviser returns the p^*, and the allowed ranges to the bidder.

2.4.3 Profiler

The profiler is a mathematical unit, whose purpose is to take the data collected offline with the testbed process, and convert it to performance as a continuous and smooth function of load and memory. Once the profiler is loaded, it processes the raw testbed data into the $P(l, m)$ function in order to be used by the *adviser*.

The data is loaded into a matrix P_{ij} and two vectors, l_i and m_j. The profiler is also given a *valuation* function $V(P)$. When the adviser asks the profiler for the valuation of a point (l, m), the profiler interpolates the performance 2D function using a bivariant spline approximation (with predefined knot values). If the point is higher than the maximal known load or memory, saturation is assumed. The calculated performance value is then translated to valuation using the defined valuation function $V(P)$ and returned to the adviser.

2.4.4 Estimator

The estimator is used to estimate the estimated unit-price as a function of memory, $p'(q)$, from previous auction results. The estimator is configured with an array of estimation memory points, q_i, and data validity maximal time, $a_{\max}$. A list of bill lists, b_i, is used to save history data for memory range closest to q_i. Each data item holds the bill and the round.

History Maintenance

After each round r, the auction results that are saved in the estimator are the won memory q, and the won bill b. Then, a bill, b_{ij}, is added to the bi list:

$$b_{ij} = \frac{q \cdot b}{q_i} \mid q \in \left[\frac{q_{i-1} + q_i}{2}, \frac{q_i + q_{i+1}}{2}\right) \quad ,$$

which gives an approximated bill to the q_i point.

Estimation

The estimation is done for a bid point q, and round r. The estimator first erases all data whose age is smaller than the allowed predefined age, in the group: $\{b_{ij} \mid r_{ij} < r - a_{\max}\}$.

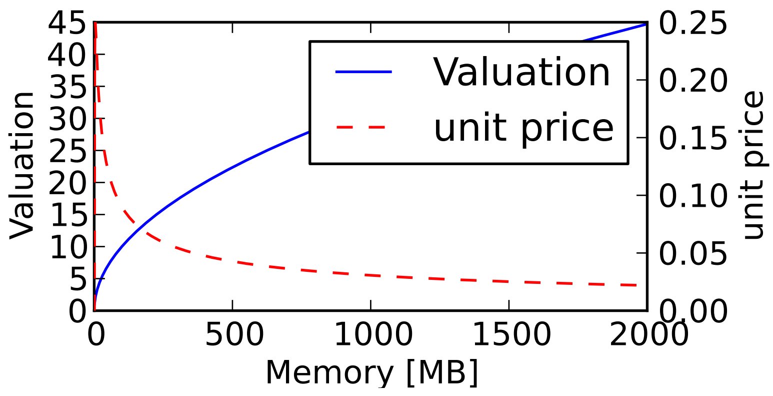


Figure 2.8: An example of valuation, and the corresponding unit price function.

Then an estimation of the bill, b'_i, for each bid point q_i is calculated according to a weight which is relative to the inverse of the age:

$$w_{ij} = \frac{1}{r - r_{ij}} \quad ,$$

$$b'_i = \frac{\sum_j w_{ij} \cdot b_{ij}}{\sum_j w_{ij}} \quad .$$

The memory weighting is calculated according to the distance from the bid value q as follows. Different weighting functions can be chosen, such that the closest q_i to q, the higher its weight. We chose the following function:

$$u_i = \begin{cases} \frac{1}{|q_i - q|} & \text{if } q \neq q_i \\ 1 & \text{otherwise} \end{cases} \quad .$$

The following bounds are then estimated:

- Upper bound:

$$\bar{B} = \sum_i \{\frac{u_i \cdot b'_i}{\min\{q_i, q\}}\} / \sum_i u_i \quad .$$

- Upper bound by extra p:

$$\bar{B_{p'}} = \sum_i \left\{ u_i \cdot \left(\frac{b'_i}{q} + \frac{q - q_i}{q} \cdot \frac{b'_i - b'_{i-1}}{q_i - q_{i-1}} \right) \right\} / \sum_i u_i \quad .$$

- Lower bound:

$$\underline{B} = \sum_i \left\{ \frac{u_i \cdot b'_i}{\max\{q_i, q\}} \right\} / \sum_i u_i \quad .$$

Then, the unit price estimation is calculated by averaging the lower bound and minimal upper bound, taking into account $p_{\min}$, the maximal rejected unit price of the last auction, which is part of the auction API:

$$p' = \frac{\min\{\underline{B}, p_{\min}\} + \min\{\bar{B}, \bar{B_{p'}}, p_{\min}\}}{2} \quad .$$

This evaluation gives an estimation of the unit price such that data which is older or whose q_i is farther from q will have less influence on the estimated value.

2.5 Implementation

In this section, the implementation of system is described, demonstrating how the described rules and flow were inserted into the MOM architecture. Implementation is described both for the host side, which implements the auctioneer, and the guest side, which implements the bidder.

2.5.1 MOM Architecture

Ginseng's code is based on MOM [Lit11]. In the MOM implementation, the controlling system is composed of three main components called guest manager, policy and controllers.

- **Guest Manager**. The guest manager is responsible for collecting updated statistical data. It attaches a thread called the guest monitor to each guest, which is alive as long as the guest is running. The guest manager periodically queries the guest monitors for updated data. Each guest monitor then invokes a set of collectors which collect data on the guest. This information might be accessible to a process running on the host, or received by communicating with a cooperative agent running inside the guest.

- **Policy**. The policy is invoked periodically, and determines control parameters by inspecting the data of the guest manager, in order to achieve better performance. The policy is an abstraction of a unit that makes decision which will be implemented in the nearby future, according to data that was collected in the recent past.

- **Controllers**. The controllers implement the control decisions that the policy determined.

2.5.2 Auctioneer - Host-side Policy Implementation

Ginseng has uses MOM as basic control mechanism in order to enable the MPSP auction on the host by defining its own set of collectors, set of controllers, and its own policy.

Collectors

Collectors asynchronously collect data on the guests and host in order to make policy decisions. Ginseng, which is a black box system, doesn't need any information on the running guest, or the host.

Collectors were implemented only for experiment purposes. Information about the host memory, page faults and CPU were collected by querying the `/proc/meminfo`, `/proc/vmstat` and `/proc/stat` and more data on the guest internals was collected by communicating with the agent, known as bidder (see section 2.4.1), which is already running on the guest.

- **Memory-stat collector**

  The memory data is collected by querying `/proc/meminfo` and `/proc/vmstat` system files, and calculating the following values:

  – The total available memory, which is written in the `MemTotal` row in `/proc/meminfo`.

  – The unused memory, which is written in the `MemFree` row in `/proc/meminfo`.

  – The cached pages and buffers, which are written in the `Buffers` and `Cached` rows in `/proc/meminfo`.

  – The total free memory, which is calculated as the sum of unused, cached and buffers.

  – Minor and major page faults, which are calculated by subtracting two adjacent samplings of the `pgfault` and `pgmajfault` rows, respectively, in `/proc/vmstat`.

  – Swap in and out, which are calculated by subtracting two adjacent samplings of the `pswpin` and `pswpout` rows, respectively, in `/proc/vmstat` system file.

- **CPU usage collector**

  CPU usage is obtained from the system file `/proc/stat`. In that file, there is a line for each running CPU and one line for the average of all CPUs. In each line there are 7 values, measured in jiffies, corresponding to: (1) user mode processing time, (2) nice user mode processing time, (3) kernel mode processing time, (4) idle time, (5) time spent waiting for I/O to complete, (6) time spent servicing IRQ interrupt and (7) time spent servicing soft IRQ interrupts. The sum of all those numbers is the total CPU time. For each line, two values were calculated:

  – The CPU usage, calculated by subtracting the sum of items 1 to 3 in two adjacent samples and dividing by the subtraction of two samplings of the total CPU time.

  – The I/O percent, calculated by subtracting item 5 in two adjacent samples and dividing by the subtraction of two samplings of the total CPU time.

The collected data is used to make controlling decisions. Those decisions are made by the policy, which is introduced in the following subsection.

Policy

The Ginseng algorithm is implemented in the policy stage of the MOM model. The algorithm consists of three main stages:

1. **Announcing stage**. In this stage, the *announcer* calculates the guest base memory and the memory for auction and sends the data to the guests. It is also responsible for collecting the bids from the guests.
2. **Auction stage**. The auctioneer processes the bids from the guests according to the MPSP algorithm and calculates the allocations and payments for the auction period.
3. **Notification stage** In this stage, the notifier (see section 2.5.2) notifies the guests of the auction results.

Announcer

The announcer announces an auction according to the Ginseng auction API: the auction round number (an identifier), the amount of memory to be sold in the announced auction, and for each guest, its next base memory allocation. For convenience the host also declares the reclaim factor used in the base memory calculation. Then, the auctioneer waits a defined period for the bids from the guests and then closes the auction bid submission.

Auctioneer

Here the allocation and payment calculation is described, according to the MPSP algorithm, described in section 2.1.

The array, `bids`, represents the guests' bids, such that `bids[i].p` is the unit price that guest i is willing to pay, p_i, and `bids[i].qrs` points to a list of tuples of the allowed ranges of guest i: `bids[i].qrs[j]` is the range $(r, q)_{ij}$, which defines the range $[r_{ij}, q_{ij}]$.

For each guest i, the unit price is checked for correctness: $p_i \geq 0$, and the correctness of the allowed ranges: $r_{ij} \leq q_{ij}$, $q_{ij} \leq r_{i,j+1}$, $r_{ij} \geq 0$. If the bid fails the check, a default bid is given: $p_i = 0$, $(r, q)_i = [(0, 0)]$.

As described in section 2.1.2, the bids are then shuffled, sorted descending by the last allocation of each corresponding guest, and then sorted by descending p_i value. The last sort is the most significant.

The allocation and bills are calculated using the main function
`calc_alloc_bills(bids, q_max)`. It gets the bids list, `bids`, and the memory for

auction, `q_max`, as its input. The highest sc allocation vector, m_i, according to the allocation rule, is then calculated using the function: `alloc(bids, q_max)`, which will be discussed later. The bill vector, bill_i, is calculated for each guest i as follows: If $m_i = 0$, then $\text{bill}_i = 0$. Otherwise, the highest sc allocation must be calculated without guest i, and with the same auction memory: `alloc(bids[`$\setminus\{i\}$`], q_max)`. The bill is then calculated according to the payment rule, as the sum of valuations the other guests didn't get because guest i won the memory m_i.

The function `alloc` returns the output of a recursive function `rec_alloc(bids, q_max, prealloc, SW_min)`, and its purpose is to return the allocation with the highest sc. The `bids` and `q_max` are the received parameters, the `prealloc` is a recursive parameter, referring to an array of an amount of memory that will be initially allocated to the guest, initiated to zeros. `SW_min` is also a recursive parameter indicating the maximal sc that was already calculated, and it is a lower bound to limit the recursive calls.

The function uses another function, `_alloc(bids, total, prealloc)`, described in algorithm 2.2 which initially allocates all the guests with the memory in `prealloc` and then goes over the guests (sorted as mentioned before) and allocates for each guest i the maximal $\max_j \{q_{ij}\}$ value it requested. If the remaining memory, rem, is not enough, that is, $\text{rem} < \max_j\{q_{ij}\}$, it allocates the remaining memory only if it is within any allowed range $\text{rem} \in [r_{ij}, q_{ij}]$. If the remaining memory for allocation is not the maximal request or not in an allowed range, the recursion splits into a 2-tree and considers two optional routes, as shown in figure 2.2. The *split point*, which is the amount of memory in which we split guest i, is defined as:

$$m_{\text{split}} = \min_j\{r_{ij} \geq \text{rem}\} \quad .$$

.

If no split information was received after calculating the allocation with `_alloc`, the allocation is valid, has the highest sc, and can be returned. If split information was received, the recursion continues in two different directions:

- Preallocation: The split guest i is given (an additional) preallocation of the split point m_{split}. Because of that preallocation, r_{ij} must be subtracted from its bid ranges and the ranges below r_{ij} vanish.

- Trimming: The split guest i remains with the allowed ranges below and equal to the split point.

The recursion parameter, `sc_min`, is updated every time an allocation is calculated. In each node, if the sc of the first allocation is lower than the already known `sc_min`, the recursion returns and doesn't split the tree. The recursion returns the allocation with the highest sc value.

Notifier

The notifier sends the auction results back to the client, according to the Ginseng auction API. The notifier sends the auction round as an identifier for the results, the allocated memory, the bill, the unit price of the won memory and some more information about the last auction: a boolean tie parameter (indicates whether the guest had the same unit price as another guest), the minimal accepted unit price, and the maximal rejected unit price.

Controllers

The balloon actuator allocates the new memory to the guests according to the MPSP auction policy decisions.

2.5.3 Bidder - Guest-Side Implementation

Guests utilize an economic learning agent to rent more or less physical memory. Each guest's agent acts on its behalf according to its valuation-of-memory function within the framework of the MPSP protocol. The guest is free to use any agent it wishes provided it speaks Ginseng's API.

The bidder is also based on MOM's controlling abstraction. It is also invokes a set of collectors in order to know the current state of the application and OS. Periodically an advising policy is invoked and determines the needed resources and allowable unit price that will improve the performance or utility. By running a server, a guest can receive data about the current auction rules and previous auction results, and submit its bid for the following auction according to the adviser's decisions.

Collectors

The bidder employs several collectors, all of which, other than the load collector are for experimental purposes.

- **Load collector**. The load collector queries the application for current load status. This is the only collected data that is used by the adviser to make the bid decision.
- **Memory statistics collector**. The memory statistics collector is the same one used in the host side (see section 2.5.2).
- **CPU usage collector** The CPU usage collector is the same one used in the host side (see section 2.5.2)
- **Program CPU usage**. The program CPU usage collector collects CPU usage of specific processes: the bidder and the tested application. The CPU usage information of each process can be found in the `/proc/[pid]/stat` system files, according to the PID of the process. In each file there is a series of numbers, the

14th and 15th values of which correspond to CPU time spent in user mode and in kernel mode respectively, measured in jiffies. Dividing the subtraction of the sum of those two numbers in two adjacent samples, with subtraction of two adjacent samples of the total CPU time (from the CPU usage collector), results in the CPU usage of the process.

Server

The server serves host messages. It updates the bidder data when receiving auction messages from the host's announcer or notifier and sends the bid to the bid-collector, according to the adviser's last suggestion.

2.6 Dynamic Memory Cloud Computer

Existing applications and out-of-the-box OS configurations are not suitable for a dynamic memory cloud computer like the one we have developed. In section 2.6.1, the requirements for *dynamic memory applications* are described. These are applications that can produce maximum performance according to the current available memory of the OS and are able to free memory on demand. In section 2.6.2 an approach of hinting the guest before changing its memory is described and its advantages are explained. A special OS configuration that enables high memory usage without OS interference is described in section 2.6.3. We claim in this section that implementing these approaches in a dynamic cloud computer without memory overcommitment can increase its performance.

2.6.1 Dynamic Memory Applications

In dynamic cloud environments, applications should know how much memory they have, use it when it is available and free it when it is taken. Applications hat do not meet these requirements are not suitable for our system. Here we explain why many existing applications do not meet our requirements. The requirements themselves are detailed below.

In [HZPW09], performance was measured as a function of the memory allocation and it was shown that the performance behaves as a step function of the memory allocation: the performance was negligible up to the point where the heap size was a certain percent of the allocated memory; above that point, the performance rose and remained constant no matter how much memory was allocated. We blame this step function behavior on the static Java heap model. Once the memory allocation was too small to fit the constant Java heap and the OS memory, the OS swapped parts of the heap, and the performance severely decreased. Graphs with similar behavior are also presented in [GHDS$^+$11, HGS$^+$11]

Most programs today will behave similarly since they are not aware of the available memory in the OS. They are not programmed to be used in a dynamic memory cloud computer. Their performance does not improve when the there is more memory available and, when memory is scarce, they do not reduce their memory signature. They trust the OS to manage their memory by swapping. For example, Memcached [Fit09], a widely used key-value caching application, is usually used in front of databases. Memcached comes out of the box with predefined constant cache size. Benchmarking it with a constant cache size of 500 MB inside a guest for different guest RAM values results in massive thrashing when the memory allocation for the guest is below the used memory of the system (figure 4.4), similar to what was measured in [HZPW09].

In order to maintain good performance memory is low by avoiding thrashing and using the memory efficiently, and achieve better performance by making use of high memory, when it is available, we developed the following requirements for dynamic applications:

- The application's performance must be exposed and quantitative, such that the performance as a function of memory can be evaluated.
- The application must know to use different memory quantities. Logic dictates that it should, improve performance by using more memory.
- The application must know to change its memory signature quickly enough. In that way, memory growth will be reflected in a fast increase in performance. Moreover, before memory is reclaimed, the application will be able to free it fast enough to avoid thrashing.
- The application must expose an API for changing its memory signature.

For use in our system, we developed Memory Consumer, a synthetic dynamic memory benchmark, as well as a modified version of Memcached, a widely-used key-value storage cloud caching application.

Memory Consumer

Memory Consumer is a dynamic memory application, designed to give a linear up to a saturation point performance graph, without any cache, network communication or CPU overheads. The application is initiated with three parameters: *saturation memory*, m_{max}, the amount of memory where the Memory Consumer performance graph saturates (the performance stays constant when the memory increases); *spare memory*, m_{spare}, the amount of memory that should remain free, in order to give flexibility to the memory of unmanaged processes; and *wait time*, T_{wait}, the waiting time between two subsequent memory write attempts, used in order to reduce CPU consumption.

While Memory Consumer is running, it tries to write to a random 1 MB sized cell out of a range of $m_{\text{max}} - m_{\text{spare}}$ cells. If the address is within the range of the currently *available memory*, 1 MB of data is actually written to the memory address, and it is considered a hit. After each attempt, whether a hit or a miss, it sleeps for T_{wait} seconds, so that misses cost time. The application's performance is defined as the hit rate. The *available memory* is defined by subtracting m_{spare} from total available memory in the OS. Memory Consumer follows the OS available memory and changes its memory allocation accordingly.

Dynamic Memcached

Dynamic Memcached is a version of Memcached [1] that changes its heap size on the fly to respond to OS available memory changes. We drove the load of Memcached with *memslap* client. The application's performance was defined as the get hit rate, such that an increase in cache size results in higher performance. We programmed Memcached to expose an interface for changing its cache size.

We programmed an additional application tier, initiated with a parameter m_{spare}. The Memcached tier follows the OS available memory and sends orders for Memcached to change its cache size to the amount of the subtraction of m_{spare} from the total available memory in the OS.

2.6.2 Memory Change Explicit Hinting

We introduce an explicit memory allocation change hinting mechanism, which explicitly hints to guest about an upcoming memory allocation change, before the change is implemented.

Waldspurger [Wal02] mentions that the allocation rate should be limited, to avoid stressing the guest OS. In [Wan09] a limit of memory change was set to 20% of the current allocation, and a time dependent rate is not mentioned. The change rate was set this way because pages are not ready to be reclaimed. In [GHDS^{+}11, HGS^{+}11] the memory change rate was limited to 64 MB/second, but no reason is given for doing so. This limitation is most important to the reduction of the guest memory, but not important at all for its enlargement. All of those approaches implicitly hint the guest that the memory is about to change, by slowly reducing its memory and causing it pressure.

In the mempressure control group proposal (see [Vor13]), a mechanism of cooperation between the kernel and applications running inside the control group was proposed, in order to greatly reduce the memory without thrashing. This mechanism helps both the kernel and the application avoid swapping.

We used a similar approach, avoiding the limitation of memory change rate, by explicit hinting the guest with the next round's memory allocation before the memory is actually controlled by the balloons.

[1]Dynamic-Memcached is available from `https://github.com/ladypine/memcached`.

In this way, the program can free the memory before it is taken, and the taken memory is not referenced when it is taken. The memory controller application tier we programmed gets the hint of the upcoming memory value, and also queries the `/proc/meminfo` to get the actual available memory. The desired memory size is then sent to the application as a memory target, for which the application needs to adapt its size.

Memory Controller Application Tier

The memory controller is an application tier that is configured with a parameter ,spare (a typical value of 50 MB), which defines the amount of memory that must be left free.

The memory controller considers the total available memory to be the minimal value between the hinted memory and the actual memory:

$$\text{total} = \min\{\text{hinted}, \text{actual}\} \quad .$$

Then it queries the `/proc/meminfo` for the used memory and calculates the unused memory:

$$\text{unused} = \text{total} - \text{used} \quad .$$

It then defines the difference it has to correct as

$$\text{diff} = \text{unused} - \text{spare} \quad .$$

The application tier calculates two controlling parameters that are sent to the application, which needs to adapt its memory signature accordingly. The first is the current usage of the memory: usage , and the second is the target memory usage:

$$\text{target} = \text{usage} + \text{diff} \quad .$$

This way the controller adjusts the memory signature of the application even if the requested target of memory does not exactly equal the actual memory usage of the application, and the spare memory is always left free.

2.6.3 Linux OS Configuration

In our experiments we used a specially configured OS in the guests. In this configuration the kernel ignores memory pressure, and keeps its cached pages and buffers. The Linux kernel has its own mechanisms to control the memory pressure. Controlling the kernel behavior without hacking it can be done with the aid of `sysctl` system call, which changes the different kernel parameters.

