Ginseng: Market-Driven Memory Allocation

Orna Agmon Ben-Yehuda[1] Eyal Posener [1] Muli Ben-Yehuda[1,2] Assaf Schuster[1] Ahuva Mu'alem[1,3]

[1]Technion – Israel Institute of Technology [2]Stratoscale [3]Ort Braude

{ladypine, posener, muli, assaf, ahumu}@cs.technion.ac.il

Abstract

Physical memory is the scarcest 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. How can real-world cloud providers allocate memory efficiently to those (selfish) clients who value it the most?

We present *Ginseng*, the first market-driven cloud system that allocates memory efficiently 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. Ginseng achieves a 6.2×–15.8× improvement (83%–100% of the optimum) in aggregate client satisfaction when compared with state-of-the-art approaches for cloud memory allocation.

Categories and Subject Descriptors D.4.2 Operating Systems [*Storage Management*]: Main memory

Keywords KVM; Memory Overcommitment

1. Introduction

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources to their clients. As competition between providers gets tougher and prices decrease, 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 cloud computing platforms today [12, 15, 17, 25, 27, 37]. Google, for example, had to begin charging for memory usage in addition to CPU usage: not charging for memory made the scaling of applications that use a lot of memory and little CPU time "cost-prohibitive to Google." [8]. Other platforms (such as Amazon EC2) offered virtual machines with varying amounts of memory to begin with, thereby charging clients for memory usage in addition to CPU and I/O usage. In general, today's cloud computing clients buy a supposedly-fixed amount of physical memory for the lifetime of their guests.

Providers can greatly reduce their expenses by using less memory to run more client guest virtual machines on the same physical hosts. This can be done transparently by means of memory overcommitment [23, 37]. When memory is overcommited, the clients have no way to discern how much physical memory they are actually getting. Due to the lack of transparency and difficulties with providing a given level of quality of service when overcommitting memory, some providers refrain from memory overcommitment and let their hardware go underutilized. Others simply reduce their clients' quality of service.

Clients would much prefer to have full visibility and control over the resources they receive [2, 28]. They would like to pay only for the physical memory they need, when they need it [3, 11]. By granting clients this flexibility, providers can increase client satisfaction: clients interested in high quality of service (QoS) will be able to choose a non-overcommited machine, while budget-conscious clients will be able to enjoy the cloud at low prices when demand is low. 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—yields benefits for clients, whose satisfaction is improved, and providers, whose hardware utilization is improved.

Previous physical memory allocation schemes assumed that guest virtual machines are *white boxes*: that they are run by fully cooperative clients, who let the host know precisely what each guest is doing, how well it performs, how much benefit additional memory would bring to it, and the importance of the workload to the client [12, 15, 17, 27]. These systems allocated memory efficiently and improved the overall system's performance, but were unsuitable for real-world commercial clouds, because the assumption that the host has full, accurate information on all aspects of guest performance is unrealistic.

As recent commercial cloud trends of price dynamicity and fine-grained resource granularity [2] indicate, real-world cloud 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

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2764-0 /14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576197

not share this information with their provider unless it is in their own best interest to do so.

When the host allocates memory solely according to guest-provided input, rational and selfish 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.

Alternatively, the host can allocate memory according to passive *black-box* measurements taken outside of the guests [18, 25, 37, 39, 40]: for example, by monitoring I/O and inferring major page faults [18], or by monitoring use of physical pages to balance the guests' need for physical memory [39]. However, in such cases the guests have an incentive to bias the measurement results, e.g., by inducing unnecessary page faults or accessing unnecessary memory. Furthermore, such passive measurements can only compare the guests by externally visible metrics such as throughput and latency, which are valued differently by different guests under different circumstances.

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?

Our first contribution towards solving this problem is the **Memory Progressive Second Price (MPSP) auction**, a game-theoretic market-driven mechanism which 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).

Our second contribution is **Ginseng** itself, a market-driven cloud system for allocating memory efficiently to selfish black-box virtual machines. It is the first full implementation of a single-resource Resource-as-a-Service (RaaS) cloud [2]. Ginseng is the first cloud platform to optimize overall client satisfaction for black box guests. 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. We also build a strategic agent for the MPSP auction.

Ginseng supports static-memory applications—legacy applications that require some fixed quantity of memory and do not perform better with more memory, but is tailored for *elastic-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. Elastic-memory applications are becoming more common thanks to platforms that facilitate their development, such as Salomie et al.'s database [29], Java runtime with balloons [12, 17, 29], CRAMM [38], or dynamic heap adjustment for garbage-collected environments [14, 16]. In addition, applications designed for the Linux mempressure control group are elastic by design. **Our third contribution** is two elastic-memory benchmark applications: an **elastic-memory version of Memcached**, a widely-used key-value cloud application, and **MemoryConsumer**, an elastic memory benchmark we developed.