The Linux kernel has memory watermarks[2]. When the free memory in the system

[2]In the kernel there are three watermarks. `watermark[WMARK_MIN]`, `watermark[WMARK_LOW]` and `watermark[WMARK_HIGH]`, will be referred as WM_{min}, WM_{low} and WM_{high} respectively.

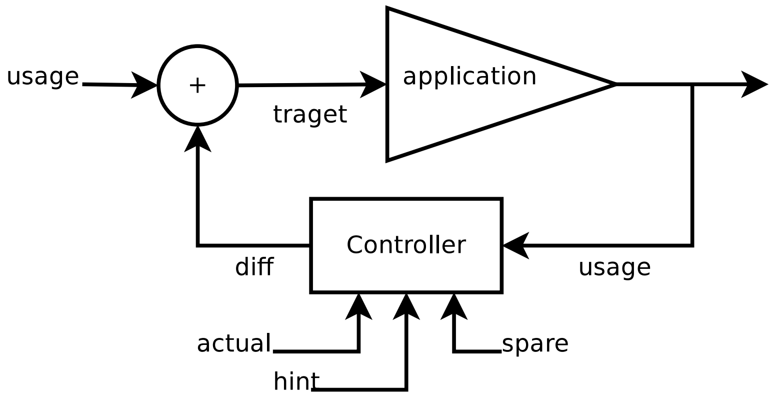


Figure 2.9: The memory controller application tier as a simple feedback control loop. The controller takes the memory usage, the current available memory and the hint from the auction *notifier*, and calculates the difference between the current application's memory signature and the required memory signature.

gets below a watermark, the kernel changes its memory pressure policy. The kernel handles memory pressure by swapping processes' pages, and by reducing its cached pages and buffers. The following kernel parameters affect this behavior:

- `vm.min_free_kbytes` affects the kernel memory watermarks in the low memory zones. In the kernel function `setup_per_zone_wmarks` in `/mm/page_alloc.c`, three watermarks are defined. The minimum watermark, $\mathrm{WM}_{\mathrm{min}}$, is defined by:

  $$\mathrm{WM}_{\mathrm{min}} = \mathrm{pages\_min} \cdot \mathrm{present\_pages}/\mathrm{sum\_lowmem\_pages} \quad .$$

  The minimum watermark defines the low and high watermarks as $\mathrm{WM}_{\mathrm{low}} = 1.25 \cdot \mathrm{WM}_{\mathrm{min}}$ and $\mathrm{WM}_{\mathrm{high}} = 1.5 \cdot \mathrm{WM}_{\mathrm{min}}$ respectively.

  There is no control on the watermarks in high memory zones. They are always 1/1024 of the present pages in the zone, limited from above by 128 pages. But this zone is always empty in 64-bit architectures, which is the one we were using.

  In order to achieve high memory usage without having the kernel experience memory pressure and starting the page freeing algorithm, we wanted to lower all the watermarks as much as possible. By setting `vm.min_free_kbytes = 0`, we made all the watermarks 0 for the low memory zones. This means that the kernel won't start swapping even if we achieve 100% of memory usage.

- `vm.overcommit_memory` affects the memory overcommitment policy of the kernel. By setting it to the constant `OVERCOMMIT_ALWAYS`, the function `__vm_enough_memory` in the kernel file `mm/mmap.c`, will always return 0. As a result, a process will never get the no memory error (`-ENOMEM`) when it tries to allocate memory.

- `vm.swappiness` can be set between 0 and 100, and controls whether the kernel preference between two available options for freeing memory. For higher values the kernel will prefer to swap the memory of the processes to the disk, while for lower values it will prefer to reduce the kernel's buffers and cached pages.

The above configuration allowed us to achieve high memory usage (we left only 50 MB of free memory) without hurting the kernel's buffers and cached pages, without swapping the process to the disk, or even without thrashing. We might note that this is only applicable if the application conforms with the requirements specified in section 2.6.1.

2.7 Hardware Description and OS Configurations

We used a cloud host with 12GB of RAM and two Intel(R) Xeon(R) E5620 CPUs @ 2.40GHz with 12MB LLC. Each CPU has 4 cores with hyper-threading enabled, for a total of 16 hardware threads. The host ran Linux with kernel `2.6.35-31-server #62-Ubuntu`, and the guests ran `3.2.0-29-generic #46-Ubuntu`. We limited the memory in the grub to 10000 MB, in order to create the needed memory overcommiment in the limited number of available CPUs, assuming each guest is assigned with one CPU, and one CPU is reserved for the host.

To reduce measurement noise, we disabled EIST, NUMA, and C-STATE in the BIOS and Kernel Samepage Merging (KSM) [AEW09] in the host kernel. To prevent networking bottlenecks, we increased the network buffers.

2.8 `libvirt` setup

`libvirt` is an abstraction layer for virtualization. It enables definition of virtual machines and virtual networks with XML files. We defined a large enough number of VMs (13) with consecutive and fixed mac addresses, and configured them with the same virtual network. In addition there is option in `libvirt` to define a DHCP server, and attach a fixed IP address for each mac address of each guest. By doing so, the host can know all the IP addresses of the guests and communicate with them with the auction messages.

We also created another *master-machine* which we used only to change the VM configuration, and did not use for experiments. it had a `qcow2` image. Before each experiment, with the `qcow2` technology, we could create new copies of the master image for all the VMs in no time, thus starting the experiment on a new and clean image. Right after that the machine started to run, and an SSH connection was detected, a copy of the updated code was sent from the host to the guest to start the experiment with the updated code. Since we used python, there was no use for compiling the new code.

Chapter 3

Testing Methods

We tested our system with experiments and simulations. Experiments were conducted by running guests on the host cloud computer. Each guest ran the tested application, which was loaded by a load producing program from the host. We conducted three kind of experiments: memory change experiments, in which we tested the reaction of the application to a sudden memory change, testbeds, in which we profiled the performance of the application in different memory allocations and different loads, and benchmarking experiments, in which we compared Ginseng performance to the alternative memory management systems. In the simulations we used the Ginseng code to noiselessly simulate the system with a wide variety of parameters and for a large number of rounds.

The hardware we used for the experiments is described in section 2.7. We used the `libvirt` library to manipulate the KVM virtualization environment. The `libvirt` configuration is described in section 2.8.

3.1 Memory Change Experiments

Memory change experiments were conducted with the goal of finding the configuration of the tested programs that could handle sudden memory changes, specifically, defining the T_{load} and T_{mem} (see section 2.3.1). In these experiments, the application ran inside a guest and was controlled by its memory controller (see section 2.6.2). The application was subjected to a constant load from a load producing program, running on the host. During the experiments, the guest was subjected to two sudden memory changes. First the memory was increased by a certain amount and then it was decreased by the same amount, back to the initial size. The host hinted the guest 5 seconds before each memory change (see section 2.6.2).

We measured the application performance during each experiment. In each conducted experiment, we changed the application and load producing program parameters, in order to find a configuration in which the application met dynamic memory application requirements (see section 2.6.1). In the memory change experiments, each memory change is of 1000 MB. We assume that if the application can handle this amount of change, it can handle all the memory changes in our system.

3.2 Testbeds

In the *testbed* process, the dependency of application performance on a number of parameters was measured. We defined test values for each parameter. Then we measured the performance in all possible combinations of all these values.

The "staging server" is a process in QClouds [NKG10] that creates a static linear relation between QoS and resource allocation using least mean square of the samples. Ginkgo [HGS+11, GHDS+11] also had a profiling stage, in which a non-linear relationship was created between the performance as a function of load and memory allocation.

The testbeds for Memory Consumer and Memcached were similar. The performance measurement was the application hit rate and the parameters were the available memory in the OS and the application load. The application load for Memcached was number of concurrent memslap threads and for Memory Consumer the load was the number of concurrent threads that try to write to the memory.

In the testbed experiment we ran the tested application with the dynamic memory controller inside the guest and the load producing program from the host. We started by allocating the guest with the minimal defined memory amount. Then, we decreased the memory to the maximal value and decreased it back to the minimal value, step by step, according to the defined set of memory values. In each memory step we first loaded the application for a "warm-up", after which we measured the application's performance for the defined set of load values. Each measurement, including the warm-up, was for a fixed, predefined duration. We took care to hint the guest 5 seconds before an upcoming memory change.

The memory range was limited from below by the minimal OS and application memory need and from above, a bit more than the performance saturation: the performance remained constant while the memory was increased. The load range wasn't limited from above by the CPU utilization. Higher loads created unrelated bottlenecks. All limits were found empirically.

To make the testbed represent a high-loaded but not overloaded system, we conducted the testbed process on four guests running the same tested application. The guests were always under the same load and allocated with the same memory amount. We chose four guests because more than that would overcommit the memory on part of the test.

After the testbed, we could produce an average of the performance for each measuring point in the load-memory plane or examine the performance hysteresis due to memory increase or decrease. For a set of memory allocations m_i and loads l_j, we created the performance matrix of the average measured values $P_{ij} = P(l_i, m_j)$. Those vectors were used as a database for the adviser's profiler.

For Memcached and Memory Consumer the measured memory range was between 600 MB and 2400 MB, and the load was between 1 and 10. The measurement duration was 200 seconds for Memcached and 60 seconds for Memory Consumer.

3.3 Benchmarking Experiments

The experiments were conducted in order to compare Ginseng to other cloud memory management alternatives. We defined several experiment sets, each with a different workload. In each set we experimented with varying numbers of guests, and for each number of guests we evaluated all the different memory management alternatives.

In each experiments set, the lowest number of guests was when the cloud server memory was not overcommited. The highest number of guests was when there was no memory for auction in the Ginseng setup. Above this number, Ginseng is just like the static alternative.

Load was defined for Memcached and Memory Consumer as the number of concurrent requests being made. We used two load schemes: *static loads*, where each guest's load is constant over time, and coordinated *dynamic loads*. In coordinated dynamic loads, each pair of guests exchange their loads every T_{load}. The load-exchange timing is not coordinated among the different guest pairs in the experiments. Loads are in the range $[2, 10]$. The total load is always the number of guests $\times 6$, so that the aggregate hit rate of different experiments will be comparable.

Before each experiment, copies of the master guest image are created for each guest (see section 2.8), to create a clean start. The guests are then started, and once an SSH connection is established, the updated code is copied to all the guests.

Thereafter, the tested application is invoked with its dynamic memory controller tier, and, in the case of Ginseng system, a bidder server inside the guest. The host then starts the memory controller process: Ginseng, MOM or none of them. When all of the above are running, the experiment actually starts, and will stop after a defined duration.

In order to apply load to the tested application inside the guests, a thread is created for each guest that repeatedly invokes the load producing program for a duration of T_l, to produce load calculated by averaging the guest's load function in the duration time. The performance of the tested applications were sampled from the load producing program, but it can be sampled by querying the tested application itself. After the experiment is over, all the processes are closed, the guests shut down, and the data saved.

We dedicated hardware thread 0 to the host and pinned the guests to hardware threads $1 \dots N$. The load producing programs were randomly (uniformly) pinned to threads $(N + 1) \dots 15$.

3.3.1 Configurations

The benchmarking experiments are characterized by the benchmarked workload and the guest's valuation of the performance. We defined the guest valuation function as a multiple of a *significance coefficient*, s_i, a scalar value that describes guest i's significance relative to the other guests and a *shape function*, $S(P)$, a function that describes how the guest valuates its performance, as follows:

$$V(P) = s_i \cdot S(P) \ \ .$$

Below we detail the different configurations used in various combinations in the experiments.

Workload

- **Memory Consumer**, configured as in the testbed experiments with a 2000 MB saturation point, and 50 MB spare memory and 0.1 seconds for sleep time between memory write attempts.

- **Memcached**, configured with dynamic allocation. Memslap ran inside the guest, configured as in the testbed experiments with a key size of 249 bytes, a value size of 1024 bytes, a window size of 500 KB and a get/set ratio of 3:7. Each invoking of memslap was for 10 seconds. A network delay of overall 1 millisecond of the guest local loopback was emulated.

Shape function

- **Second order shape function.** This kind of shape function characterizes on-line games and social networks, where the memory requirements are proportional to the number of the users, and the income is proportional to user interactions, which are proportional to the square of the number of users. A combination of this shape function with the Memory Consumer performance graph can be seen in figure 3.1. The shape function is defined as follows:

$$S(P) = P^2 \ \ .$$

- **Piecewise linear shape function.** This kind of shape function characterizes service level agreements that distinguish usage levels by unit price. A combination of this shape function with Memcached performance graph can be seen in figure 3.2. The shape function is defined as follows:

$$S(P) = \begin{cases} 0.001 \cdot P & \text{if } P < P_0 \\ P & \text{otherwise} \end{cases} \ ,$$

where P_0 is the minimal performance required. We set $P_0 = 1.4[Khits/s]$ for Memcached.

- **Performance function.** Used to test an environment where the performance has direct influence on the valuation, or a configuration where all the clients are equal and to compare Ginseng to other cloud memory management systems without the influence of the valuation.

$$S(P) = P$$

  Using this function allows us to compare Ginseng to the alternative cloud memory management systems in an environment where Ginseng does not have its advantage of being aware that different guests have different valuations of performance.

Significant coefficient distributions

- **Pareto distribution.** A distribution of the s_i coefficients according to the economic Pareto function.

- **Three significant guests.** A distribution where three guests are much more significant than the other guests:

$$s_i = \begin{cases} 4000 & \text{if } i = 1 \\ 3000 & \text{if } i = 2 \\ 2000 & \text{if } i = 3 \\ 1 & \text{otherwise.} \end{cases} .$$

Alternative Memory Management Systems

The different memory management systems we evaluated with were:

- **Static**. The entire cloud computer memory, other than a fixed amount reserved for the host, was evenly divided among the guests. In this system the memory is not overcommited.

- **Host-swapping**. More than the available memory was allocated to the guests, leaving the memory to be managed by kernel policies.

- **Memory Overcommitment Manager (MOM)** [Lit11]. A memory overcommitment controller, described in section 1.3. We configured MOM with the memory management policy according to host and guest memory pressure. We didn't use the KSM controlling policy since we disabled the KSM in the OS (see section 2.7).

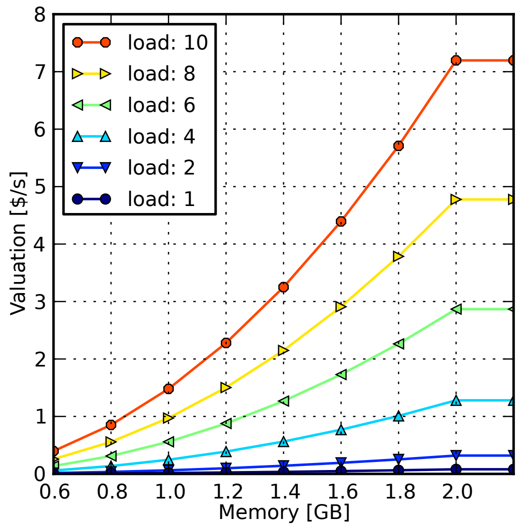


Figure 3.1: The second order shape function combined with the Memory Consumer performance graph, presented as a function of load and memory allocation. This kind of shape function characterizes on-line games and social networks, where the memory requirements are proportional to the number of the users, and the income is proportional to user interactions, which are proportional to the square of the number of users.

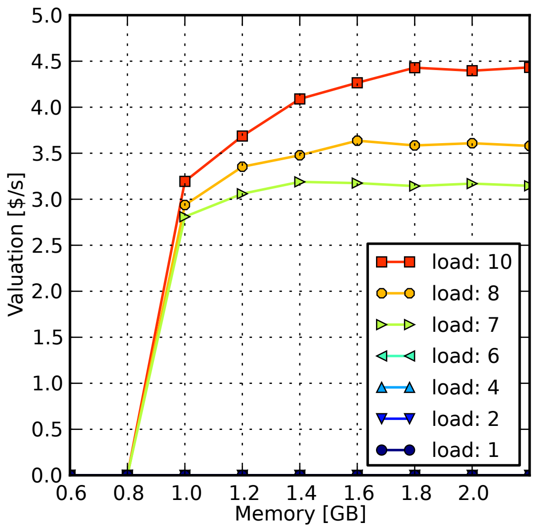


Figure 3.2: The piecewise-linear valuation function combined with the Memcached performance graph, presented as a function of load and memory allocation. This kind of shape function characterizes service level agreements that distinguish usage levels by unit price.

Ginseng and static are memory management systems in which the memory is not overcommited. In those configurations we dedicated 500 MB for the host. In the static system, each guest got $(10000 - 500)/N$ MB, where N denotes the number of guests. In the Ginseng system, each guest's m_0 memory was 600 GB, and the memory sold in the auction was

$$q_{\max} = (10000 - 500 - 600 \cdot N) \ .$$

The host-swapping and MOM systems do overcommit memory. We initially allowed the guests' memory to grow to a limit of 10000 MB (by setting the `libvirt maxmem` parameter). Using these systems in the experiments resulted in massive host swapping (thrashing) and caused the host to freeze. Hence, we reconfigured these systems with memory allocation limit (`libvirt maxmem` parameter) as the performance saturation point, which was 2000 MB for both tested applications. We denote those systems with the prefix "hinted".

In our experiments, 2000 MB is the most memory any rational guest would ask for, since performance remains flat with any additional memory beyond 2000 MB. This white-box configuration, which is based on our knowledge of the experiment design, is intended to get the best performance out of the alternative memory allocation methods. The initial and maximal memory values are summarized in table 3.1.

3.3.2 Simulation Comparison

For each evaluated point, two more measurements were added, describing simulation results for a matching configuration:

1. **Ginseng Simulation**. Simulation of a matching configuration, with the same bidding strategy and the same rules. This simulation shows the theoretical results. The results reveal the upper bounds of the Ginseng system. The difference between the simulated and the measured performance might be due to memory waste (see section 2.1.2), or to accompanying noise from bottlenecks in other resources, such as CPU, cache or network bandwidth. A detailed comparison between the predicted and measured performance is given in 4.3.16.

2. **Upper Bound**. The upper bound is the result of simulations of a matching configuration, in which, instead of allocating memory by a function, an optimization program calculates the allocation that will result in the highest possible social welfare. This upper bound is the highest possible theoretical social welfare that can be achieved in the system[1].

[1] Note that both the optimization program and Ginseng maximize the social welfare. The performance is not maximized and higher performance can be achieved. If the valuation function were set such that $V(P) = P$, the performance of the system would have been maximized

System/Memory [MB]	Initial	Maximal
Ginseng	*bare*	10000
Static	$(10000-500)/N$	—
MOM	*bare*	10000
Host-swapping	10000	—
Hinted MOM	*bare*	2000
Hinted host-swapping	2000	—

Table 3.1: Ginseng and the alternative systems: initial and maximal memory values. N denotes the number of guests.

3.4 Simulations

The simulations evaluate various aspects of Ginseng's performance, and were augmented with experimental results. The simulations reuse Ginseng's algorithmic core with simulated guests that use the same strategy as real guests, and can be seen, therefore, as emulations of the Ginseng process.

The simulations differ from the experiments only with regard to application performance. In the simulations the performance is a function $P(l, m)$, which is derived from the testbed results, and stays constant throughout the simulation, and between simulations. On the other hand, in the experiments the performance is measured on-line from the application and is subjected to degradation from memory waste (see section 2.1.2), or accompanied by noise from bottlenecks in other resources, such as CPU, cache or network bandwidth.

3.4.1 Static Simulations

In the static simulations we simulated the Ginseng system running with 10 guests under constant load for 1000 rounds. Each of the 10 guests had a different load, and thus a different performance function. For all the guests the valuation was defined to be equal to the performance $V(P) = P$, and the *bare* memory was 600 MB, the same value as used in the benchmarking experiments. In the static simulations we simulated all the combinations of reclaim factors in the range of $[0.1, 1]$, with an overcommitment ratio in the range of $[1, 4]$.

In each simulation the *auction memory*, $q_{\max}$, was defined through the overcommitment ratio, OC, as follows:

$$q_{\max} = \left(1 - \frac{1}{\mathrm{OC}}\right) \cdot \sum_i m_{0i} + \frac{1}{\mathrm{OC}} \cdot m_{\max} \quad , \tag{3.1}$$

where $m_{\max}$ is the sum of the guest's memory demand, which is constant. This describes the actual overcommitment of the system, relative to a minimal allowed value of the m_0 memory. Measured values in the static simulations were social welfare, sum of guest utilities, host revenue and an upper bound on waste, ties, and inefficiency.

Waste and Ties. As discussed insection 2.1.2, rapid memory ownership changes leads to degraded performance. This does not appear in the simulation results, because the performance is driven by a function and not by real measurements. We try to quantify this loss by measuring that memory. We defined the *upper bound on memory waste due to ownership changes* as the maximal total allocated memory minus the static allocations over the last 40% of the auction rounds:

$$\text{Waste}(t) = \max_{\tau=t-40}^{t} \sum_{i=1}^{N} m_i(\tau) - \sum_{i=1}^{N} \min_{\tau=t-40}^{t} m_i(\tau) \ .$$

Ties do not cause cycles because when they are broken, preference is given to the previous owner, leading to a stable solution.