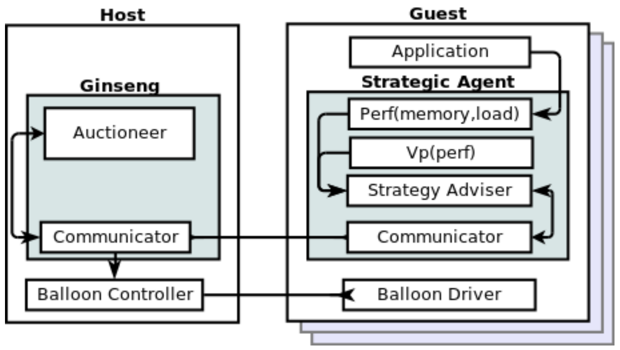


Figure 1. Ginseng system architecture

Ginseng achieves a $6.2\times$ improvement in aggregate client satisfaction for MemoryConsumer and $15.8\times$ improvement for Memcached, when compared with state-of-the-art approaches for cloud memory allocation. Overall, it achieves 83%–100% of the optimal aggregate client satisfaction.

2. System Architecture

Ginseng is a market-driven cloud system that allocates memory to guests using guest bids for memory. It is implemented for cloud hosts running the KVM hypervisor [20] with Litke's memory overcommit manager MOM [23]. It controls the exact amount of physical memory allocated to each guest via libvirt using *balloon drivers* [37]. The *balloon driver* is installed in the guest operating system. The host's *balloon controller* controls the balloon driver, inflating or deflating it. When inflating, the balloon driver allocates memory from the guest OS and pins it, so that the guest OS won't attempt to swap it out; the balloon driver then transfers this memory to the host. When deflating, the balloon driver frees memory back to its OS, in effect giving the OS more memory from the host. Libvirt supplies an API to balloon drivers in different hypervisors, improving portability.

Ginseng has a host component and a guest component, as depicted in Figure 1. The host component includes the *Auctioneer*, which runs the MPSP auction. The auctioneer's *communicators* communicates asynchronously with the guests'*communicators* according to the auction's protocol specified in **Section 4**. The host's communicator also instructs the balloon controller how to allocate memory between guests. The balloon controller inflates and deflates the balloon drivers inside the guests. The guest's economic learning agent acts on behalf of the client. The *strategy adviser* is the agent's brains. Our implementation of an adviser is described in **Section 6**, but the client is free to choose a different logic.

3. Memory Auctions

Ginseng allocates memory efficiently because its guests bid for the memory they want in a specially-designed auction that the host conducts in quick rounds. We begin by supplying the background to the auction, whose protocol is defined in Section 4.

In Ginseng, each guest has a different, changing, private (secret) *valuation* for memory. This valuation reflects how much additional memory is worth to each guest. We define the aggregate benefit of a memory allocation to all guests—their satisfaction from auction results—using the game-theoretic measure of *social welfare*. The social welfare of an allocation is defined as the sum of all the guests' valuations of the memory they receive in this allocation. An efficient memory auction allocates the memory to the guests such that social welfare—guest satisfaction—is maximized.

VCG [7, 13, 34] 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. VCG auctions do so by charging each participant 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 [24].

Various auction mechanisms, some of which resemble the VCG family, have been proposed for *divisible* resources, in particular for *bandwidth sharing* [19, 21, 26]. For practical reasons, bidders in these 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 [21]. None of the auctions proposed so far for divisible goods are suitable for auctioning memory, because memory has two characteristics that set it apart from other divisible resources. First, transferring memory too quickly between two participants leads to waste; Second, the participants' valuation functions might not be concave; that is, the law of diminishing marginal utility might not always apply to memory, e.g., as in Figure 2b. However, the PSP auction optimally allocates a divisible resource if and only if all the valuation functions are monotonically rising and concave. Other bandwidth auctions also rely on the monotonically rising concave property of the valuation functions.

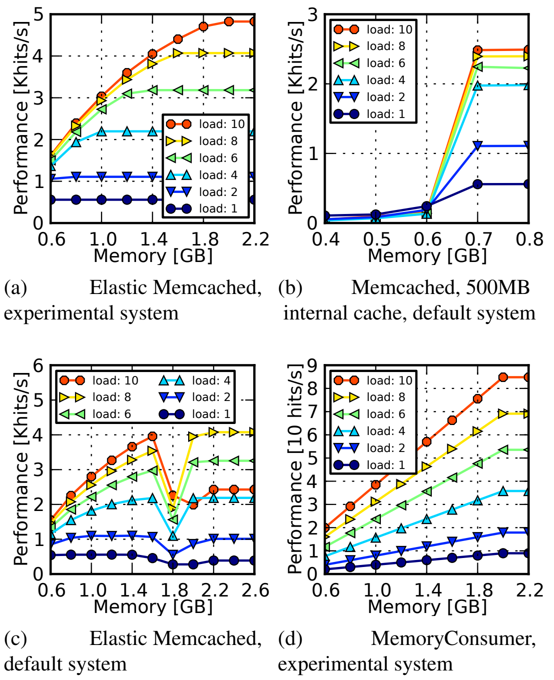
The *memory valuation function*, which describes how much the guest is willing to pay for different memory quantities, depends on the load the guest is under, the performance gain or loss it expects from more memory given that load, and the value of performance to the guest. Formally, it is $V(\textit{mem}, \textit{load}) = V_{perf}(\textit{perf}(\textit{mem}, \textit{load}))$, where $V_{perf}(\textit{perf})$ refers to the valuation of performance as described below, and $\textit{perf}(\textit{mem}, \textit{load})$ describes the performance the guest can achieve given a certain load and a certain memory quantity.