Inefficiency. With the aid of the optimization program (presented in section 3.4.2), we were able to find the optimal allocation for maximal social welfare, sc_{max}, in each round. The inefficiency quantifies the aggregate valuation degradation experienced due to the mechanism design and the bidding language, defined as:

$$\text{Inefficiency}(t) = 1 - \frac{\text{sc}}{\text{sc}_{\text{max}}} \ .$$

3.4.2 Optimization Simulations

The optimization simulation is able to run the same configuration of as the regular simulation, static, or dynamic. Instead of running the Ginseng memory allocation algorithm we run an optimization program that finds the allocation for the maximal sc value possible.

The solution is obtained using a binary linear programming method. The extra allocation vector, which is the memory a guest will have above its m_0 memory, defined as:

$$m_j = j \cdot \Delta m$$

where the maximal allocation can be all the auction memory q_{max}, or the saturation memory of the application, to make the solution faster.

The allocation matrix is binary, guest i gets extra allocation m_j if and only if $A_{ij} = 1$. The matrix is defined by:

$$A_{ij} = 1, 0$$

Let us define a helper matrix of possible memory allocation j for guest i:

$$M_{ij} = m_{0i} + m_j$$

The valuation matrix of guest i with extra allocation j is defined as:

$$V_{ij} = V_i \left(A_{ij} \cdot M_{ij} \right)$$

The optimization problem was set to maximize the sum of valuations, limited to give only one allocation to each guest, and to allocate only the available memory:

$$\begin{aligned}
&\max_{A} \sum_{ij} V_{ij} \\
&\text{s.t.} \\
&\sum_{j} A_{ij} = 1 \\
&\sum_{ij} \{A_{ij} \cdot M_{ij}\} \leq q_{\max}
\end{aligned} \tag{3.2}$$

This problem was solved using the python optimization library `pymprog`. The conditions were set with the `st` method, the problem definition was set using the `maximize` method, and the solution achieved using the simple `solve` method.

Chapter 4

Results

In this chapter the results of the different experiments and simulations are presented. Memory Consumer and Memcached were tested for memory change response, profiled in a testbed for their performance dependency on memory and load, and were used to compare Ginseng to the alternative memory management systems (static, hinted host-swapping and hinted-MOM) in benchmarking experiments. Both workloads were also used to simulate the Ginseng system under different parameter sets. We present static and dynamic simulations, and compare them to the optimized solution. In all the results the overcommitment ratio is defined by:

$$\mathrm{OC} = \frac{m_{\mathrm{host}} + \sum_i \left[\Delta M_i + m_i^{\mathrm{sat}}\right]}{m_{\mathrm{total}}} \quad ,$$

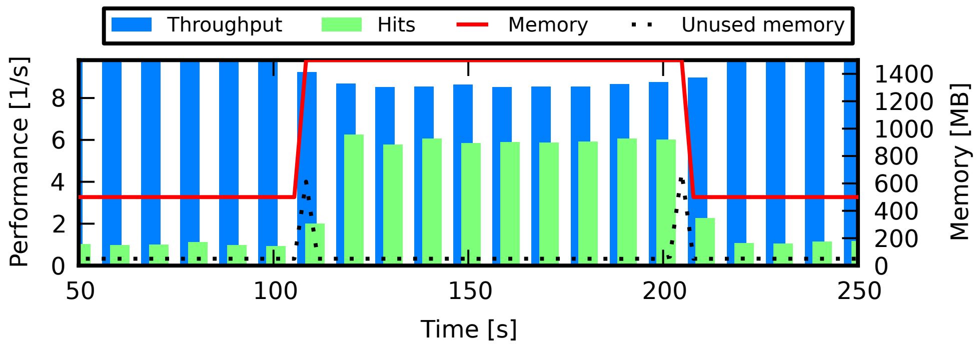
where m_{host} is reserved for the host, ΔM_i is the allocation difference, described in section 2.3.2, m_i^{sat} is the memory in which the guest's performance saturate, its value depends on the running application and the load value and m_{total} is the total memory available. In the experiments we used $m_{\mathrm{host}} = 500$ MB. The allocation difference value depends on the maximal memory of the balloon, we obtained $\Delta M_i = 54$ MB. We limited the total memory of the host to $m_{\mathrm{total}} = 10012$ MB, to have no memory for auction when the guests number reached the number of available CPUs.

4.1 Memory Change Experiments

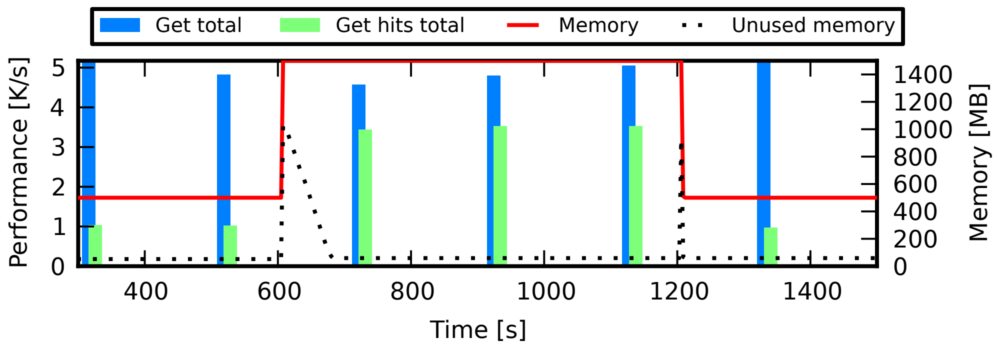
In the memory change experiments, the tested application inside the guest is subjected to constant load while the guest memory allocation is subjected to two sudden changes. Workload configuration was empirically determined to meet the the dynamic memory application requirements, as described in section 2.6.1.

Memory Consumer was configured with $T_{\mathrm{load}} = 10$ seconds. It can be seen in figure 4.1(a), that Memory Consumer performance (hits) was relatively constant when the memory was constant, and when it was higher, the performance increased. In the unused memory graph, we can see from the first peak that it takes Memory Consumer less than 10 seconds to fill the extra 1000 MB, and from the second peak that Memory Consumer succeeded in releasing the memory before it was taken. We can see that the throughput slightly decreased when the memory was high, due to the memory write overhead.

Memcached was configured with $T_{\text{load}} = 200$ seconds. It can be seen in figure 4.1(b) that Memcached performance (hits) was relatively constant when the memory was constant, and when it was higher, the performance increased. In the unused memory graph, we can see from the first peak that it takes Memcached about 100 seconds to fill the extra 1000 MB, and from the second peak that Memcached succeeded in releasing the memory before it was taken.



(a) Memory Consumer



(b) Memcached

Figure 4.1: Memory Consumer (figure 4.1(a)) and Memcached (figure 4.1(b)) performance measurement under constant load and sudden changes in memory allocation. The hits and throughput were measured in order to understand the application performance. The unused memory was measured in order to see the dynamic memory application response to memory change.

4.2 Testbeds

In this section the testbed results are presented. First, the Memory Consumer and Memcached testbed results of the final configuration are presented. These results were used as the adviser's application profiling function in the benchmarking experiments. They were also used as the performance function in the simulations. Thereafter, testbed results of different Memcached configurations are presented. The first is Memcached as it comes out-of-the-box, with fixed cache size, and the second is dynamic Memcached, with no special OS configurations.

4.2.1 Memory Consumer Testbed

In the testbed, Memory Consumer was configured with a 2000 MB saturation point, and 50 MB spare memory and 0.1 seconds for sleep time between memory write attempts. Loads between 1 and 10 concurrent requests were sampled, and memories between 600 MB and 2400 MB. Each point was measured for 60 seconds.

The Memory Consumer testbed results are presented in figure 4.2. It can be seen that the performance is linear up to the saturation point, and remained constant afterwards, exactly in accordance with design. The result function is concave and non-decreasing. Each point was measured once after the memory was increased and once after the memory was decreased. A comparison of those measurements shows no hysteresis.

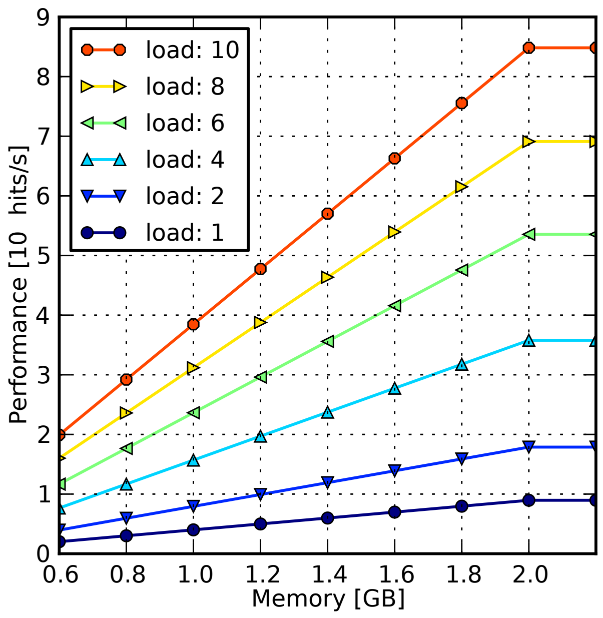


Figure 4.2: Testbed results for Memory Consumer. The performance (hit rate) as a function of memory allocation for different load values. The linear behavior up to a saturation point is exactly in accordance with design.

4.2.2 Dynamic Memcached Testbed

In the testbed, Memcached was configured with dynamic allocation and memslap (the load producing program) was configured with a key size of 249 bytes, a value size of 1024 bytes, a window size of 100 KB and a get/set ratio of 3:7. Loads between 1 and 10 concurrent requests and memories between 600 MB and 2400 MB were sampled. Each point was measured for 200 seconds.

The Memcached testbed results are presented in figure 4.4. It can be seen that for each load the performance increased up to the saturation point and remained constant afterwards. We can also see that the saturation point differs for each load. As the load increases, the performance saturates with higher memory allocation. The result function is non-decreasing and concave in most of its regions. Each point was measured once after the memory was increased and once after the memory was decreased. A comparison of these measurements shows no hysteresis.

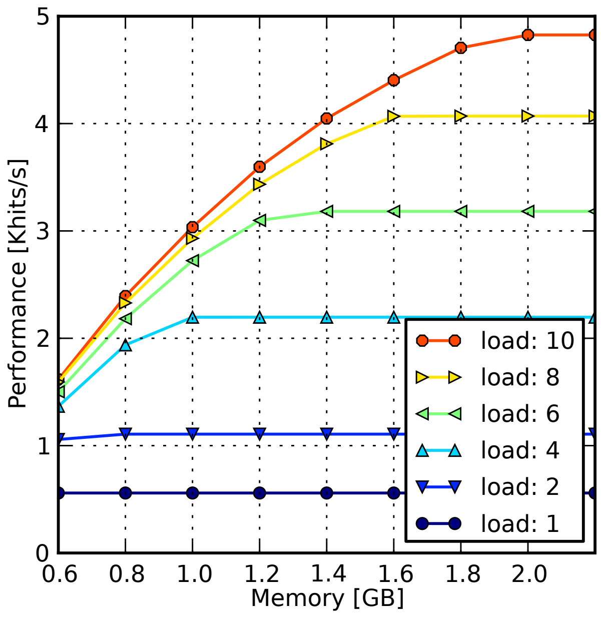


Figure 4.3: Testbed results for dynamic Memcached. The performance (hit rate) as a function of memory allocation for different load values. Dynamic Memcached with the OS configuration gives a non-decreasing and mostly concave performance function, which is consistent and stable.

Executing the memslap process on the host side and sending requests to the Memcached application on the guest side produced high load on the virtual network between the host and the guests. We found that the network had trouble transporting this large traffic, and essential packets found it difficult to reach their destination. Therefore, we changed the configuration such that we ran both Memcached and memslap inside the guest, and all the requests were passed using the local loopback address. By doing so the network was no longer a bottleneck and the one CPU assigned to the guest became the critical path. By configuring the local loopback to have an emulated delay using the `netem` tool, we emulated the network bottleneck again without actually stressing any resource of the system. We used a network delay of 1 millisecond[1] and only had to reconfigure the memslap window size to 500 KB to get results which behave similarly to the configuration where memslap ran on the host. The testbed results for this configuration can be seen in figure 4.4.

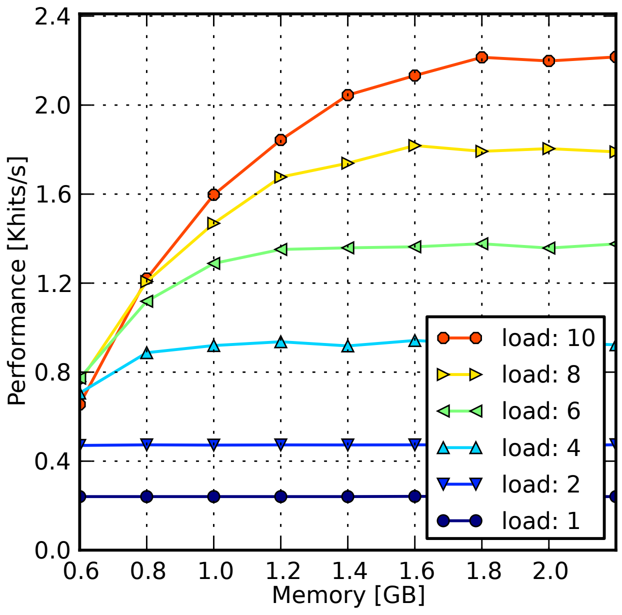


Figure 4.4: Testbed results for dynamic Memcached with memslap running on the guest side. The performance (hit rate) as a function of memory allocation for different load values. Running memslap from inside the guest released the network pressure caused by running it from the host. An emulated network delay was used to emulate the behavior of concurrent clients on a single CPU.

[1]To emulate a 1 millisecond network delay in the local loopback, a value of 0.5 millisecond was set. Since the delay is emulated twice on every request (one in each direction), the result is a total delay of 1 millisecond.

4.2.3 Testbed Examples of Infelicitous Configurations

In the previous sections (sections 4.2.1 and 4.2.2), workable and stable performance testbed results were presented. In this section, we will present how Memcached behaves out-of-the-box, or with the default OS configuration, in order to explain how we arrived at the final configuration.

Static Memcached

Memcached, as it is an out-of-the-box application, has a fixed cache size. The cache size is defined when the Memcached process starts, and is constant as long as Memcached runs. The static Memcached was configured with cache size of 500 MB, and memslap was configured with the same parameters as in section 4.2.2. Loads between 1 and 10 concurrent requests, and memories between 300 MB up to 900 MB were sampled. Each point was measured for 200 seconds.

The testbed results are shown in figure 4.5(a). The application shows very poor performance, under any load, up to an allocation of about 600 MB. For allocations of 700 MB and upward, the performance for each load is almost constant. The non-functionality of Memcached for memory allocations lower than 700 MB is due to severe swapping. This can be seen in the sampling of guest OS major-faults, in figure 4.5(b). Static Memcached, configured with 500 MB cache size, needs at least 700 MB to operate, but higher memory allocations are no use to it.

This kind of application is not suitable for a dynamic memory cloud computer. It requires a fixed amount of memory, and its performance does not improve from additional memory allocation.

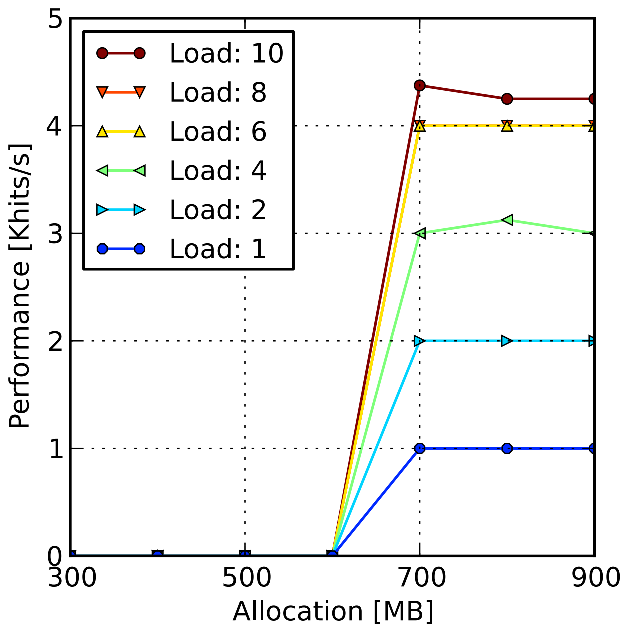
Dynamic Memcached without any Special OS configuration

The dynamic configuration of Memcached allows it to change its cache size. A larger cache size increases the hit percent of the requests, and thus the hit rate. Gaining more benefit from extra memory is not enough; dynamic memory applications also need the OS to trust them. If the OS interferes with their memory and swaps it to the disk because of memory pressure, the performance of the application will decrease when it starts to consume a large percent of the available memory.

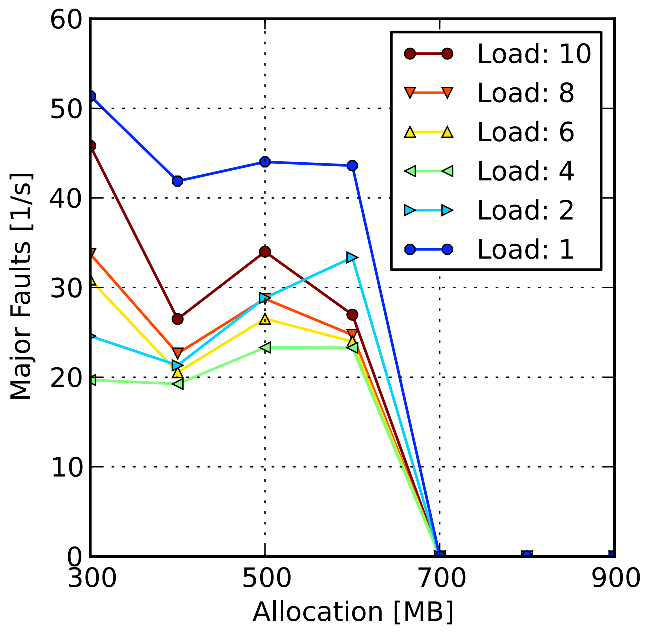
In this testbed, dynamic Memcached and memslap were configured with the same parameters as in section 4.2.2. The guest OS, however, had the default configuration, and was not configured as described in section 2.6.3.

The testbed results can be seen in figure 4.6. The performance graph shows unexpected behavior. There is an unexplained drop in the performance at a memory allocation of around 1800 MB, and inconsistent behavior for higher memory allocations. Those behaviors can be only explained by OS interference.

This example, which shows unaccepted results, proves that the guest OS configuration for trusting the dynamic memory application is a necessary companion for the dynamic



(a) Performance



(b) Major faults

Figure 4.5: Testbed results for unmodified (static) Memcached, configured with 500 MB cache size. In (a) it is shown that Memcached had very poor performance with a memory allocation lower than 700 MB, and had almost constant performance value in higher memory allocations. In (b) it can be seen that the poor performance is due to high number of major faults, and that there are no major faults at all when the performance achieve their maximal value.

memory application.

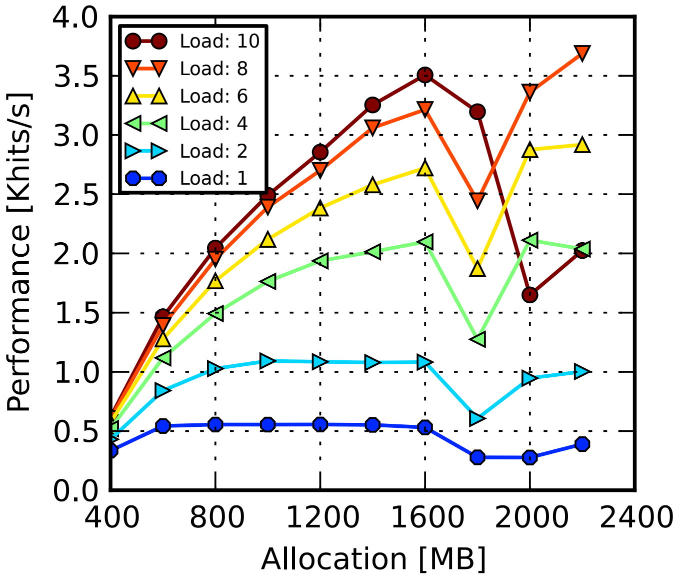


Figure 4.6: Testbed results for dynamic Memcached with unconfigured OS. Unexpected and inconsistent behaviors in high memory allocations can be explained by the OS interference with the application's allocated memory.

4.3 Benchmarking Experiments

The benchmarking experiments were conducted in order to compare Ginseng to other cloud memory management alternatives (see section 3.3.1) and the simulation of the same experiments (see section 3.3.2). We conducted the experiments in sets, where each set is characterized by the workload, the guest valuation functions and the distribution of the wealth of the guests. In each set we experimented with various numbers of guests, and for each number of guests we evaluated all the different memory management alternatives.

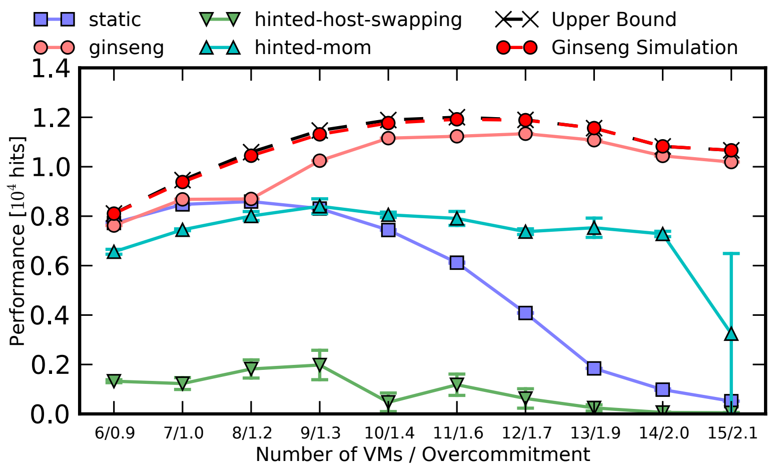
The guests were subject to dynamic loads, which were exactly the same for the different evaluated systems. In the Ginseng system the reclaim factor was set to 1, and guests used the strategy described in section 2.4.1. In each experiment set, a different combination of workload, guest valuation shape function and guest significance coefficient distribution, as described in section 3.3.1, was chosen.

General note for the graphs. Ginseng results are compared to the different memory management systems as a function of the number of guests and overcommitment ratio, for dynamic load experiments. The solid lines show the results of the experiment, while the dashed lines represent the Ginseng simulations and the optimized simulations.

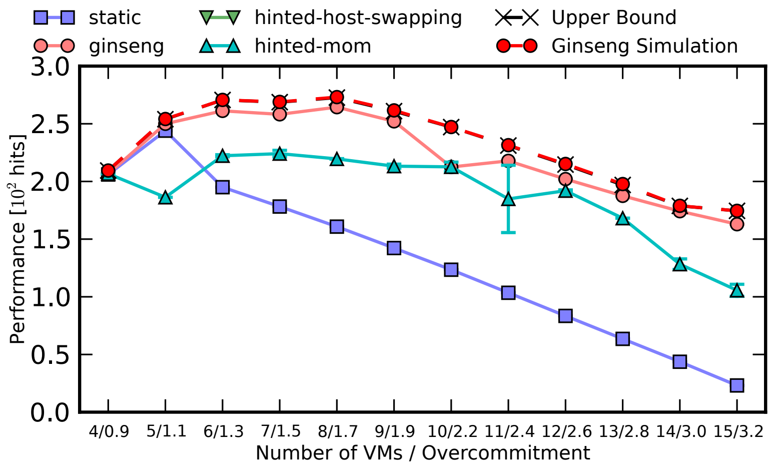
4.3.1 Results for valuation function equal to performance

In this experiment sets, the Memcached and Memory Consumer guests were configured with valuation function equal to the performance. The significance coefficients were all set to one. These experiments demonstrate the efficiency of Ginseng for maximizing the performance of an application. The guests, which are configured with valuation equal to the performance, bid according to the maximal performance. Since the host is looking for the maximal social welfare allocation, it is also looking for the maximal performance allocation. For Memcached, each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. For Memory Consumer, each experiment lasted 30 minutes, with $T_{\text{load}} = 200s$.

In this case the social welfare and the performance are the same, up to the measured unit. The comparison of Ginseng performance to the alternative systems is shown in figure 4.7(a).



(a) Memcached



(b) Memory Consumer

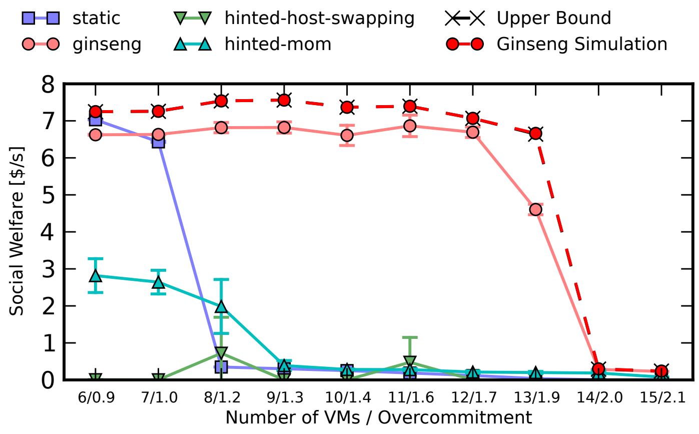
Figure 4.7: Results for Memcached and Memory Consumer with valuation function equal to performance. See explanation general note on the beginning of section 4.3

4.3.2 Results for Memcached with a piecewise linear valuation shape function and significance coefficients corresponding to three significant guests

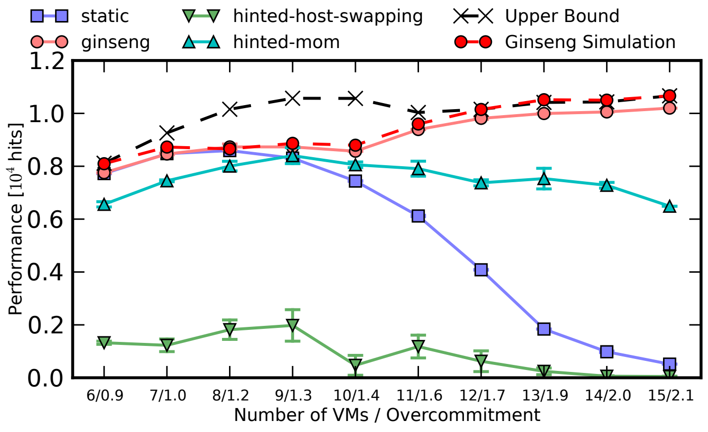
In this experiment set, the Memcached guests were configured with a piecewise linear performance valuation shape function and the significance coefficients correspond to three significant guests. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-5}.

The comparison of social welfare between the different alternative systems is shown in figure 4.8(a), and comparison of performance is also available in figure 4.8(b).

When the number of significant guests is small, and the valuation shape function is piecewise linear, the maximal possible social welfare is constant as long as those guests can have memory sufficient for their minimal required performance. It drops down when there is not enough memory for auction for the minimal performance value.



(a) Social welfare



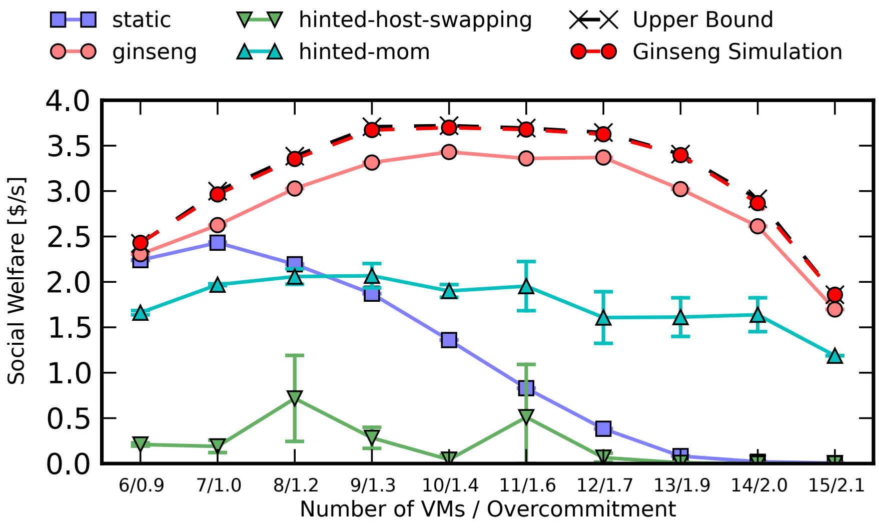
(b) Performance

Figure 4.8: Results for Memcached with a piecewise linear valuation shape function and significance coefficients corresponding to three significant guests. See explanation general note on the beginning of section 4.3

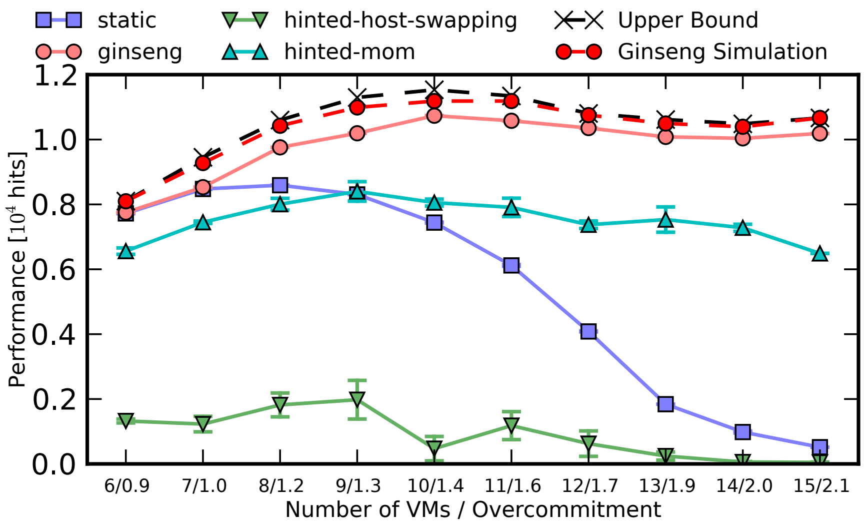
4.3.3 Results for Memcached with a second order valuation shape function and Pareto distributed significance coefficients

In this experiment set, the Memcached guests were configured with a second order valuation shape function and Pareto distributed significance coefficients. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-7}.

The comparison of social welfare between the alternative systems is shown in figure 4.9(a), and comparison of performance is also available in figure 4.9(b).



(a) Social welfare



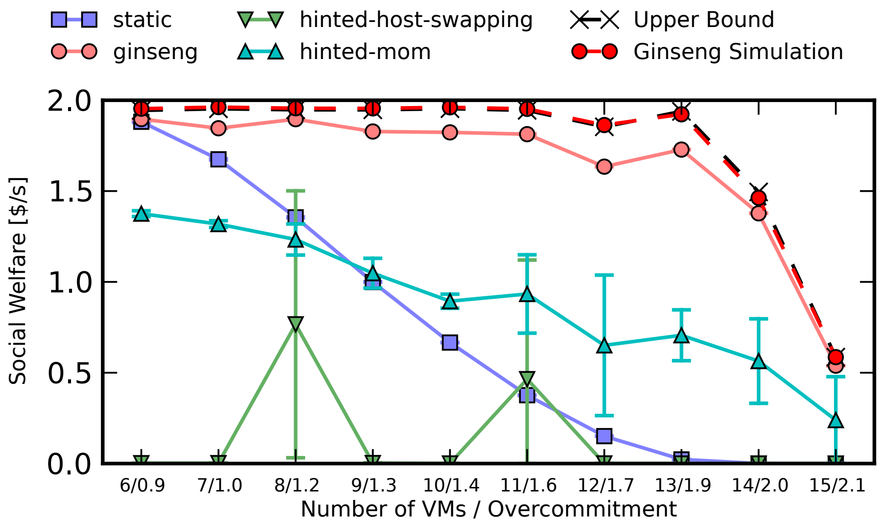
(b) Performance

Figure 4.9: Results for Memcached with a second order valuation shape function and Pareto distributed significance coefficients. See explanation general note on the beginning of section 4.3

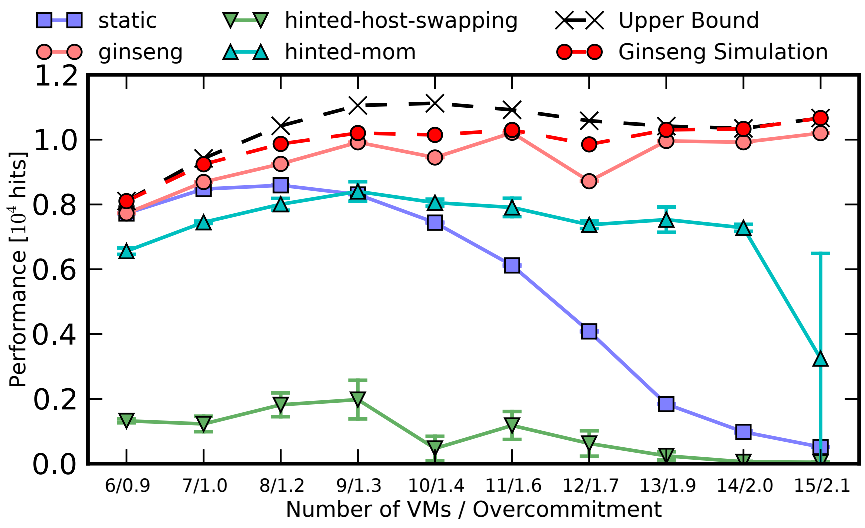
4.3.4 Results for Memcached with a second order valuation shape function and one significant guest

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients correspond to one significant guest. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-9}.

The comparison of social welfare between the different alternative systems is shown in figure 4.10(a), and comparison of performance is also available in figure 4.10(b).



(a) Social welfare



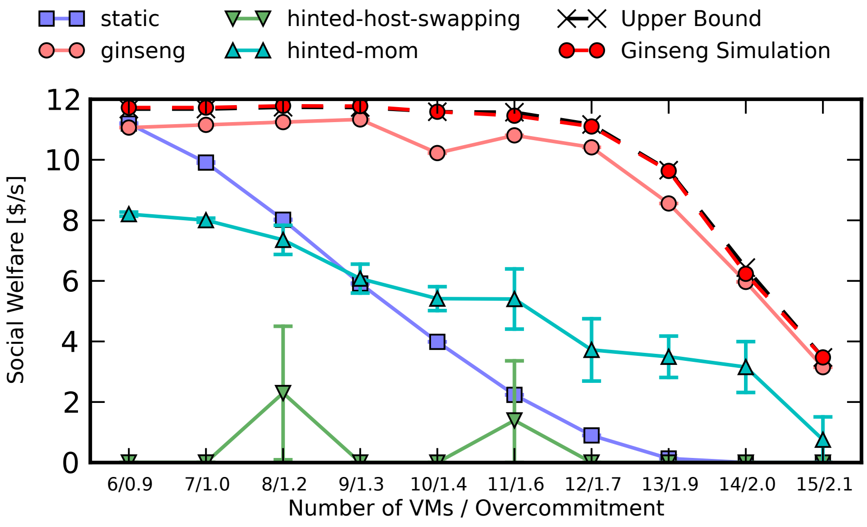
(b) Performance

Figure 4.10: Results for Memcached with a second order valuation shape function and one significant guest. See explanation general note on the beginning of section 4.3

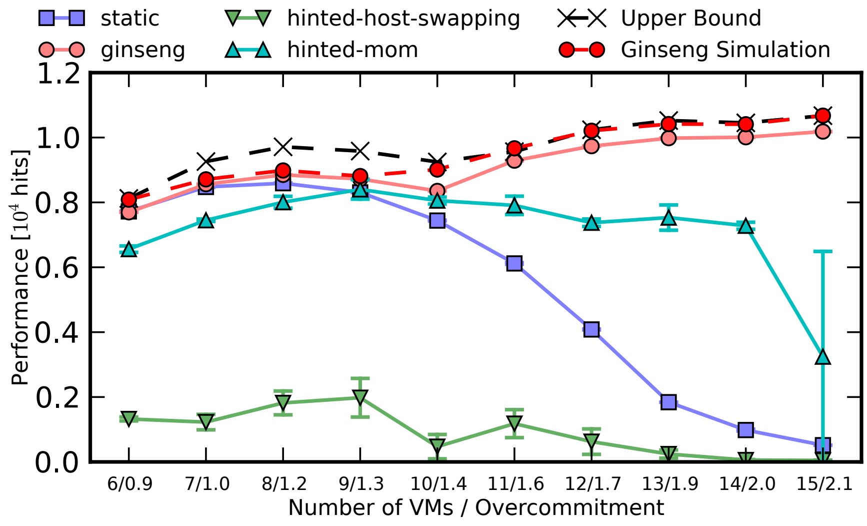
4.3.5 Results for Memcached with a second order valuation shape function and three significant guests

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients correspond to three significant guests. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-9}.

The comparison of social welfare between the different alternative systems is shown in figure 4.11(a), and comparison of performance is also available in figure 4.11(b).



(a) Social welfare



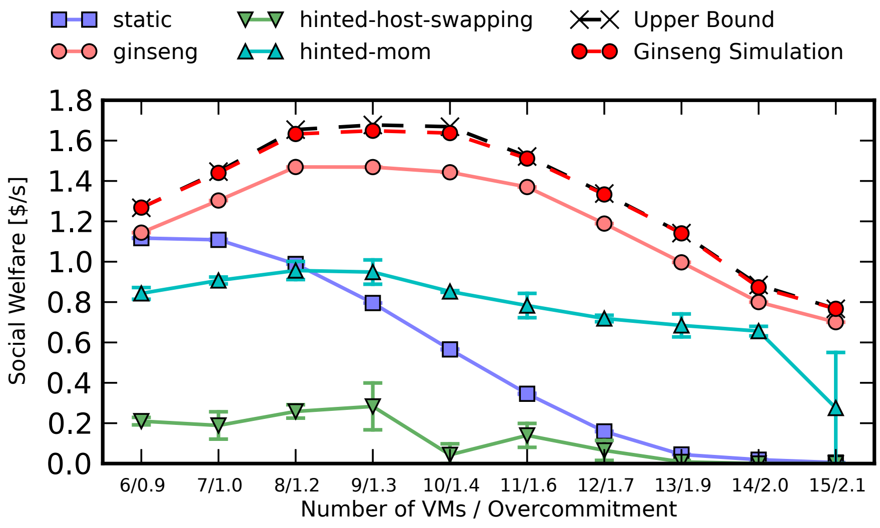
(b) Performance

Figure 4.11: Results for Memcached with a second order valuation shape function and three significant guests. See explanation general note on the beginning of section 4.3

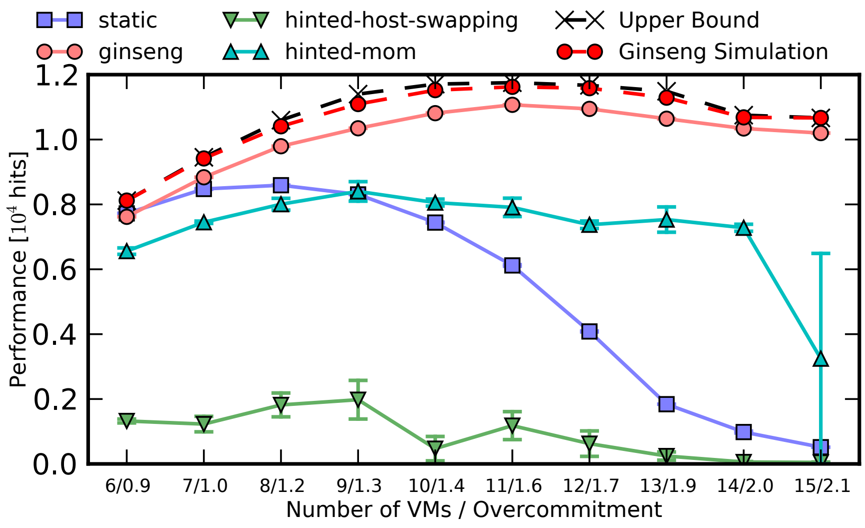
4.3.6 Results for Memcached with a second order valuation shape function and equal guests

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients were equal to one another. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-7}.

The comparison of social welfare between the different alternative systems is shown in figure 4.12(a), and comparison of performance is also available in figure 4.12(b).