Performance might be measured in page hits per second for a webserver, "get" hits per second for a caching service, transactions per second for a database, trades per second for a high-frequency trading system, or any other guest-specific metric. For our experiments, an offline mapping of performance as a function of memory and load (as done by Hines et al. [17] and Gordon et al. [12]) was accurate enough, as we demonstrate in Section 8.2. However, real-world performance may depend on many variable conditions. To this end, performance can be measured online as several works demonstrate [39–41]. An important feature of the MPSP auction is that it does not require the guest to have its performance defined for any memory value. Hence, the guest can keep a moving window of its latest performance measurements, which reflect best the current conditions under which it operates.

The guest's owner's (i.e., the client's) valuation of performance function, $V_{perf}(\textit{perf})$, describes the value the client derives from a given level of performance from a given guest. This client-specific function is private information of each client. It is based on economic considerations and business logic.

For example, an e-commerce website that typically makes \$100 sales and needs to display $10,000$ web pages on average to generate a single sale might measure its performance in displayed pages per second, and value each displayed page at \$0.01. For this client, $V_{perf}(\textit{perf}) = \frac{\$0.01}{page} \cdot \textit{perf}$. Another client might require the same average number of displayed pages to make a sale, but its typical sale would be \$10 only. For this client, $V_{perf}(\textit{perf}) = \frac{\$0.001}{page} \cdot \textit{perf}$. Both clients will need to know $\textit{perf}(\textit{mem}, \textit{load})$: how many pages they can display per second when given various amounts of memory and under the current conditions (e.g., load).

If either of these functions is non-concave or not monotonically rising, the composed function may be non-concave or not monotonically rising as well. Indeed, guest performance $\textit{perf}(\textit{mem}, \textit{load})$ is not necessarily a concave, monotonically rising function of physical memory. For example, the performance graph of off-the-shelf memcached in our experimental environment is monotonically rising, but not concave (Figure 2b). This non-concave function resembles a step function, and is typical of the operating system's efforts to handle memory pressure through swapping. Non-concavity may also result from differences in the size and frequency of use of various working sets, swapping policies, or garbage collection operations [31]. Our *elastic memcached*, in contrast, has a concave, monotonically rising performance graph in the same experimental environment



(a) Elastic Memcached, experimental system

(b) Memcached, 500MB internal cache, default system

(c) Elastic Memcached, default system

(d) MemoryConsumer, experimental system

Figure 2. Application performance (“get” hits per second for Memcached, hits per second for MemoryConsumer) as a function of guest physical memory, for different load values. The load is defined as the number of concurrent requests made to the application.

(Figure 2a). However, in a default system configuration, its performance graph is neither concave nor monotonically rising (Figure 2c), due to a network bottleneck that was prevented in the experimental environment. This bottleneck is an example of a problem that a real production system might encounter. It cannot fine-tune its setup parameters and redesign its software on-the-fly; it has to make do with what it measures. Ginseng is designed to support such ad hoc, real-life valuation-of-memory graphs that are neither concave nor monotonically rising.

Auction protocols that assume monotonically rising concave valuation functions either interpret a bid of unit **p**rice and **q**uantity (p, q) as willingness to buy exactly q units 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.

MPSP supports monotonically rising concave memory valuation functions in the same way that the PSP auction supports them. In addition, it supports non-concave and non-monotonic valuation functions by specifying *forbidden ranges*. These are forbidden memory-quantity ranges for a single price bid. The guest can use forbidden ranges to cover domains in which its average valuation per memory unit is lower than its bid price. By definition, MPSP will not allocate the guest a memory quantity within its forbidden ranges. Rather, it will optimize the allocation given the constraints. The guest can thus avoid getting certain memory quantities in advance while still expressing a variety of desired quantities. The forbidden ranges are designed to efficiently convey information about functions which are concave, monotonically rising in separate ranges. However, the terminology does not restrict the guest valuation functions in any way. In particular, the guest can bid for a specific desired point (p, q) by setting the open range $(0, q)$ as a forbidden range.

4. MPSP: Repeated Auction Protocol

In Ginseng, each guest has some permanent *base memory*. Guests pay a constant hourly fee for their base memory, and it is theirs to keep as long as they run. In each auction round, each guest can bid for extra memory. Ginseng calculates a new memory allocation after every auction round and guests rent the extra memory they won. In the next auction round the same memory will be put up for auction again.

The constant fees for base memory are designed to provide the lion’s share of the host’s revenue from memory, such that the host can afford to rent the extra memory for the sole purpose of optimizing social welfare, thereby attracting more guests. The price of base memory is not affected by the prices paid for extra memory.

Ultra high-end clients with hard QoS requirements are expected to pre-pay for all the memory they need in advance, to ensure that they always get the resources they need. Ultra low-end clients are expected to pre-pay only for as much memory as they need to operate the guest OS and limit their bids, so that they can temporarily rent additional resources later while staying within their budget. The clients spanning the range between those extremities are expected to choose a flexible deal according to their needs.

Here we describe one MPSP auction round, accompanied by a numeric example.

Initialization. Each guest i is set up with its *base memory* as it enters the system. For example, guest 1 runs memcached and pre-pays for 1.4GB, while guest 2 runs MemoryConsumer and pre-pays for 0.6GB.

Auction Announcement. The host computes the *free memory*—the maximal amount of memory each guest can bid for—as the *excess* physical memory beyond the amount of memory in use by the host and the sum of base memories. It then informs each guest of the free memory and the auction’s closing time, after which bids are ignored. In the example, the machine has 4GB. The host uses 1.6GB, and the guests pre-paid for 2GB, so the host announces an auction for 0.4MB.