(a) Social welfare



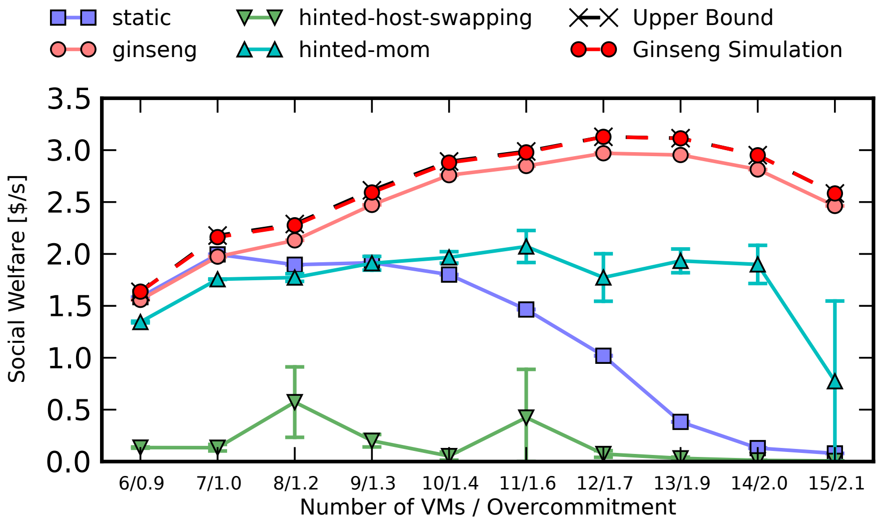
(b) Performance

Figure 4.12: Results for Memcached with a second order valuation shape function and equal guests. See explanation general note on the beginning of section 4.3

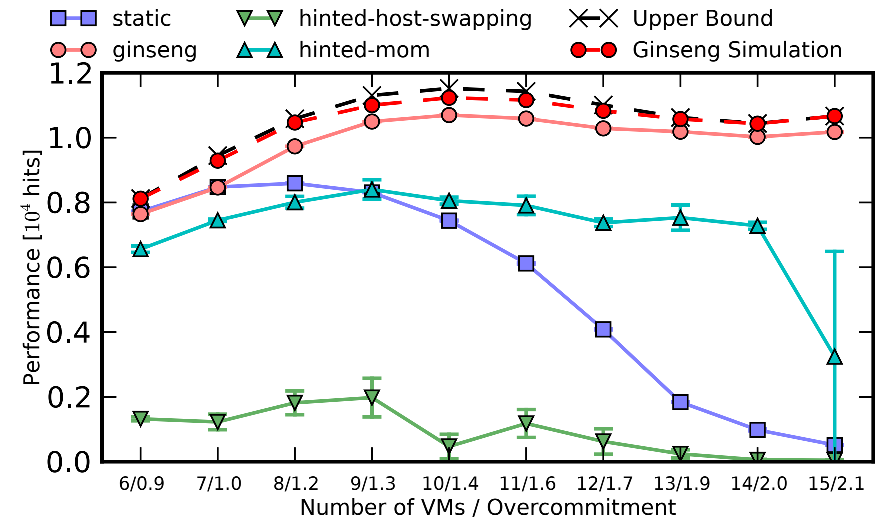
4.3.7 Results for Memcached with a linear valuation shape function and Pareto distributed significance coefficients

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients were drawn from the Pareto distribution. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-4}.

The comparison of social welfare between the different alternative systems is shown in figure 4.13(a), and comparison of performance is also available in figure 4.13(b).



(a) Social welfare



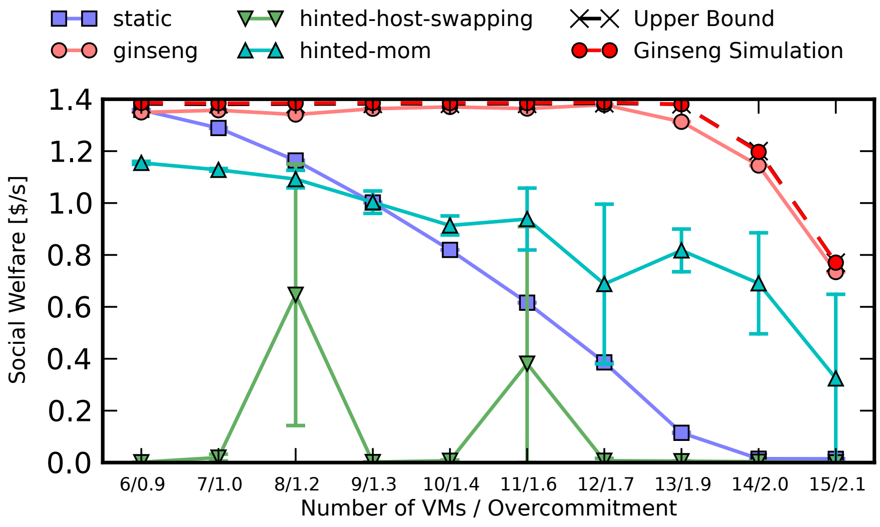
(b) Performance

Figure 4.13: Results for Memcached with a linear valuation shape function and Pareto distributed significance coefficients. See explanation general note on the beginning of section 4.3

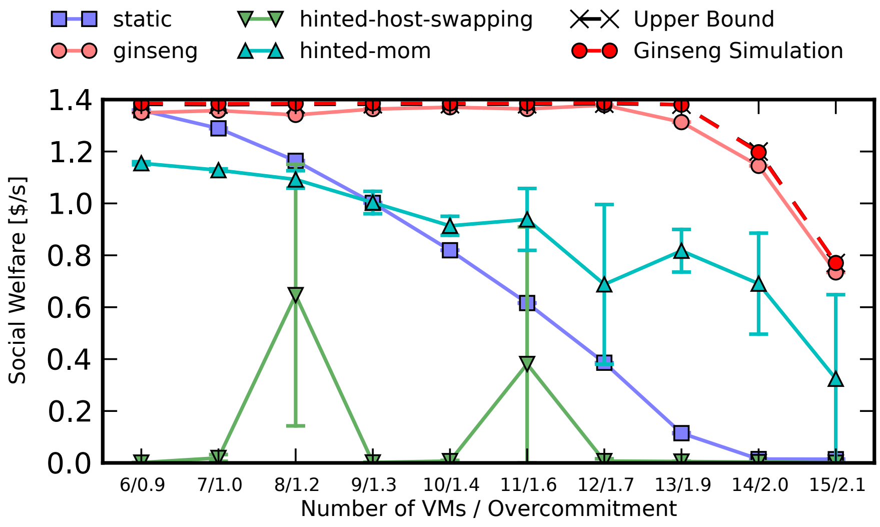
4.3.8 Results for Memcached with a linear valuation shape function and one significant guest

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients correspond to one significant guest. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-6}.

The comparison of social welfare between the different alternative systems is shown in figure 4.14(a), and comparison of performance is also available in figure 4.14(b).



(a) Social welfare



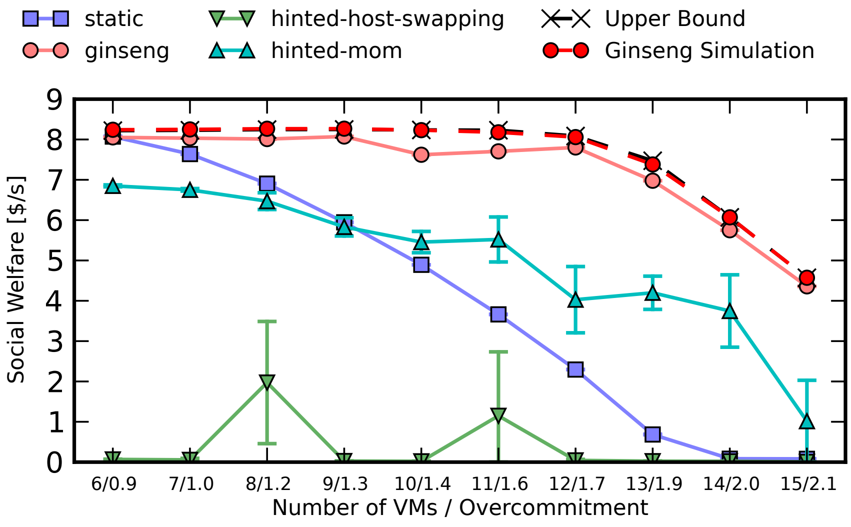
(b) Performance

Figure 4.14: Results for Memcached with a linear valuation shape function and one significant guest. See explanation general note on the beginning of section 4.3

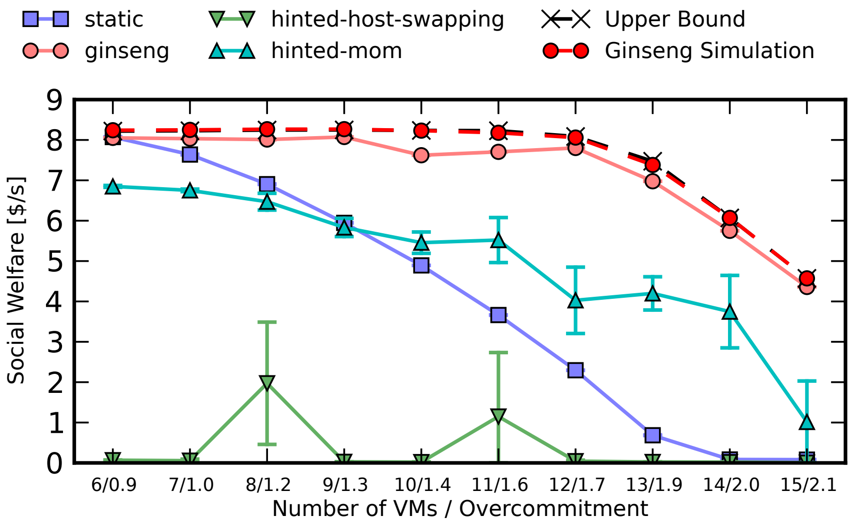
4.3.9 Results for Memcached with a linear valuation shape function and three significant guests

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients correspond to three significant guests. Each experiment lasted 60 minutes, with $T_{\text{load}} = 1000s$. The social welfare results were scaled with the scalar value of 10^{-6}.

The comparison of social welfare between the different alternative systems is shown in figure 4.15(a), and comparison of performance is also available in figure 4.15(b).



(a) Social welfare



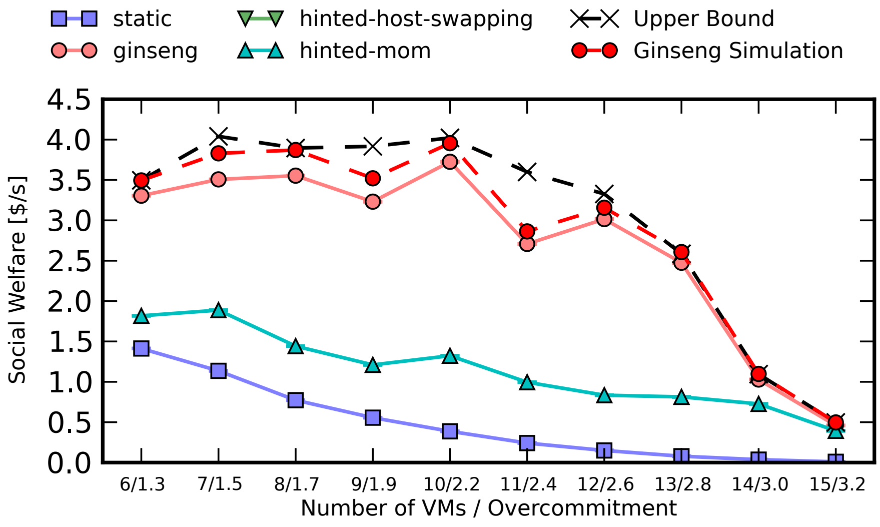
(b) Performance

Figure 4.15: Results for Memcached with a linear valuation shape function and three significant guests. See explanation general note on the beginning of section 4.3

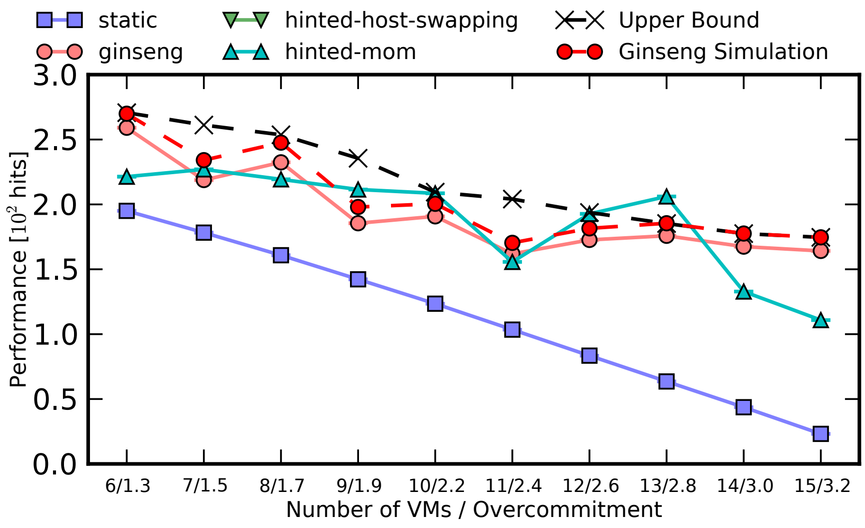
4.3.10 Results for Memory Consumer with a second order valuation shape function and Pareto distributed significance coefficients

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients were drawn from the Pareto distribution. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-7}.

The comparison of social welfare between the different alternative systems is shown in figure 4.16(a), and comparison of performance is also available in figure 4.16(b).



(a) Social welfare



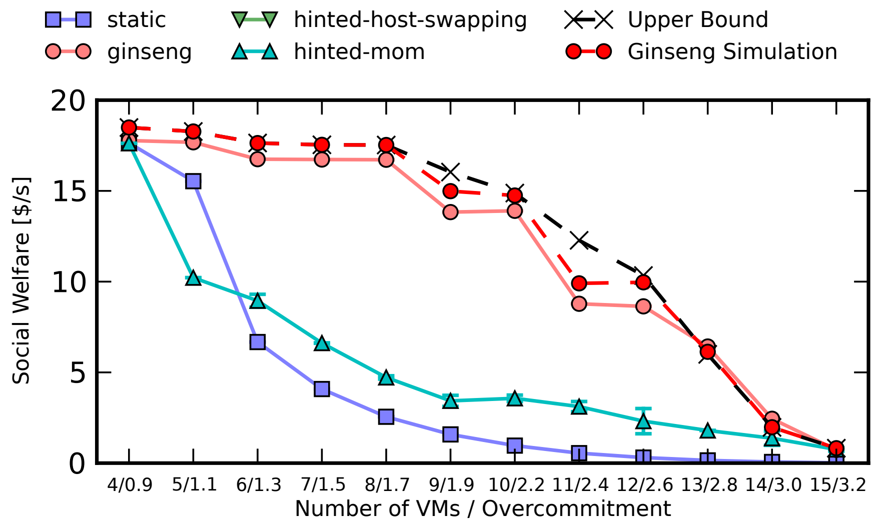
(b) Performance

Figure 4.16: Results for Memory Consumer with a second order valuation shape function and Pareto distributed significance coefficients. See explanation general note on the beginning of section 4.3

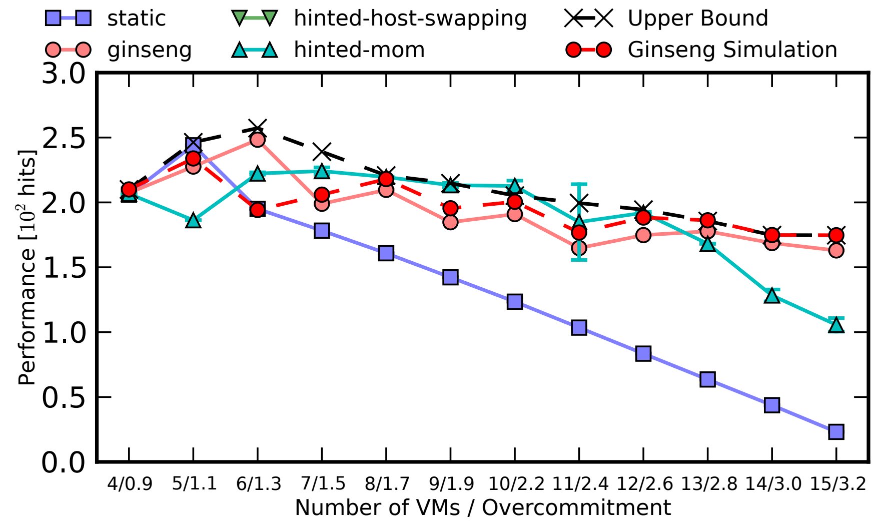
4.3.11 Results for Memory Consumer with a second order valuation shape function and three significant guests

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients correspond to three significant guests. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-9}.

The comparison of social welfare between the different alternative systems is shown in figure 4.17(a), and comparison of performance is also available in figure 4.17(b).



(a) Social welfare



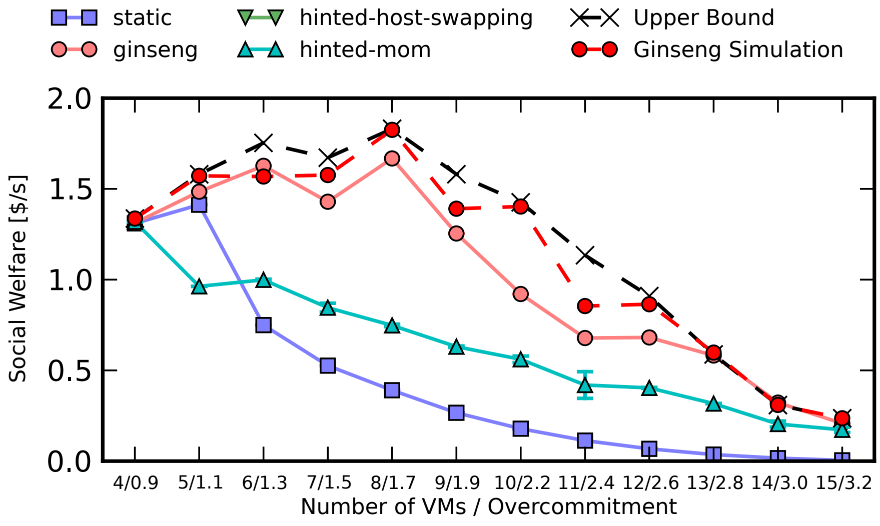
(b) Performance

Figure 4.17: Results for Memory Consumer with a second order valuation shape function and three significant guests. See explanation general note on the beginning of section 4.3

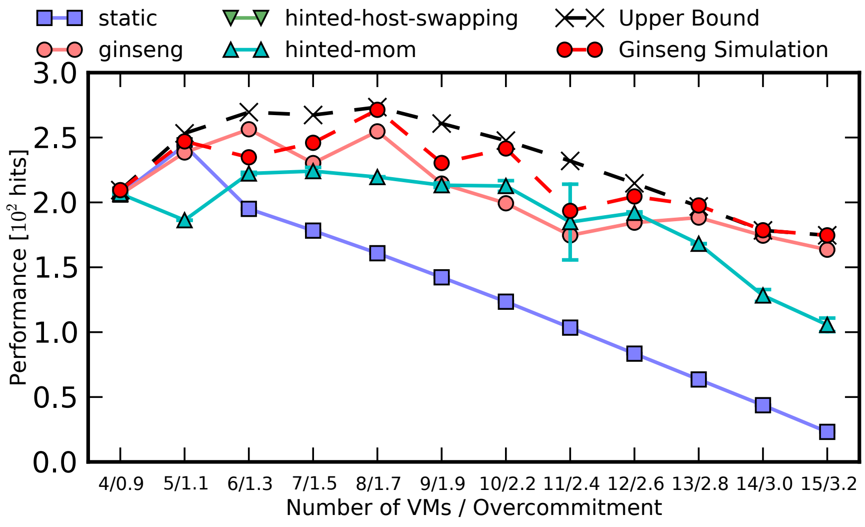
4.3.12 Results for Memory Consumer with a second order valuation shape function and equal guests

In this experiment set, the guests were configured with a second order performance valuation shape function and the significance coefficients were equal to one another. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-7}.

The comparison of social welfare between the different alternative systems is shown in figure 4.18(a), and comparison of performance is also available in figure 4.18(b).



(a) Social welfare



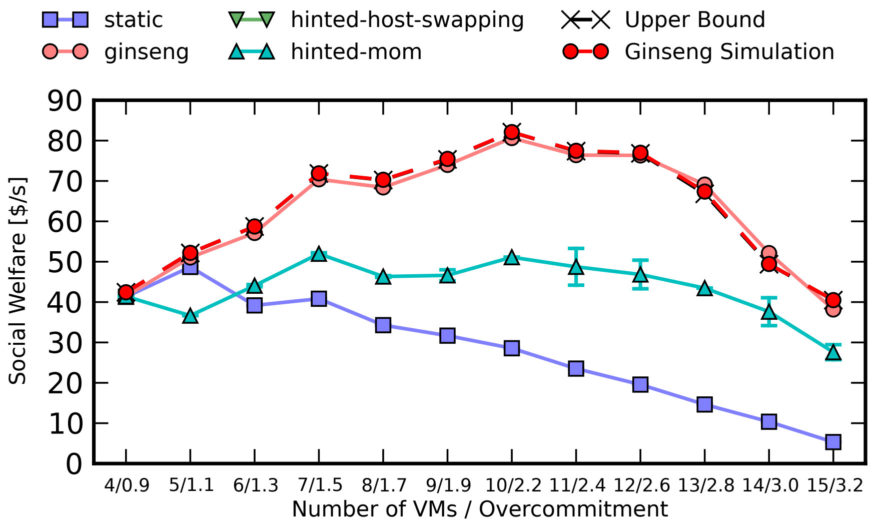
(b) Performance

Figure 4.18: Results for Memory Consumer with a second order valuation shape function and equal guests. See explanation general note on the beginning of section 4.3

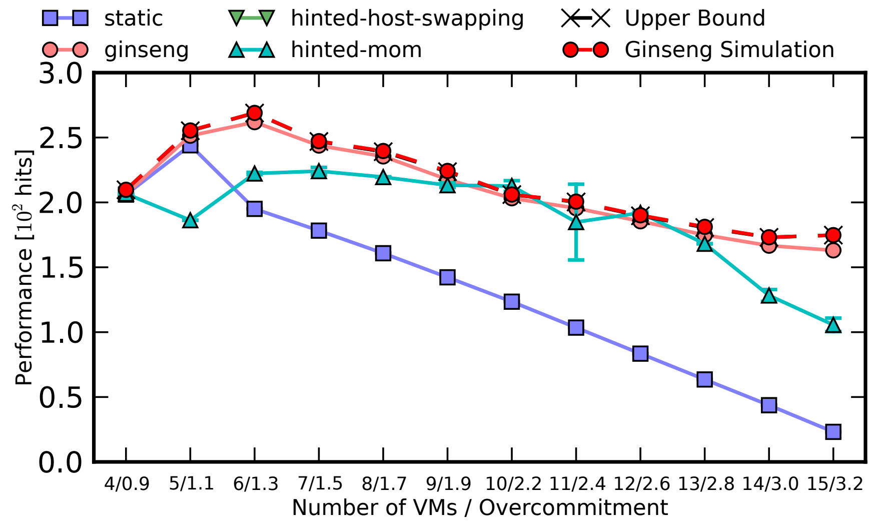
4.3.13 Results for Memory Consumer with a linear valuation shape function and Pareto distributed significance coefficients

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients were drawn from the Pareto distribution. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-4}.

The comparison of social welfare between the different alternative systems is shown in figure 4.19(a), and comparison of performance is also available in figure 4.19(b).



(a) Social welfare



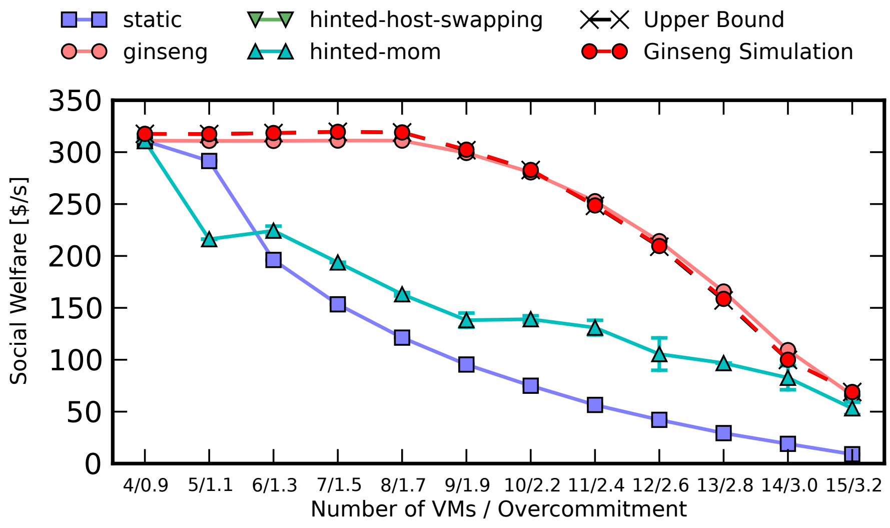
(b) Performance

Figure 4.19: Results for Memory Consumer with a linear valuation shape function and Pareto distributed significance coefficients. See explanation general note on the beginning of section 4.3

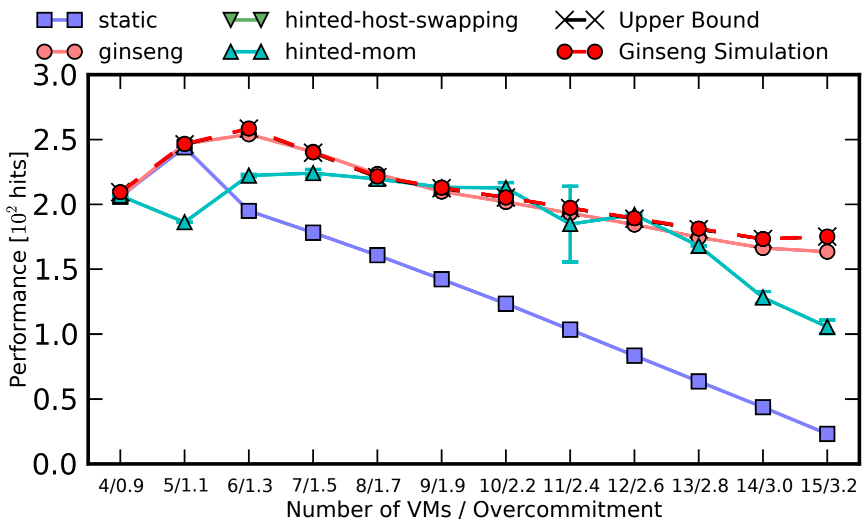
4.3.14 Results for Memory Consumer with a linear valuation shape function and three significant guests

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients correspond to three significant guests. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-6}.

The comparison of social welfare between the different alternative systems is shown in figure 4.20(a), and comparison of performance is also available in figure 4.20(b).



(a) Social welfare



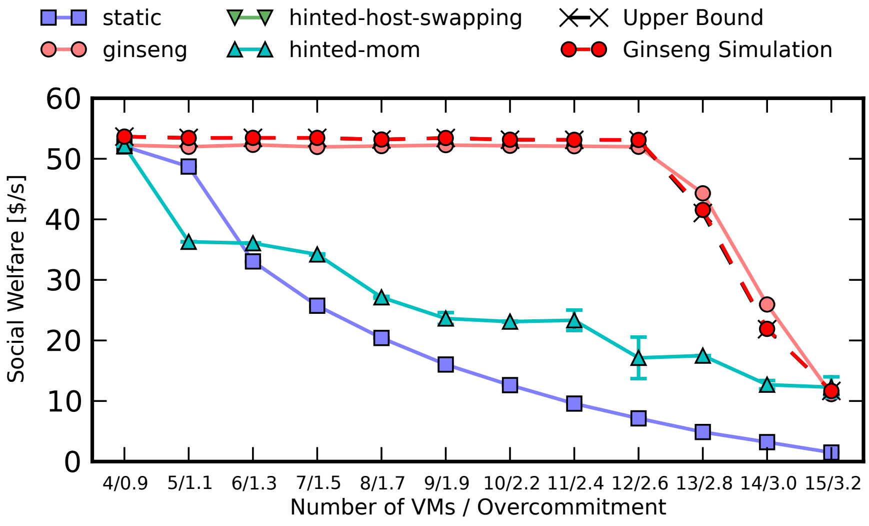
(b) Performance

Figure 4.20: Results for Memory Consumer with a linear valuation shape function and three significant guests. See explanation general note on the beginning of section 4.3

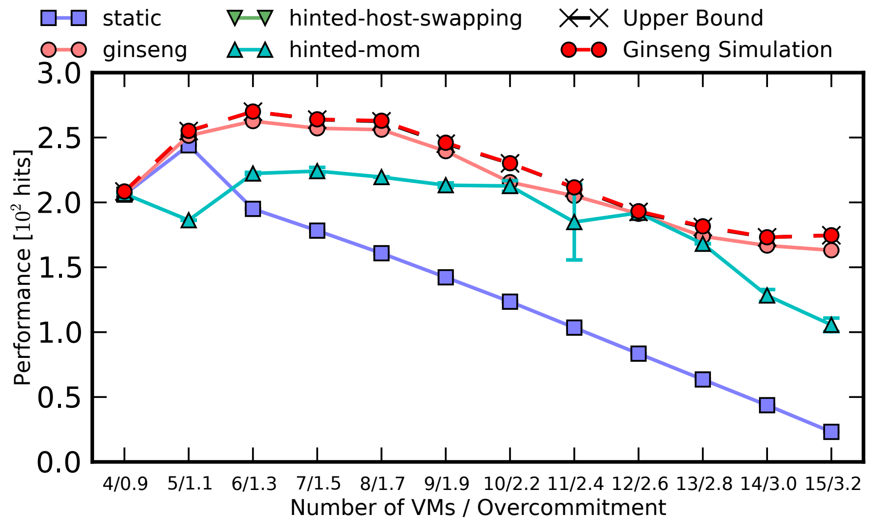
4.3.15 Results for Memory Consumer with a linear valuation shape function and one significant guest

In this experiment set, the guests were configured with a linear performance valuation shape function and the significance coefficients correspond to one significant guest. Each experiment lasted 30 minutes with $T_{\text{load}} = 200s$. The social welfare results were scaled with the scalar value of 10^{-6}.

The comparison of social welfare between the different alternative systems is shown in figure 4.20(a), and comparison of performance is also available in figure 4.21(b).



(a) Social welfare



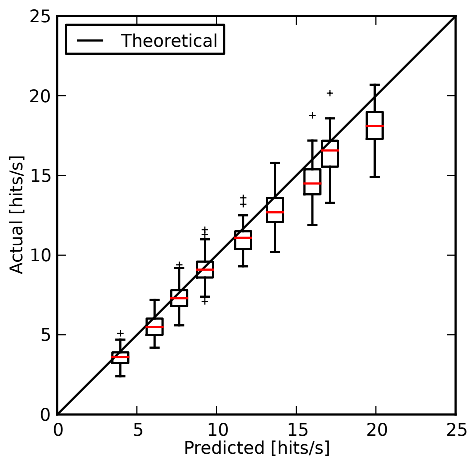
(b) Performance

Figure 4.21: Results for Memory Consumer with a linear valuation shape function and one significant guest. See explanation general note on the beginning of section 4.3

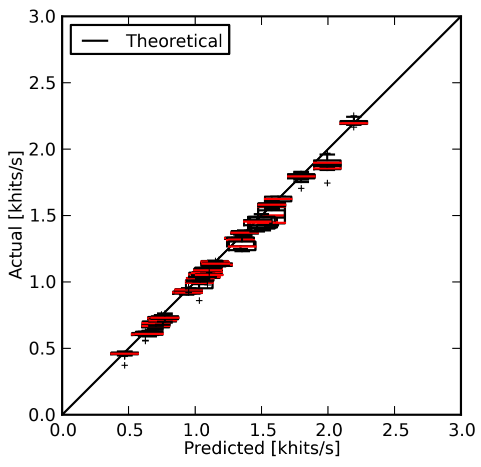
4.3.16 Offline Profiling Relevance

Threesomes of memory allocation, load and performance were sampled during the benchmarking experiments. For each pair of memory and load, the performance predicted by the guest's adviser could be computed according to the profiling function, and then compared with the actual performance from the initial threesome. This comparison shows the relevance of offline profiling.

In figure 4.22, the performance measured in the testbed was validated by comparing it with the performance measured in the experiments and shown in figures 4.2 and 4.4. We can see that the majority of the samples are close to the theoretical ideal. For Memcached we can see negligible deviation, and that most of the samples were predicted correctly. In the Memory Consumer comparison we can see that the deviation is much higher. This comparison is important because the adviser for the guest bid is making decisions according to the performance measured in the testbed. If the performance is far from the actual performance, the adviser's advice is misleading. Mistakes in prediction can be overcome by on-line measurements of the performance and a correction of the profiling function.



(a) Memory Consumer



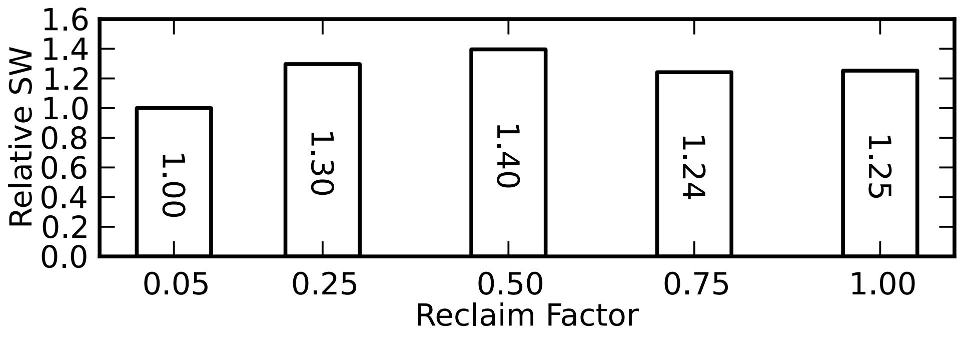
(b) Memcached

Figure 4.22: Validation of the performance predicted from the testbed, by comparing it with that measured in the experiments. For predicted performance values with at least ten values, boxes extend from the first quartile to the third, and the mean value is marked.

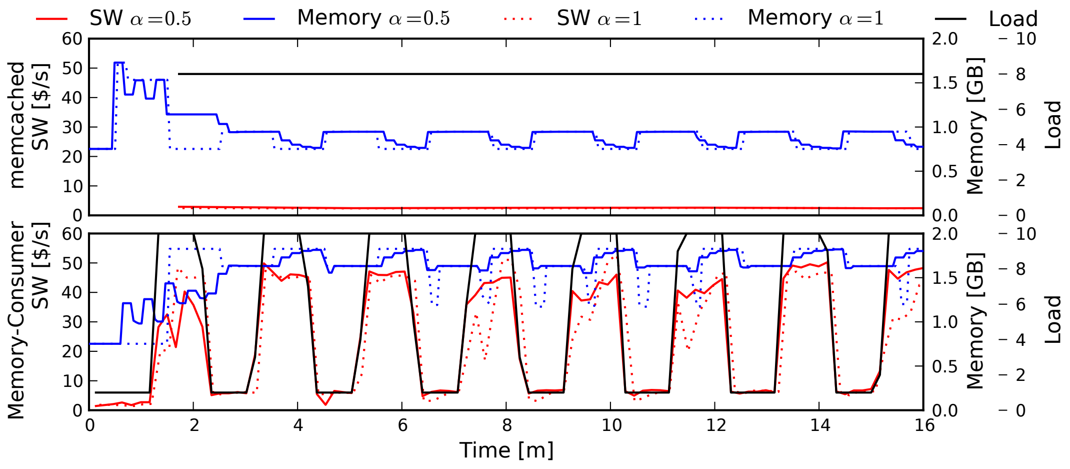
4.3.17 Reclaim Factor Influence

To examine the impact of the reclaim factor on social welfare in a real system, we combined a statically loaded Memcached guest, which is vulnerable to allocation cycles, with a dynamically loaded Memory Consumer guest, whose load changed every 60 seconds. Each guest got *bare* memory of 600 MB. We repeated the same experiment for different reclaim factors.

A comparison of the total social welfare between those experiments is presented in figure 4.23, in which the highest social welfare was achieved for reclaim factor value of 0.5, and decreased for higher or lower values. A trace of the measured load, memory allocation and valuation of the two guests for reclaim factors of 0.5 and 1 is presented figure 4.23(b), in which the reclaim factor memory restrain is made perceptible.



(a) Comparison between different reclaim factors.



(b) Traces of experiments with reclaim factor of 0.5 and 1.

Figure 4.23: Impact of reclaim factor on social welfare for a mixed workload of Memcached and Memory Consumer, and traces of the experiments demonstrating the difference in the behavior

4.4 Simulations

We used simulations to evaluate various aspects of Ginseng's performance. The simulations allowed us to run and test Ginseng with a variety of parameter ranges and for longer durations. We simulated Ginseng running with 10 guests for 1000 rounds. All the guests were configured with the adviser described in section 2.4.2, and with the performance valuation function: $V(P) = P$.

4.4.1 Memory Exchange Penalty Influence

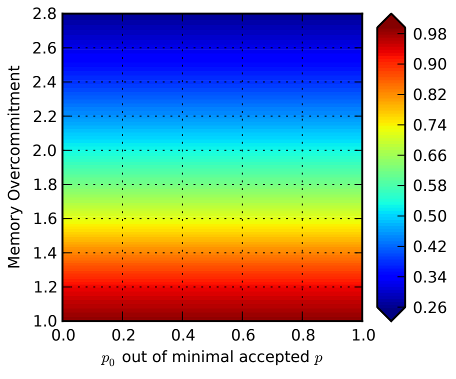
The figures present the social welfare, sum of guest utilities, and host revenue, in addition to the upper bound on waste, ties, and inefficiency. All of them are presented as a function of overcommitment and β, the percent of p_0 out of the minimal accepted bid's unit-price in the previous round.

The results of simulations testing the influence of β on Ginseng performance are presented in figure 4.26 for Memory Consumer, and in figure 4.27 for Memcached.

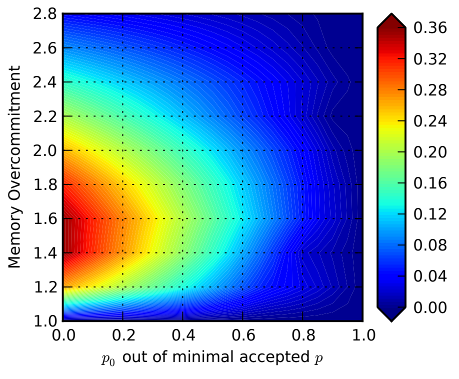
4.4.2 Reclaim-Factor Influence

The figures present the social welfare, sum of guest utilities, and host revenue, in addition to the upper bound on waste, ties, and inefficiency. All of them are presented as a function of reclaim factor and overcommitment.

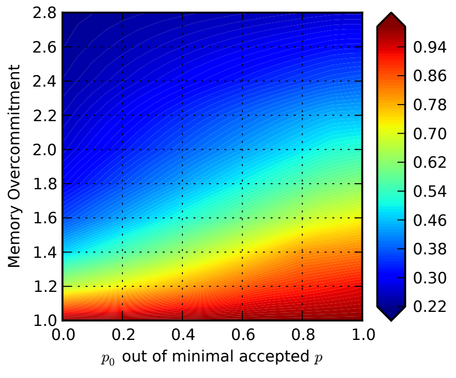
The reclaim-factor influence simulation results are presented in figure 4.26 for Memory Consumer, and in figure 4.27 for Memcached.



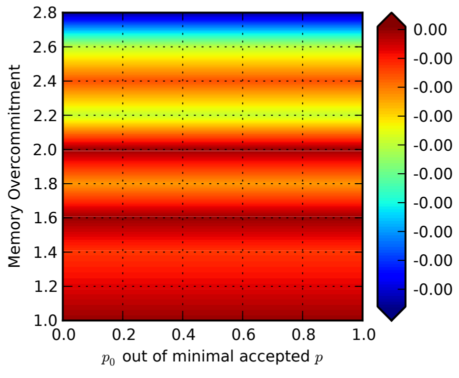
(a) Social welfare



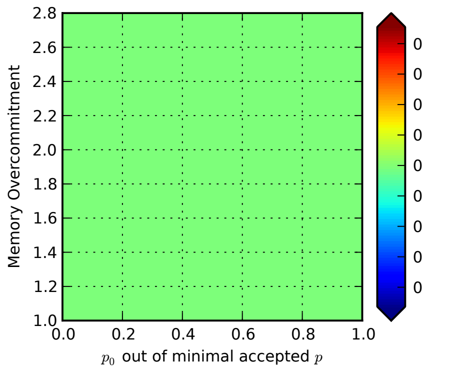
(b) Host revenue



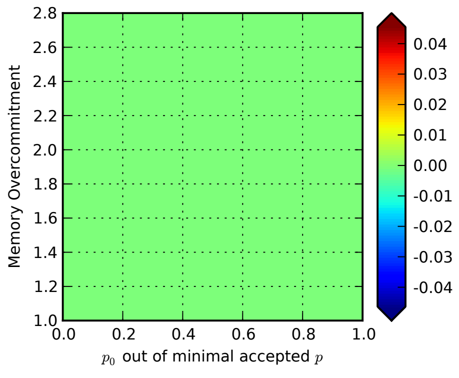
(c) Sum of guest utilities



(d) Inefficiency

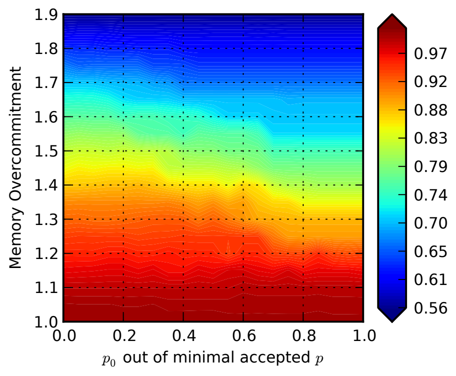


(e) Waste (MB)

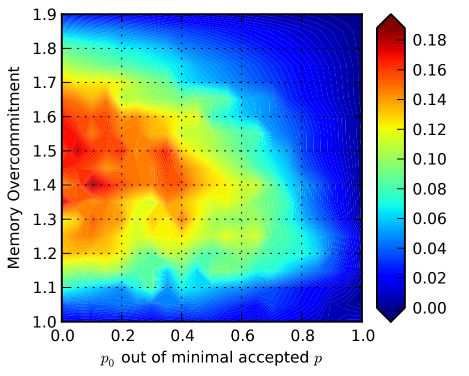


(f) Ties

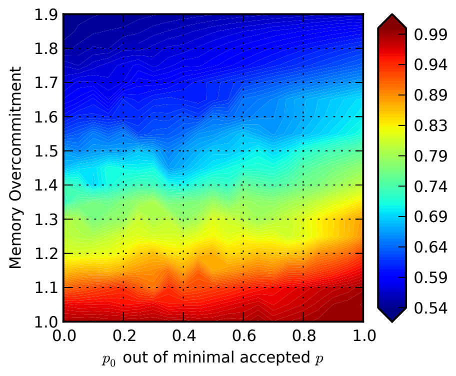
Figure 4.24: Simulation results for Memory Consumer. The impact of the β, the percent of p_0 out of the minimal accepted bid's unit-price in the previous round, and overcommitment ratio on Ginseng time-averaged performance. Social welfare, revenue and profit values are normalized by the maximal social welfare achieved in the parametric sweep.