Bidding. Interested guests bid for memory. Agent i’s *bid* is composed of a *unit price* p_i—memory price per GB 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, sorted in ascending order. We denote the desired ranges by $[r_j, q_j]$ for $j = 1 \dots m_i$, where r and q stand for restriction and quantity. The bid means that the guest is willing to rent any memory quantity within the desired range list for a unit price p_i.

In the example, both guests experience a load of 10 concurrent requests. Guest 1 values its performance at \$1 per Khit/second, and bids \$1 per GB of memory per second $(p = 1\frac{\$}{GBs})$ for any amount of memory between 0 and 0.4GB $(r_1 = 0, q_1 = 0.4GB)$, on the basis of the performance data in Figure 2a. Guest 2 values its performance at \$0.1 per hit/second, and bids \$5 per GB of memory per second for the same amount of memory $(p = 5\frac{\$}{GBs}, r_1 = 0, q_1 = 0.4GB)$, on the basis of the performance data in Figure 2d.

Bid Collection. The host asynchronously collects guest bids. It considers the most recent bid from each guest, dismissing bids received before the auction round was announced. Guests that did not bid lose the auction automatically, and are left with their base memory.

Allocation and Payments. The host computes the allocation and payments according to the MPSP auction protocol described in Section 5. For each guest i, it computes how much memory it won (denoted by ${q'}_i$) and at what unit price (denoted by ${p'}_i$). The payment rule guarantees that the price the guest will pay is less than or equal to the unit price it bid. The guest's account is charged accordingly. In the example, guest 1 loses $(p'_1 = 0, q'_1 = 0)$, and guest 2 wins all of the free memory $(p'_2 = 1\frac{\$}{GBs}, q'_2 = 0.4GB)$.

Informing Guests. The host informs each guest i of its personal results p'_i, q'_i. The host also announces *borderline bids*: the lowest accepted bid's unit-price and the highest rejected bid's unit-price ($5\frac{\$}{GBs}$ and $1\frac{\$}{GBs}$ in the example, respectively). This is information that guests can work out on their own; having the host supply it makes for a more efficient system. The guests use this information in on-line algorithms that decide how much to bid in future rounds, as described in Section 6.

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, by inflating and deflating their balloons as necessary. The purpose of the adjustment 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 memory consumption, 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. Which applications to notify and when to notify them is left to the guest's agent. In the absence of elastic applications, it is left to the guest kernel to deal with memory pressure, e.g., by shrinking internal caches.

5. MPSP: Auction Rules

Every auction has an allocation rule—who gets the goods?—and a payment rule—how much do they pay? To decide who gets the goods, the MPSP auction determines the optimal allocation of memory. This is the allocation that maximizes social welfare—client satisfaction—as described in Section 3. To determine the optimal allocation, the MPSP auction solves a constrained divisible good allocation problem, as detailed in Section 5.1. To determine how much they pay, the MPSP auction takes into account the damage they inflict on other guests, as detailed in Section 5.2. After explaining the rules we discuss their run-time complexity and provide an example for executing them. A correctness proof is also available but has been omitted for brevity.

5.1 Allocation Rule

Ginseng finds the optimal allocation using a constrained divisible-good allocation algorithm, which works in stages as described below. In each stage, Ginseng attempts a divisible good allocation by sorting the guests lexically, first by their bid unit-price, second (to break ties) by their current holdings, and last by a random shuffle. Preferring the current holder when breaking ties reduces memory waste due to back-and-forth transfers of memory between guests [39]. It also reduces the waste in comparison to a single PSP auction, in which tied guests are excluded from the allocation. Ginseng then allocates each guest its maximal desired quantities according to this order.

If there are a guest g and a forbidden range R such that g ends up with a memory quantity inside R, then the allocation is *invalid*. This can happen if g is the last guest allocated some memory and there is not enough memory left to fulfill g's entire request. Ginseng examines the social welfare of such invalid allocations. If such an invalid allocation gives a higher social welfare than the highest social welfare seen to date in a valid allocation, then Ginseng considers two constrained allocations instead of the invalid one. In the first, guest g gets a memory quantity large enough to cover all of R. In the second, g gets a memory quantity small enough such that none of it is in R. The social welfare of the *valid* allocations is compared to find the optimal allocation.

5.2 Payment Rule

The payments follow the *exclusion compensation* principle, as formulated by Lazar and Semret [21]. Let ${q''}_k$ denote the memory that would have been allocated to guest k in an auction in which guest i does not participate and the rest of the guests bid as they bid in the current auction. Then guest i is charged a unit price p'_i, which is computed as follows:

$$p'_i = \frac{1}{q'_i} \sum_{k \neq i} p_k \left(q''_k - q'_k \right) . \quad (1)$$

According to this payment rule, when guest i is charged $p'_i q'_i$, it actually pays for the damage its bid inflicted on the other guests. We note that to compute the payment for a guest that gets allocated some memory, the constrained divisible good allocation algorithm needs to be computed again without this guest. In total, the allocation procedure needs to be called one time more than the number of winning guests.