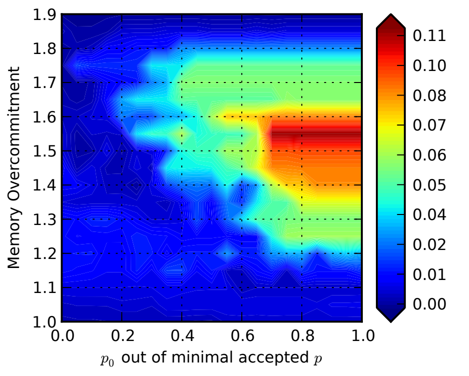
(a) Social welfare



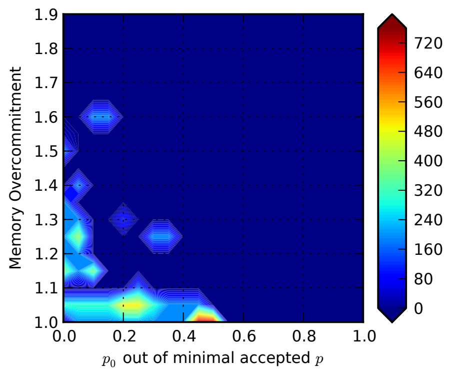
(b) Host revenue



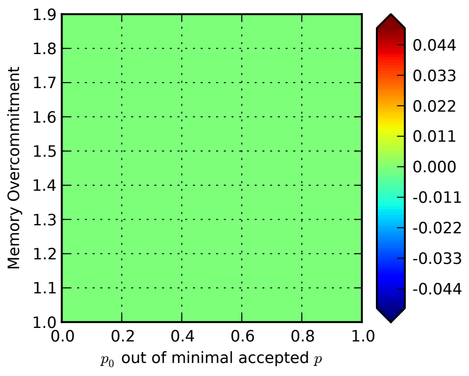
(c) Sum of guest utilities



(d) Inefficiency



(e) Waste (MB)



(f) Ties

Figure 4.25: Simulation results for Memcached. The impact of the β, the percent of p_0 out of the minimal accepted bid's unit-price in the previous round, and overcommitment ratio on Ginseng time-averaged performance. Social welfare, revenue and profit values are normalized by the maximal social welfare achieved in the parametric sweep.

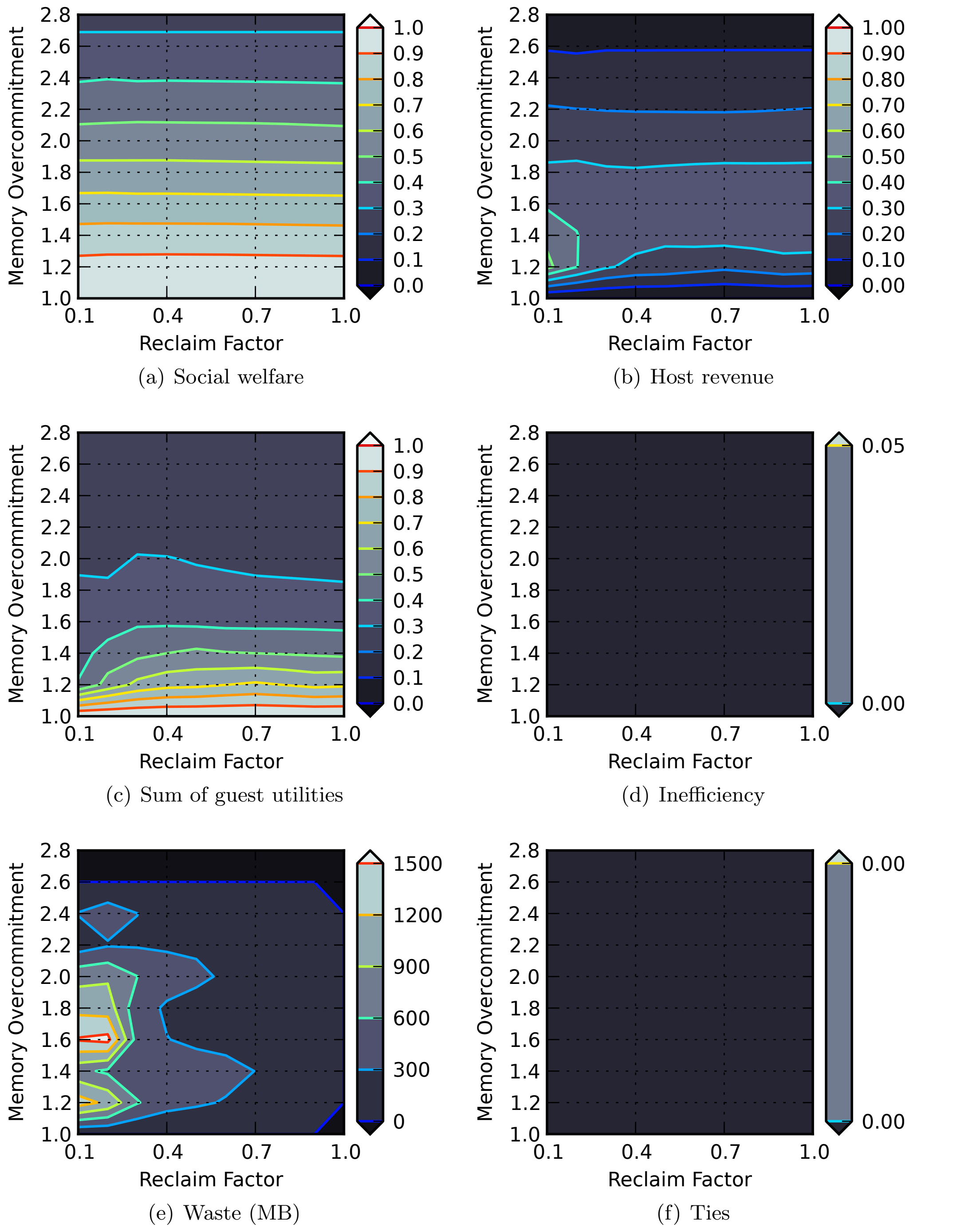


Figure 4.26: Reclaim factor simulation results for Memory Consumer. The impact of the reclaim factor and overcommitment ratio on Ginseng's time-averaged performance. Social welfare, revenue and profit values are normalized by the maximal social welfare achieved in the parametric sweep.

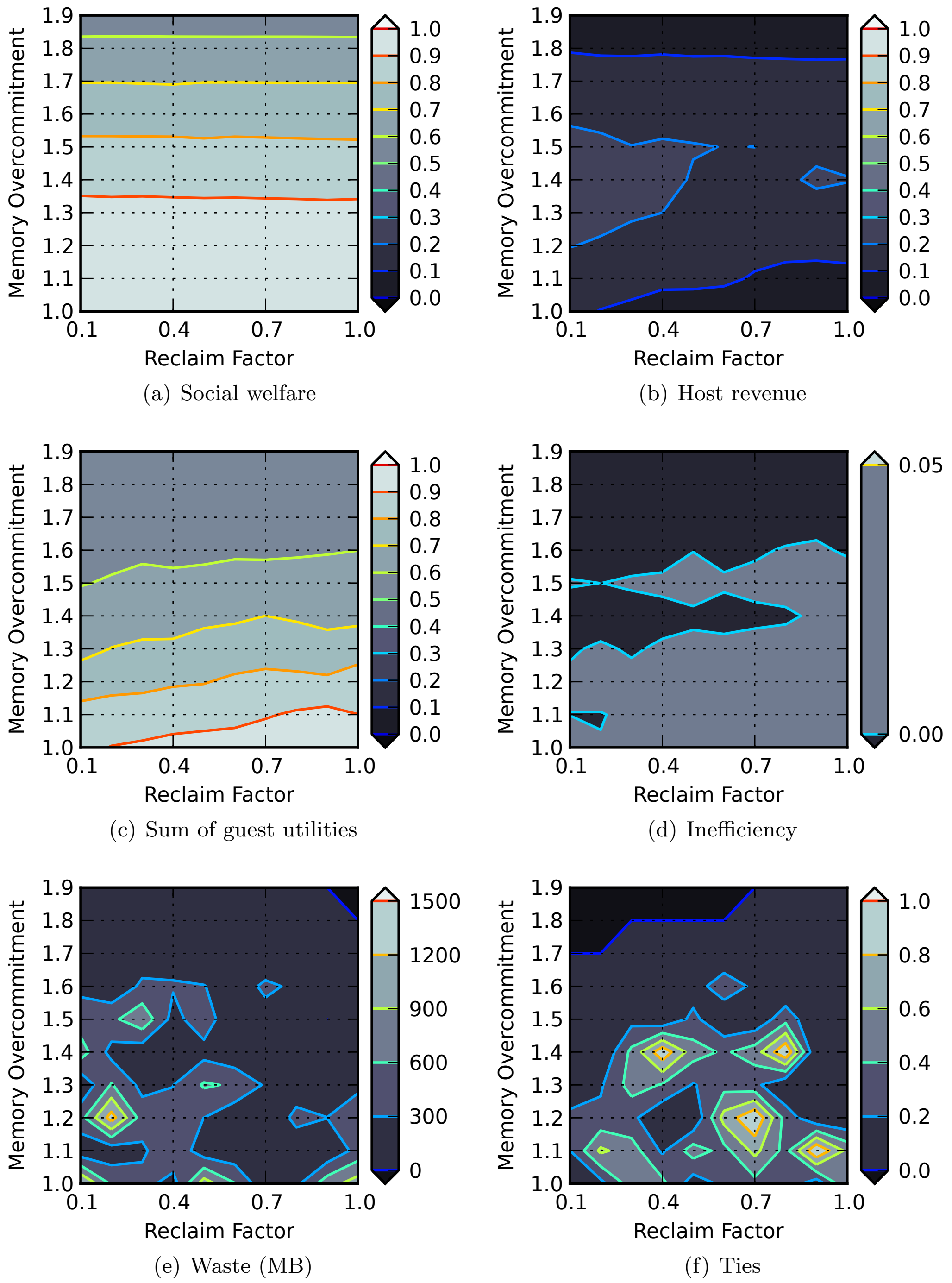


Figure 4.27: Reclaim factor simulation results for Memcached. The impact of the reclaim factor and overcommitment ratio on Ginseng time-averaged performance. Social welfare, revenue and profit values are normalized by the maximal social welfare achieved in the parametric sweep.

Chapter 5

Discussion and Conclusions

5.1 Discussion

In this section we will discuss the results of the experiments and simulations, try to quantify the efficiency of Ginseng, and understand how the suggested approaches for dynamic memory cloud computing improve the known and existing approaches.

5.1.1 Memory Change Experiments

In the memory change experiments, presented in figures 4.1(a) and 4.1(b), the effectiveness of the dynamic memory applications and guest hinting before memory change is shown. Both dynamic applications increase their performance when they have more available memory, and both avoid thrashing when the memory is reclaimed by freeing it beforehand. The system responds as expected to sudden memory changes.

Our explicit hinting approach is similar to the one proposed in [Vor13], but has not been tested yet. Less efficient approaches that limit the memory change rate were used in [Wan09, GHDS^{+}11, HGS^{+}11]. In those approaches, which use implicit knowledge, the memory degradation is dependent on the OS memory management system, which should respond to memory pressure. Moreover, the memory change process becomes unacceptably long in its duration and limited in its amount.

5.1.2 Testbed

The testbed procedure, presented in section 3.2, is similar to the "staging server" which was used in QClouds [NKG10], in which a static linear relation between QoS and resource allocation was created using least mean square of the samples. We show that a linear relation is not enough to represent this function, as can be seen even in our cleanenst testbed results in figures 4.2 to 4.4.

It is also similar to the correlation stage in Ginkgo [HGS^{+}11, GHDS^{+}11], in which a correlation between performance as function of load and memory was created. In Ginkgo, the samples were measured with allocations that differed by 32 MB, and for more than an hour in each load and memory point. However, our specially developed dynamic memory applications and special OS configuration showed that resolution of the 100-200 MB allocation difference between samples is fine enough, and moreover, a duration of 60 seconds for Memory Consumer and 200 seconds for Memcached were long enough to give small variance and no hysteresis in the results. A duration of an hour is not practical, not even for the results to be valid in a real time system.

Ginkgo also presents results similar to our testbed results, but under the magnifying glass strange behavior can be seen, such as discrepancy between the performance and load, and strange, discrepant and unexplained behavior of the performance function. The testbed results that involved our dynamic memory applications and the hinting mechanism (see figures 4.2 to 4.4) show that there is logic behind the performance function. The function increases as the memory allocation increases and higher loads result in higher performance.

In [HZPW09], results of performance profiling as a function of memory are also presented. Their results show an example of a static heap size allocations with OS thrashing for memory allocation which are lower than the minimal requirement. These results are similar to the results we experienced on the out-of-the-box Memcached with un-configured OS, as can be seen in figure 4.5. Our dynamic memory application approach, as presented in section 2.6.1, avoids this kind of behavior, which is not suitable for a dynamic memory cloud computer.

Support for the dynamic memory application approach is presented in [HGS$^+$11, GHDS$^+$11]. JavaBalloons is a modification of JVM that allows heap size changes. But the profiling graph of the JavaBalloons also shows static heap size behavior that suggests that they are inefficient.

5.1.3 Experiment results

The experiments were conducted in order to compare Ginseng to other cloud memory management alternatives, for different workloads and different valuation function shapes and distribution. In this section we discuss the comparison, for which we used black-box clients for Ginseng and white-box clients for the alternatives.

The memory exchange penalty, as presented in section 2.1.5, was not used in the presented results. The conditional allocation rules, as presented in section 2.1.2, were used instead. The second approach, like the first, is used in order to prevent costly memory exchanges, and has a similar effect. The disadvantage of conditional allocation is that we cannot assume truthful guests. Since our guests were programmed to be truthful, the results are still valid for our developed system. However, using the memory exchange approach might yield slightly different results.

The social welfare achieved by Ginseng was evaluated and compared to the social welfare of each of the five other methods listed in table 3.1 for a varying number of guests on the same physical host. The social welfare and performance of the different experiments is compared in section 4.3. The figures contain two upper bounds for the social welfare, achieved through simulations, which are presented in section 3.4. The tighter bound results from a simulation of Ginseng itself, and the looser bound results from a white-box on-line simulation. The MOM and host-swapping methods (in their not hinted version) yield negligible social welfare values for these experiments, and are not presented in the graphs.

Ginseng achieves much better social welfare than any other memory management system alternative for both workloads. It improves social welfare by $\times 31.5$ for Memcached and up to $\times 6.2$ for Memory Consumer, compared with all other approaches. Since each guest is allocated a fixed amount of memory ($bare$) on startup, as the number of guests increases, the potential for social welfare increases, but our host has less free memory to auction.

In all experiment sets, Ginseng achieves around 83%–100% of the optimal social welfare.

Using Ginseng does not guarantee that the performance of the system will be maximized, since Ginseng was designed to maximize the social welfare of the system. However, we show that by setting the valuation function to be proportional to the performance, Ginseng improves the performance up to $\times 1.6$ for the Memcached workload, relative to the alternative systems.

Reclaim Factor influence

The results of reclaim the factor experiment, as described in section 4.3.17, are given in figure 4.23 for various reclaim factors. The trace of load, memory and valuation for two of these experiments can be seen in figure 4.23(b). The experiment shows that lowering the reclaim factor reduces the penalty that Memory Consumer suffers when conditions change and it needs to change its strategy. When the reclaim factor is lower, the system gets sluggish and does not stabilize before the load changes again.

In real systems there is a trade-off between system responsiveness and limitation of allocation cycles that does not exist in simulations. This experiment proves the importance of the reclaim factor as a knob for the host to control the system's stability.

Offline Profiling

In figure 4.22 we compare our benchmarks' predicted performance with the actual measured performance values during Ginseng experiments. The comparison shows that the profiled data is accurate enough, as can be seen when comparing Ginseng's results to its simulations.

The performance prediction is important to the advising strategy of the guest. More accurate advising bid recommendations will lead to higher social welfare. Measuring the performance on-line with feedback to the advising unit and a correction of the profiler function can improve the accuracy of the bids.

5.1.4 Simulation Results

The simulation results are presented in section 4.4. The simulations were conducted in order to help evaluate various aspects of Ginseng's performance and enable to run and test Ginseng in a variety of parameter ranges and for longer durations.

Memory Exchange Penalty Influence

Determining p_0, by β, as described in section 2.1.2, causees p_0 to be higher as the unit price of the submitted bids is higher and the prices of the bids increased. This technique gives an adaptive value of p_0 to the currently submitted bids. Nevertheless, more sophisticated techniques could be employed, to improve the performance of the system.

The influence of β was studied through simulations. Figures 4.24 and 4.25 presents simulation results of the Memory Consumer and Memcached benchmarks respectively. Each is simulated in the entire space of memory overcommitment and β values. Since β directly influences p_0, as higher β results in higher p_0 values, we will consider the β horizontal axis, as a suggestion on p_0 values. Additionally we can see that in the sense of economic behavior of the system (social welfare, host revenue and guests utility), both workloads behave quite similarly.

It can be seen that p_0, has negligible (if any) influence on the social welfare, and as expected, a higher overcommitment ratio reduces the social welfare, as fewer goods are being sold to the same number of clients. On the other hand, p_0 has a strong influence on the host revenue. Higher values reduce host revenue significantly. On the vertical axis, we can see that the overcommitment demonstrate a supply and demand behavior on the host revenue; for small overcommitment ratios, where the supply is high, and for high overcommitment ratios, where there is hardly any supply, the host revenue is low, and for the middle range of overcommitment where there is the lowest ratio of supply over demand, the host revenue rises.

Memory Consumer, with the linear dependency of value and memory, exhibits nice behavior: no inefficiency, no memory waste and no ties over the whole range. On the other hand, Memcached, with the concave valuation functions, the behavior is less satisfactory. Inefficiency accrues for middle range overcommitment ratios and higher p_0 values. In the middle range overcommitment ratio the supply and demand ratio is low and the market prices are high; moreover, for higher β values, the p_0 values comprise a significant portion of the bidder's payment. The result is that in this area we expect the highest p_0 values. Higher p_0 values reduce memory exchange and the difference between the best social welfare to the Ginseng social welfare increases. This is expressed in higher inefficiency. We should keep in mind that this inefficiency is for a system that does not suffer from exchanging memory, and a real experiment might show better performance due to the cost of exchanging that memory. Memory waste is zero on most of the areas in the graph, but might be shown in the places where memory overcomitment is low and the p_0 is low. This might happen due to small values of the memory exchange penalty term.

Reclaim-Factor Influence

Memory Consumer static simulation results can be seen in figure 4.26. It is shown that the reclaim factor has low impact on the host revenue and sum of guest utilities, and no impact on the social welfare and the inefficiency. The inefficiency ranges from 0 in a well provisioned system to 35% for an overcommitment ratio of 3.5. The inefficiency can be reduced by using a richer bidding language [MT04a]. There are no ties in the Memory Consumer simulations, and usually no waste either. We attribute the lack of ties to the different slopes of the Memory Consumer performance graphs for the different loads (in figure 4.2).

figure 4.27 shows the static simulation results of Memcached guests. It can be seen that in this case as well, the impact of reclaim factor on the social welfare and inefficiency is negligible. The main influence on the sum of guest utilities is the overcommitment. We can see a greater influence of reclaim factor on the host revenue, but the trends are similar to the Memory Consumer simulation: host revenue has lower values in the high and low overcommitment values, and in the middle overcommitment range, higher values for lower reclaim factor.

In contrast to Memory Consumer tie graph, which had a value of zero for the entire range, Memcached tie graph shows that the simulations could have had up to 80% of the rounds with tied guests. This can be explained by the Memcached performance graphs which share the same slope in their lower parts (figure 4.4). This is consistent with our design assumption in section 2.1.2, that ties do happen in real life, and supports the claim that they must be efficiently dealt with.

Host Revenue

Ginseng does not attempt to maximize host revenue directly. Instead, it assumes that the host charges an admittance fee for cloud services and maximizes the aggregate client satisfaction (the social welfare). Maximizing social welfare improves host revenues indirectly because better-satisfied guests are willing to pay more. Likewise, improving each cloud host's hardware (memory) utilization should allow the provider to run more guests on each host. Nevertheless, it is interesting to examine the host's direct revenues.

In the both of the simulations, for small overcommitment ratios (< 1.3) the host revenue is negligible ($< 5\%$ of the maximal social welfare): the guest utilities (see figures 4.26(c) and 4.27(c)) equal to their valuations (see figures 4.26(a) and 4.27(a)). As the overcommitment ratio increases, host revenue decreases because there is less memory to rent. When the host revenue is zero and the social welfare is high, as in the case for the low overcommitment range, the system is functioning well and is in a state of equilibrium. In this state, guests are more considerate of their neighbors thanks to the exclusion compensation principle. Our guests reach such equilibrium using indirect negotiations that result from their learning strategy (in section 2.4.2).

5.2 Conclusions

Ginseng is the first cloud platform that allocates physical memory to selfish black-box guests while maximizing their aggregate benefit. It does so using the MPSP auction, in which even guests with non-concave valuation of memory are incentivized to bid their true valuations for the memory they request. Using the MPSP auction, Ginseng achieves an order of magnitude of improvement in the social welfare.