5.3 Complexity

The problem that the MPSP algorithm solves—finding the memory allocation that maximizes the social welfare function—is defined over the domain of memory quantities that guests agree to rent. This domain is not convex because the forbidden ranges create "holes" in it. Maximizing a function over a non-convex domain is at least as hard as the knapsack problem, and therefore NP-hard. In the worst case the algorithm needs to compute the social welfare which results from each forbidden range being completely allocated or completely denied: 2^M different divisible allocations, where M is the number of all the forbidden ranges in all the bids. Each such allocation takes $O(N)$ to compute, where N is the number of bids, and each payment rule requires $O(N)$ allocations to be computed. Hence, the time complexity of MPSP is $O(N^2 \cdot 2^M)$.

Nevertheless, for real life performance functions, a few forbidden ranges should be enough to cover the non-concave regions. We observed one forbidden range for off-the-shelf memcached and zero forbidden ranges for elastic-memory applications. Given the relatively small number of guests on a physical machine, the algorithm's run-time is reasonable: we observed less than one second using a single hardware thread, even in experiments with 23 guests.

6. Guest Strategy

So far, we discussed the Ginseng system's architecture, and the MPSP memory auction from the auctioneer's point of view. But what should guests who participate in MPSP auctions do? How much memory should they bid for and how much should they offer to pay? In an exact VCG auction, the guests can inform the host about their valuation for different memory quantities. However, the reduced MPSP bidding language lightens the computational burden on the host and leaves the choice of memory quantity with the guest. An intermediate approach—the multi-bid auctions—is discussed in Section 10.

In this section we present the bidding strategy we developed. It is used by the guests in the performance evaluation in Section 8. Our guest wishes to maximize the utility it estimates it will derive from the next auction. This is a natural class of bidding strategies in ad auctions [5].

Our guest needs to decide how much memory to bid for, and at what price. We show in Section 6.1 that for any memory quantity, the best strategy for the guest would be to bid its true valuation for that quantity. To choose the maximal quantity it wants to bid for, the guest compares its estimated utility from bidding for the different quantities, as described in Section 6.2.

6.1 Choosing a Bid Price

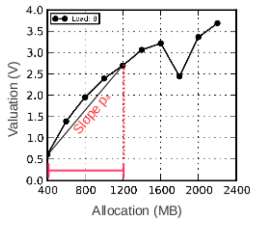
In this subsection we assume the guest has decided how much memory (q_m, or q for short) it wants to bid for and show how much it should bid for it. For the simple case of an exact desired memory quantity ($m = 1$, $r_m = q_m = q$), for any value q, bidding the mean unit valuation of the desired quantity (the slope $s(q) = \frac{V(base+q)-V(base)}{q}$) is the best strategy, no matter what the other guests do. By bidding lower than $p = s(q)$, the guest risks losing the auction; by bidding higher it risks operating at a loss (paying more than what it thinks the memory is worth).

For less simple cases when the guest bids for a range of memory quantities up to q, if the valuation function is (at least locally, in the range up to q) concave monotonically rising, bidding $p = s(q)$ is still the best strategy for q regardless of other guests' bids: $s(q)$ is the guest's minimal valuation for the range because the unit valuation drops with the quantity. See, for example, Figure 3a, where the valuation function is above the line connecting the valuation of 1200 MB with the base (400 MB) valuation. Since the connecting line's slope is the mean unit valuation of 1200MB, any point above the line is a point whose mean unit valuation is higher than the mean unit valuation of 1200MB.

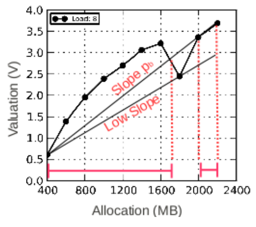
In the remaining cases, the valuation function is non-concave or not even monotonically rising. In such functions, the mean unit-valuation $s(q)$ may rise locally with quantity: in Figure 3b, for example, $s(2200MB) > s(1800MB)$. This means that simply bidding for 2200MB with a unit-price of $s(2200MB)$ may result in getting a memory quantity for which the guest is not willing to pay that much. The guest can avoid getting quantities for which the mean unit valuation is lower than its bid price by excluding those quantities from its bid using the forbidden ranges mechanism. In this example, the guest uses a forbidden range to exclude the quantities $[1700, 2000]$ MBs of memory from its range, since it is not economical to bid for them with a unit price of $s(2200MB)$.

The forbidden ranges mechanism allows the guest to bid $s(q)$ without the risk of operating at a loss. However, the guest may have something to gain by bidding with a unit-price that is less than $s(q)$. If the guest does not get the maximal memory quantity it bid for, it can try exploring its strategy space. It can retain q, lower the bid price, and decrease the forbidden ranges. Thus the guest allows the host to give it a partial allocation in more cases, when the alternative might be not getting any memory at all. In Figure 3b, the lowest bid-price worth exploring is labeled the *low slope*: it eliminates the need for forbidden ranges.

When the auction has reached a *steady state*—when a guest's won goods and payment turn out the same in subsequent auctions in response to the same strategy—the guest already knows how much memory it can get for any bid. The



(a) Single range



(b) Multiple range

Figure 3. Strategies for choice of unit price for two maximal quantities, using the same valuation function. Figure 3a demonstrates a single desired range strategy for a concave monotonically rising part of the valuation function. Figure 3b demonstrates a multiple desired range strategy for a non-concave, not even monotonically rising part of the valuation function.

guest is incentivized to raise its bid price to a maximum, thus increasing the exclusion compensation that other guests pay and making them more considerate. Hence, our guests always bid $s(q_m)$.