Although Ginseng focuses on selfish guests, it can also benefit altruistic guests (e.g., when all guests are owned by the same economic entity). In this case, economic valuations can distinguish between guests that perform the same function for different purposes, such as a test server vs. a production server.

In the research, we have shown the importance of dynamic memory applications, the efficiency of explicit guest hinting, and the contribution of the guest OS being configured to trust the dynamic memory application. Adopting those approaches was necessary to increase the system utilization.

Ginseng is the first concrete step toward the Resource-as-a-Service (RaaS) cloud [ABYST12]. In the RaaS cloud, all resources, not just memory, will be bought and sold on the fly. Extending Ginseng to resources other than physical memory remains as future work.

5.3 Future Work

The MPSP auction is uncharted territory with regard to game theory: the reclaim factor, which reduces waste, introduces private guest-states that change over time and affect the guests' valuation of additional memory chunks. When the base memory changes, it also changes the forbidden ranges. In addition, valuations may change at random due to dynamic loads. In this work we only analyzed guest strategies with a horizon of one auction round. In simpler problems of repeated games without private states, there may be rational strategies which are irrational to play as a stage game. This may also be the case here: there are strategies that are irrational in the stateless game (with $\alpha = 1$), but are rational in the private-state game. For example, if a guest expects a fast increase of demand for memory, it can plan ahead and bid for more memory than it currently needs. It will benefit from keeping its payments lower for several rounds, until the system reaches a new equilibrium. Even in a stateless game, prediction of other guests' bids may incentivize a guest to lie about its valuation in a repeated VCG auction. Analysis of such strategies calls for new theoretic approaches.

The performance of the guest's agent can be enhanced through learning. It can predict load changes, and bid according to predictions rather than according to the current load. It can predict the impact of its bid on allocation results. It can update the $P(m)$ on-line. It can use the knowledge it collected to devise side-channel attacks on its neighbors.

The host can learn how to dynamically change the reclaim factor. To this end it can utilize the total amount of memory that was bid for, in addition to its own plans to add or remove guests. It can also utilize black-box measurements to assess the rate at which conditions change.

Ginseng currently only allocates memory. It can be expanded to allocating resource bundles (e.g., memory, IO, and CPU) [ABYST12]. It can also be expanded to allow shedding memory, for example when the decay brings the guest to an undesired memory amount, or when the guest's requirements quickly change, but the reclaim factor is small.

Bibliography

[ABYST12] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Raas: Resource as a service. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Ottawa Linux Symposium (OLS)*, pages 19–28, 2009.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AM85] Robert J. Aumann and Michael Maschler. Game theoretic analysis of a bankruptcy problem from the talmud. *Journal of Economic Theory*, 36(2):195–213, August 1985.

[BBB$^+$08] Junjik Bae, Eyal Beigman, Randall Berry, Michael L. Honig, and Rakesh Vohra. An efficient auction for non concave valuations. In *9th International Meeting of the Society for Social Choice and Welfare*, 2008.

[BCI$^+$07] Christian Borgs, Jennifer T. Chayes, Nicole Immorlica, Kamal Jain, Omid Etesami, and Mohammad Mahdian. Dynamics of bid optimization in online advertisement auctions. In *International Conference on the World Wide Web (WWW)*, pages 531–540, 2007.

[CAT$^+$01] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[Cla71] Edward H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1):17–33, Sep 1971.

[DFH$^+$12] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: on fair sharing of multiple resources. In *Innovations in Theoretical Computer Science Conference (ITCS)*, pages 68–75. ACM, 2012.

[DN10] Shahar Dobzinski and Noam Nisan. Mechanisms for multi-unit auctions. *Journal of Artificial Intelligence Research*, 37:85–98, 2010.

[EOS07] Benjamin Edelman, Michael Ostrovsky, and Michael Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *American Economic Review*, 97(1):242–259, March 2007.

[Fit09] Brad Fitzpatrick. Memcached- free and open source, high-performance, distributed memory object caching system, August 2009. `http://memcached.org/`.

[GGW10] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management (CNSM)*, pages 9–16. IEEE, 2010.

[GHDS$^+$11] Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *ASPLOS RESoLVE '11: Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE) workshop*, 2011.

[GLV$^+$08] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2008.

[GN12] Avital Gutman and Noam Nisan. Fair allocation without trade. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 2, pages 719–728, 2012.

[Gro73] Theodore Groves. Incentives in teams. *Econometrica*, 41(4):617–631, Jul 1973.

[GZH$^+$11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2011.

[Heg10] John Hegeman. Facebook's ad auction. Talk at Ad Auctions Workshop, May 2010.

[HGS$^+$11] Michael Hines, Abel Gordon, Marcio Silva, Dilma Da Silva, Kyung Dong Ryu, and Muli 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.

[HZPW09] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui 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*, IM'09, pages 630–637, Piscataway, NJ, USA, 2009. IEEE Press.

[JADAD06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 14–24, 2006.

[Kel97] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[KKL$^+$07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, pages 225–230, 2007. `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`.[Accessed Apr, 2011].

[Lit11] Adam G. Litke. Memory overcommitment manager, 2011. `https://github.com/aglitke/mom`.

[LPLT12] Brendan Lucier, Renato Paes Leme, and Eva Tardos. On revenue in the generalized second price auction. In *International Conference on the World Wide Web (WWW)*, 2012.

[LS99] Aurel Lazar and Nemo Semret. Design and analysis of the progressive second price auction for network bandwidth sharing. *Telecommunication Systems - Special issue on Network Economics*, page http://comet.columbi, 1999.

[Mag08] Dan Magenheimer. Memory overcommit... without the commitment. In *Xen Summit*. USENIX association, June 2008.

[MMHF09] Grzegorz Miłoś, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference (ATC)*, 2009.

[MT04a] Patrick Maillé and Bruno Tuffin. Multi-bid auctions for bandwidth allocation in communication networks. In *IEEE INFOCOM*, 2004.

[MT04b] Patrick Maillé and Bruno Tuffin. Multi-bid versus progressive second price auctions in a stochastic environment. *Quality of Service in the Emerging Networking Panorama*, pages 318–327, 2004.

[NKG10] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkha. 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, New York, NY, USA, 2010. ACM.

[OZN$^+$12] Zhonghong Ou, Hao Zhuang, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon EC2. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[PHS$^+$09] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 13–26. ACM, 2009.

[PKRS11] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. In *ACM HotNets*, 2011.

[SFM$^+$06] Martin Schwidefsky, Hubertus Franke, Ray Mansell, Himanshu Raj, Damian Osisek, and JongHyuk Choi. Collaborative memory management in hosted linux environments. In *OLS '06: 2006 Ottawa Linux Symposium*, 2006.

[SM11] Vyas Sekar and Petros Maniatis. Verifiable resource accounting for cloud computing services. In *ACM Cloud Computing Security Workshop (CCSW)*, 2011.

[SSGW11] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing (SOCC)*, page 5. ACM, 2011.

[USR09] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), 2009.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16(1), 1961.

[Vor13] Anton Vorontsov. The mempressure control group proposal. LWN.net, January 2013.

[Wal02] Carl A. Waldspurger. Memory resource management in Vmware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[Wan09] Weiming Zhaoand Zhenlin Wang. Dynamic memory balancing for virtual machines. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, pages 21–30, 2009.

[WJC$^+$10] Hongyi Wang, Qingfeng Jing, Rishan Chen, Bingsheng He, Zhengping Qian, and Lidong Zhou. Distributed systems meet economics: pricing in the cloud. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[WTLS$^+$09] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *ACM/USENIX Int'l Conference on Virtual Execution Environments (VEE)*, pages 31–40, 2009.

מפורש על ידי יצירת לחץ זכרון, ובכך גורמת לאורח לפנותו. שיטות אלו איטיות ואינן מתאימות לשימוש בענן בעל זכרון דינמי, ובנוסף מדמות את פעולת לקיחת הזכרון של בלון האורח (balloon driver), כפי שמבוצעת בפועל בכל מקרה. שיטת ההתראה המפורשת נבדקה והוכחה כיעילה בהפעלת שינויים גדולים ומיידיים בזכרון, ויכולה למנוע העברת דפי זכרון לכונן הקשיח (swap) בשיתוף עם אפליקציה מתאימה. כמו כן, מוצג כיצד היא מביאה לשיפור בביצועים. אנו בנוסף מציגים כיצד שינוי במערכת ההפעלה מאפשר להשתמש באחוז גדול של הזיכרון ללא התערבותה. אנו מראים כי במערכת בה הזכרון לא מוקצה־יתר השינוי מביא לשיפור ביצועים.

במחקר פותחו סביבות בדיקה שונות. שתי סביבות בדיקה ראשונות הן לבדיקת אורח עם יישום זיכרון דינמי בשילוב עם מערכת התראה מפורשת מוקדמת ושינוי הקונפיגורציה של מערכת ההפעלה. בראשונה נמדדה תגובת היישום לשינויי זכרון גדול ופתאומי. במערכת זו ניתן למדוד כמה מהר היישום יכול להפיק תועלת מעליה בזכרון, והאם הוא מספיק לשחרר את הזכרון מספיק מהר לאחר שנודע לו הצורך בכך. מערכת שניה לאפיון בביצועי היישום תחת עומסי עבודה שונים ותחת הקצאות זיכרון שונות. המערכת מפיקה פוקנציה של הביצועים בתלות בעומס ובזכרון המשמשת לאורח לחזות את ביצועיו במהלך הריצה. שתי סביבות בדיקה נוספות פותחו על מנת לבדוק את ג׳ינסנג. מערכת אחת הריצה את ג׳ינסנג וחלופות אחרות לניהול זכרון תחת תנאים שווים, על מנת שנוכל להשוות ביניהן. בנוסף פותחה מערכת סימולציה בה יכלנו לדמות ריצה של ג׳ינסנג עם אורחים דמיוניים. יכלנו לבדוק איך פרמטרים שונים משפיעים על האלגוריתם של ג׳ינסנג, ולבחון את אלגוריתם האסטרטגיות של האורחים.

השיטות שהוצגו לשימוש בזכרון דינמי בענן הוראו כיעילות, שינויי זכרון גדולים ומיידיים הוצלחו להתקבל ללא העברת דפים לכונן, ותגובת האפליקציות להגדלת הזכרון נמדדה בשניות ספורות, פחות מ 10 שניות לג׳יגה בייט עד לקבלת ביצועים מקסימליים ביישום Memory Consumer, וכ 100 שניות לג׳יגה בייט לקבלת ביצועים מקסימליים ב memcached. שני היישומים הצליחו לפנות ג׳יגה בייט של זכרון בפחות מ 5 שניות.

בתוצאות הניסויים ג׳ינסנג הציגה שיפור של $6.2\times$-$31.5\times$ עבור הרווחה החברתית הכוללת, או $1.6\times$ עבור הביצועים, בהשוואה לגישות העדכניות ביותר לניהול הקצאת הזכרון בענן. בנוסף היא השיגה 83%-100% מהרווחה החברתית האופטימלית האפשרית.

נבחנו שני מנגנונים נוספים לצמצום העברת הזכרון. האחד הוא פרמטר בשם reclaim factor. פרמטר זה קובע כמה מהזכרון שנוסף לאורח יוחזר למארח לאחר סיום תקופת המכרז. קביעת ערך נמוך משאירה חלק מהזכרון זמן רב יותר אצל הלקוח. לאחר בחינה של המנגנון נרה כי הוא מכניס רעשים לתוצאות, ויעילותו לא ברורה. השני שנבחן הוא מנגנון בו המארח מוכן להקטין את הרווחה החברתית בגלל המחיר שהיא עולה מבחינת העברת הזכרון בין לקוח אחד לאחר. בגלל שהעברת הזכרון היא בעלת עלות כלשהי אנו דורשים ששינוי הזכרון יתבצע רק אם הרווחה החברתית שלו גבוהה באחוז כלשהו מאשר הרווחה החברתית המתקבלת מהשארת ההקצאה הקודמת, וזאת במידה וההצעות שהתקבלו אינן שונות באחוז כלשהו מההצעות הקודמות שהתקבלו. על ידי הכנסת המנגנון צומצמה העברת זכרון תכופה בין לקוחות דומים וביצועי המערכת שופרו. עם זאת, המנגנון מבטל את תכונת ה incentive compatibility, ואם היה נבחן עם לקוחות שאינם בהכרח אומרים את האמת, התוצאות היו יכולות להיותרעות במיוחד.

במחקר שולבה אסטרטגיית שחקן המשתתף במכרז. השחקן צריך לדעת להעריך את הביצועים שלו כפוקנציה של הזכרון שעומד לרשותו, ומהערכת שווי הביצועים הוא יכול להעריך את שווי הזכרון. בנוסף, השחקן יכול להעריך את הרווח מאותם אזורי זכרון. השחקן המשתתף במשחק מציע מחיר יחידה אותו הוא מוכן לשלם, ואזורי זכרון אותם הוא מוכן לרכוש במחיר זה. המחיר אותו הוא מציע הוא ערכו האמיתי של הזכרון משום שזו האסטרטגיה הדומיננטית עבור כמות זכרון זו. לעומת זאת, המשתתף יכול להציע מחירי יחידה שונים עבור בחירה שונה של אזורי זכרון. אנו מציגים אסטרטגיה הפוסלת את ההצעות שסיכויי הקבלה שלהם נמוכים, ומן האסטרטגיות בעלות הסיכויים הטובים נבחרת האסטרטגיה בעלת הערכת הרווח הגדול ביותר.

בנוסף, במחקר זה מוצגת גישה לשימוש נכון במחשב ענן המשנה באופן דינמי את הקצאות הזכרון של הלקוחות. אנו מציגים סוג חדש של יישומים: יישומי זכרון דינמי (dynamic memory applications). יישומים אלו מסוגלים לשפר את ביצועיהם כאשר לשרותם כמות גדולה יותר של זכרון, ובנוסף מסוגלים לשחררו במהירות בשעת הצורך. אנו רואים בסוג יישומים אלו כהכרחיים לסביבת ענן בו הקצאת הזכרון משתנה. במהלך המחקר פותח יישום בדיקה ייחודי, Memory Consumer. יישום זה מנסה לכתוב לכתובות זכרון באופן שוטף, אך מבצע פעולת כתיבה רק אם הזכרון באמת קיים לרשותו, כלומר, אם לאורח יש יותר זכרון, היישום יצליח לכתוב ליותר כתובות ומכיוון שקצב הכתיבות הוא המדד לביצועיו, ביצועיו ישתפרו. יישום זה מתאפיין כיישום זכרון משתנה. בנוסף מוצג שינוי של יישום הנקרא Memcached ונפוץ מאוד בתור מטמון העומד לפני מאגרי מידע. היישום המקורי הוא בעל מטמון בגודל קבוע ואינו מתאים לשימוש בתוך אורח בעל זכרון משתנה. כשגודל הזכרון של האורח יקטן מתחת לגודל זכרון המטמון, האורח יצטרך לפנות לדיסק עבור קריאות או כתיבות והביצועים ייצנחו. בנוסף, היישום לא ישפר את ביצועיו עבור תוספת זכרון, משום שהמטמון הוא בגודל קבוע. אנו מציגים שינוי של היישום כך שגודל המטמון יכול להשתנות. באמצעות יישום נוסף המבקר את גודל היישום הדינאמי אנו מצליחים לשנות את גודלו של המטמון בהתאם לזכרון הקיים אצל האורח. היישום הדינאמי משנה את גודל המטמון מספיק מהר ומראה שיפור בביצועים עבור תוספת זכרון. שני היישומים שפותחו נבדקו במהלך המחקר והוכחו בתור יעילים בסביבת זכרון דינמית.

שיטה נוספת המוצגת במחקר היא התראה מפורשת מוקדמת לאורחים במחשב הענן על שינוי קרב בהקצאת הזכרון, בטרם יישומו בפועל. שיטה זו ניצבת אל מול שיטות קודמות בהם לקיחת זכרון מן האורח מבוצעת בצורה הדרגתית. בשיטות אלו לקיחת הזכרון היתה מרומזת לאורח באופן בלתי

תקציר

זיכרון פיזי הוא המשאב היקר ביותר במערכות מחשוב העננים של ימנו. ספקי מחשוב העננים היו מעוניינים להביא את הרווחה החברתית (Social welfare) של לקוחותיהם לערך מירבי, על ידי הקצאת הזיכרון ללקוחות שהכי מעריכים אותו. אבל בעולם האמיתי, הלקוחות הללו אנוכיים: הם יגלו לספק את ערכו האמיתי של הזכרון כאשר זהו האינטרסט האישי שלהם. בתנאים כאלה, כיצד ספקי מחשוב העננים יכולים למצוא הקצאת זכרון שמגדילה את הרווחה החברתית של לקוחותיהם?

מחקר זה מציג את ג׳ינסנג (Ginseng), מערכת ניהול הקצאת זכרון דינמית מונחה שוק, הראשונה שמסוגלת להתמודד עם לקוחות אנוכיים. ג׳ינסנג גורמת גם ללקוחות אנוכיים לבקש זכרון רק כשהם צריכים אותו ולהציע את ערכו האמיתי. בנוסף היא מותאמת למכרז של זכרון בכך שהיא מצמצת העברת זכרון מיד ליד כאשר הדבר אינו נחוץ. ג׳ינסנג משנה את הקצאות האורחים באופן דינמי, ואינה מקצה־ייתר (Overcommited) אותו.

מערכת המכרז של ג׳ינסנג שייכת לקבוצת המכניזמים של affine maximizers , ודומה למכרז VCG. במהלך המשחק, ג׳ינסנג מקצה לכל אורח כמות זכרון מינימאלית וקבועה עליה הוא משלם מחיר קבוע. אחת לזמן קצוב, היא מציעה למכרז את הזכרון הפנוי. לאחר מכן, הלקוחות מציעים עבור הזכרון מחיר ליחידת זכרון ליחידת זמן אותו הם מוכנים לשלם, ואזורי זכרון אותם הם מוכנים לרכוש במחיר זה. פרוטוקול זה, מאפשר להתגבר על פונקציות הערכת זכרון קמורות ואפילו לא מונוטוניות עולות. לאחר איסוף ההצעות, ג׳ינסנג מוצאת הקצאת זכרון המביאה לרווחה החברתית המירבית של הלקוחות, על ידי אלגוריתם רקורסיבי הפורש עץ 2-tree. התשלום הנגבה מן הלקוחות נקבע לפי הנזק הנגרם ללקוחות האחרים מנצחונם במכרז.

בנוסף, הוספנו מספר מנגנונים לג׳ינסנג, להתאמתו למכרז של זכרון ולמזעור העברתו מיד ליד לעיתים תכופות. מנגנון ראשון, הוספנו מנגנון לשבירת שוויון כאשר הוצעו שתי הצעות בעלות מחיר יחידה זהה. שבירת השוויון הראשונה היא על ידי כמות הזכרון האחרונה שהוקצתה לאורחים אלו; לזה שהיה יותר זכרון תהיה עדיפות על השני. בכך, שוב אנו מצמצמים את מעברי הזכרון מיד ליד. במקרה שההקצאה האחרונה היתה גם היא זהה, אנו שוברים את השוויון בצורה רנדומאלית.

מנגנון שני, הוא איבר ענישה על העברת זכרון המהווה חלק מפונקציית המחיר החברתי (social cost), הכללה של התועלת החבתית המוגדרת במשפחת המכניזמים של ה affine maximizers. איבר זה פרופורציונאלי לכמות הזכרון שעוברת מיד ליד ומקטין את המחיר החברתי ככל שכמות הזכרון העובר גדלה. בכך הוא גורם להעדפת הקצאות בהם כמות הזכרון העובר היא קטנה יותר. איבר זה משאיר את המכניזם בקטגוריית ה affine maximizers, ובכך מבטיח את תכונת ה incentive compatibility לפיה האינטרסט של השחקנים המשתתפים הוא להמר עם הערך האמיתי של הזכרון.

המחקר בוצע בהנחייתו של פרופסור אסף שוסטר, בפקולטה למדעי המחשב.

תודות

ברצוני רוצה להודות לאנשים הבאים:

למנחה שלי, פרופסור אסף שוסטר, שנתן לי את ההזדמנות לעבור על המחקר המעניין, על ההנחיה במהלך המחקר, ועל כך שנתן את התמיכה הדרושה.

לאורנה אגמון בן יהודה, על העזרה והתמיכה שניתנה בכדי להתגבר על מכשולים, ועל ההדרכה בעולם תורת המשחקים.

למשפחתי הטובה והתומכת, שתמיד נותנים עצות טובות.

לחברתי, מריה, על הבנת הקשיים שחוויתי, על הזמן והמרחב שידעה לתת לי כשהייתי צריך, על שמירת השפיות שלי, והתמיכה בזמנים קשים.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

הקצאה דינמית של זכרון במחשבי ענן באמצעות מכרז מחיר שני

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במערכות מחשב

אייל פוזנר

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
תמוז התשע"ג   חיפה   יוני 2013

הקצאה דינמית של זכרון במחשבי ענן באמצעות מכרז מחיר שני

אייל פוזנר