6.2 Choosing a Maximal Memory Quantity q_m

To maximize the guest's estimated utility from the next auction, the guest chooses q_m. Our guest assumes it is in a steady state, and estimates its utility using past auction results. The guest assumes, for simplicity, that it will get a memory quantity of q_m if $p > p_{min}$, and 0 otherwise. p_{min} is the lowest price the guest can offer and still have a chance of getting any memory at all. It is evaluated on the basis of ten recent borderline bids that are announced by the host.

The utility estimation also requires an estimation of the unit price to be paid for the allotted memory amount. The guest estimates its utility from bidding (p, q_m) by dividing it to two components: (1) a valuation improvement from winning the memory it expects to win and (2) a charge. For concave valuation functions $V(\cdot)$, the estimated utility is maximized when $s(q_m) = p_{min}$. In such cases, the guest need only estimate and predict p_{min} to bid optimally. For other (non-concave) functions, the guest must evaluate the estimated charge to find the memory quantity that maximizes the estimated utility To this end it assesses the unit price it will pay on the basis of a historical table of (p', q') pairs, and further bounds it from above by the highest losing bid price in the last auction round. If several values of q_m maximize the estimated utility, the guest prefers to bid with higher p values, to improve its chances of winning the auction.

7. Experimental Setup

In this section we describe the experimental setup in which we evaluate Ginseng.

Alternative Memory Allocation Methods. We compared Ginseng with memory overcommitment and allocation methods available to commercial IaaS providers: *static*, *host-swapping* and *MOM*. In the *static* method, each guest is allocated a fixed amount of memory without any overcommitment. This is a common method in public clouds. When relying on *host-swapping*, each guest gets a fixed memory quantity regardless of the number of guests, and the host is allowed to swap guest memory to balance memory between guests as it sees fit. This method is the fallback of many overcommitment methods. The *Memory Overcommitment Manager (MOM)* [23] collects data from its guests to learn about their memory pressure and continuously adjusts their balloon sizes to make the guests feel the same memory pressure as the host. This is a state-of-the-art overcommitment method that is freely available, but it is not a black-box method: it relies on probes inside the guests and can be easily circumvented by a malicious one.

Workloads. To experiment with overcommitment tradeoffs, we used benchmarks of *elastic 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. We experimented with a modified *elastic memcached* and with *MemoryConsumer*, a dedicated dynamic memory benchmark. Both applications interacted with the Ginseng guest agent to dynamically adjust their heap sizes when they won or lost memory: the Ginseng agent informed the application of the upcoming change and the application reacted by reducing its working-set size accordingly, so that the system would not run out of memory when the balloon is inflated.

Elastic memcached is a version of memcached that changes its heap size on-the-fly to respond to guest memory changes. It can free the less-needed internal-cache slabs or alternatively increase its internal cache size. Memcached was driven by a *memslap* client, a standard memcached benchmarking utility. To test a large number of guests quickly, we configured memslap such that memcached's performance graphs saturated at 2GB. To this end we ran memslap with a key size of 249 bytes, a value size of 1024 bytes, a window size of 100K, and a get/set ratio of 30:70, for 200 seconds each time. The application's performance is defined as the "get" hits per second. [1]

MemoryConsumer is an elastic memory benchmark. It tries to write to a random 1MB-sized cell from a predefined range. If the address is within the range of memory currently available to the program, then 1MB 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 0.1 seconds, so that misses cost time. The application's performance is defined as the hits per second. This application is tailored as a pure memory overcommitment benchmark, in order to create clean tests, unhindered by resource bottlenecks other than memory. As with memcached, we chose a range of 1950 cells, so that performance graphs would saturate at 2GB.

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

We profiled the performance of each workload with varying amounts of memory to create its *perf*(mem, *load*) function. We measured performance under different loads for four concurrent guests without memory overcommitment, as also done by Hines et al. [17]. We gradually increased and decreased the physical memory in small steps, waiting in each step for the performance to stabilize. For memcached we measured the performance over 200 seconds, and for MemoryConsumer over 60 seconds. The *perf*(mem, *load*) graphs can be seen in Figure 2a for the elastic Memcached and Figure 2d for MemoryConsumer.

Load. We defined "load" for memcached and MemoryConsumer as the number of concurrent requests. Loads are in the range $[2, 10]$. The total load is always the number of guests $\times 6$, so that the aggregate hits per second of different experiments will be comparable. Each pair of guests exchanged their loads every T_{load}. The load values and their exchange timing were chosen to increase the diversity among the guests, as expected in a real system. Guests were further diversified by assigning them with different memory valuation functions.

Machine Setup. We used a cloud host with 12GB of RAM and two Intel(R) Xeon(R) E5620 CPUs @ 2.40GHz with 12MB LLC. Each CPU had 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`. To reduce measurement noise, we disabled EIST, NUMA, and C-STATE in the BIOS and KSM in the host kernel. To prevent networking bottlenecks, we increased the network buffers. We dedicated hardware thread 0 to the host and pinned the guests to hardware threads $1 \dots N$.

Memory Division. 0.75GB were dedicated to the host. To allow guests to both grow and shrink their memory allocations, we configured all guests with a high *maximal memory* of 10GB, most of which was occupied by balloons, leaving each guest with a smaller *initial memory*. However, when using *host-swapping* and *MOM*, extensive host-swapping caused the host to freeze when the maximal guest memory was set to 10GB. Hence we also created a hinted (*white-box*) version of each of these methods to compare against: we informed the host that the applications actually cannot benefit from the full 10GB, and that a rational guest would only need 2GB. As a result, the provider in the *hinted-MOM* and *hinted-host-swapping* methods configured the guests with at most 2GB. 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 1.

Reducing Guest Swapping. Bare metal operating systems shield applications from memory pressure by paging memory out and by clearing buffers and caches, but elastic-memory applications should be exposed to memory-pressure in order to enable them to respond. To this end we minimized guest swapping by setting `vm.min_free_kbytes` to 0. Note that this did not hinder performance of host-swapping.

Method/Memory (GB)	Initial	Maximal
Ginseng	0.6	10
Static	$11.25/N$	$11.25/N$
Host-swapping	10	10
MOM	0.6	10
Hinted host-swapping	2	2
Hinted MOM	0.6	2

Table 1. Guest configuration: initial and maximal memory values for each overcommitment method. N denotes the number of guests.

Reducing Indirect Overcommitment. Bare metal operating systems keep some memory free, in case of sudden memory pressure. The host can indirectly overcommit such memory by giving it to other guests while it is unused; the host relies on its ability to page out guests if and when sudden memory pressure occurs. Since we focus on direct overcommitment (e.g., using balloons), we made the accounting more accurate by setting `vm.overcommit_memory` to 1 in our guests, thus making the guest physical memory the exact limitation for guest memory allocations. These settings make more sense for a production VM than the default bare-metal OS settings (`vm.overcommit_memory=0`). A VM with default settings would have required and not used 300MB more on our system. These 300MB would only be available for use by other VMs.

Time Scales. Three time scales define 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 (e.g., *cache-warming* time—time for the cache to be filled with relevant data), T_{memory}; the time between auction rounds, $T_{auction}$; a typical time scale in which conditions (e.g., load) change, T_{load}. Useful memory borrowing requires $T_{load} >> T_{memory}$. This condition is also necessary for on-line learning of performance with different memory quantities. To evaluate T_{memory}, we performed large step tests, making abrupt, sizable changes in the physical memory and measuring the time it took the performance to stabilize. We empirically determined good values for T_{load} on the basis of step tests results: 1000 seconds for memcached experiments, whereas for MemoryConsumer 200 seconds are enough. We also used those step tests to verify that major page faults in the guest were insignificant (indicating hardly any guest thrashing), and to verify that there was enough time for the performance measurement method to evaluate the performance. For example, memslap required 200 seconds to start experiencing cache misses.

In realistic setups, providers should set $T_{auction} << T_{load}$, to get a responsive system. 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. We note that due to the long T_{load}, most of the auctions in the experiments did not result in memory changes, and the cache warmth was not affected.

8. Performance Evaluation

This section answers the following questions: (1) Which memory allocation method provides the most satisfied guests (i.e., the highest social welfare)? (2) How accurate is off-line profiling of guest performance?

8.1 Comparing Social Welfare

We evaluate the social welfare achieved by Ginseng vs. each of the five other methods listed in Table 1 for a varying number of guests on the same physical host. We evaluate memcached guests and MemoryConsumer guests in separate sets of experiments. Each Memcached experiment lasted 60 minutes, with $T_{load} = 1000$ seconds. Each MemoryConsumer experiment lasted 30 minutes with $T_{load} = 200$ seconds. For each set we present average results of 5 experiments. Ginseng guests use the strategy described in Section 6.

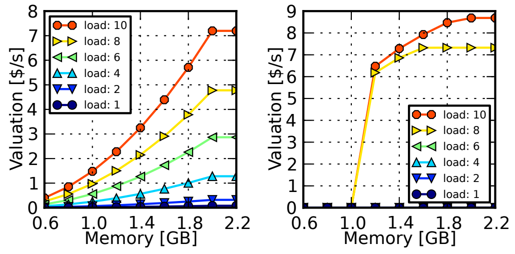
In both benchmarks, *perf*(*mem*) is a concave function. To evaluate Ginseng's abilities over non-concave functions, we used performance valuation functions $V_{perf}(\mathit{perf})$ that make the resulting composed valuation function $V(\mathit{mem})$ non-concave.

In the first experiment set (MemoryConsumer), each guest i's valuation function is defined as $V_i(\mathit{mem}) = f_i \cdot (\mathit{perf}(\mathit{mem}))^2$, where the f_i values were drawn from the *Pareto distribution*, a widely used model for income and asset distributions [22, 32]. We bounded the distribution because on-line trading does not span the whole range of human transactions: some are too cheap or too expensive to be made on-line. We used a reasonable Pareto index for income distributions (1.1) [32], and a lower bound of $10^{-4}\frac{\$}{Khit}$. The "square of performance" valuation function is characteristic of on-line games and social networks, where the memory requirements are proportional to the number of users, and the income is proportional to user interactions, which are proportional to the square of the number of users. The composed valuation function is illustrated in Figure 4a.

In the second experiment set (elastic memcached), each guest i's valuation function is defined as $V(\mathit{mem}) = f_i \cdot \mathit{perf}(\mathit{mem})$, where the f_i values were Pareto-distributed with a Pareto index of 1.36 (an empirical wealth distribution [22]), and bounded in the range $[10^{-4}, 100]\frac{\$}{Khit}$. The highest coefficient was set as:

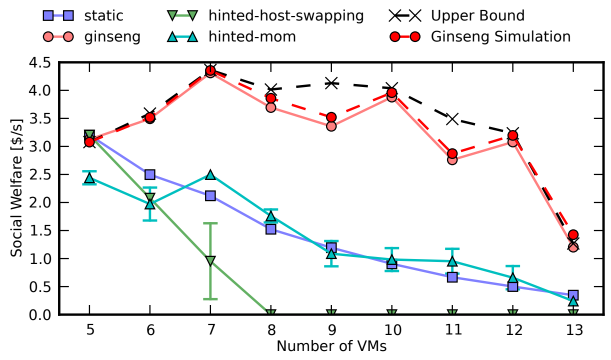
$$f_1 = \begin{cases} 0.1\frac{\$}{Khit} & \mathit{perf}(\mathit{mem}) < 3.4\frac{Khit}{s} \\ 1.8\frac{\$}{Khit} & otherwise. \end{cases} \qquad (2)$$

The piecewise-linear valuation function characterizes service level agreements that distinguish usage levels by unit price. The valuation function for the first guest is shown in Figure 4b.

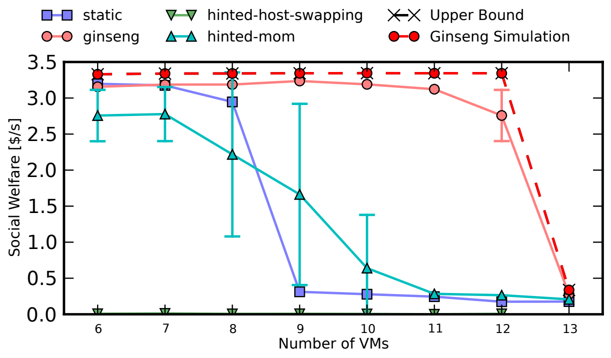


(a) MemoryConsumer (b) Elastic Memcached

Figure 4. Valuation functions for different loads



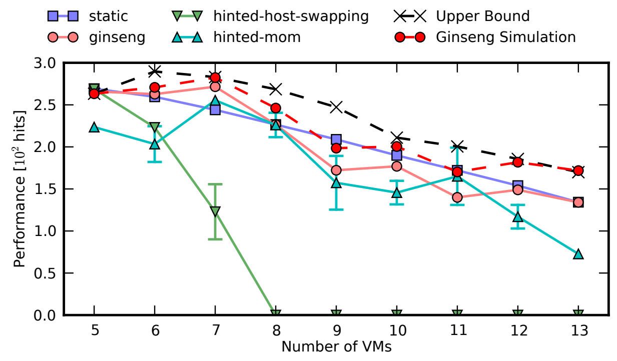
(a) MemoryConsumer, valuation is a square of performance



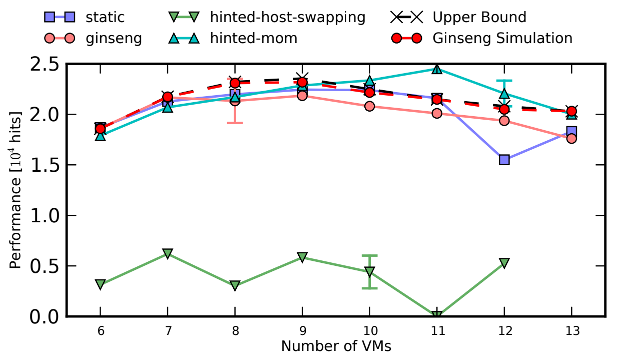
(b) Memcached, first guest valuation is piecewise linear

Figure 5. Social welfare (mean and standard deviation) under different allocation schemes as a function of the number of guests. The dashed lines indicate simulation-based upper bounds on Ginseng's social welfare.

We calculated the social welfare for each experiment using each VM's measured performance and that VM's valuation function. The social welfare of the different experiments is compared in Figure 5. The figures contain two upper bounds for the social welfare, achieved by simulating Ginseng's auction and assuming the guests perform exactly according to their predicted performance (e.g., ignoring cache warmup). The tighter bound results from a simulation of Ginseng itself. The looser bound results from a white-



(a) MemoryConsumer, valuation is a square of performance. Performance is in terms of hits per second.



(b) Memcached, first guest valuation is piecewise linear. Performance is in terms of "get" hits per second.

Figure 6. Performance (mean and standard deviation) under different allocation schemes as a function of the number of guests. The dashed lines indicate the performance according to the simulation from which the upper bounds on Ginseng's social welfare were derived. They are not the upper bound on aggregate performance: memory allocation schemes with lower social welfare may have higher aggregate performance.

box on-line simulation, which results in the theoretically optimal allocations given full offline information. The MOM and host-swapping methods yield negligible social welfare values for these experiments, and are not shown.

As can be seen in Figure 5, Ginseng achieves much better social welfare than any other allocation method for both workloads. It improves social welfare by up to 15.8× for memcached and up to 6.2× for MemoryConsumer, compared both with black-box (static) and white-box approaches (hinted-mom). As the number of guests increases, so does the potential for increased social welfare, because more individual utilities are aggregated to compose the social welfare. However, each guest is allocated a fixed amount of memory (*base*) on startup, reducing our host's free memory, which is available for auction; hence the relative peak in social welfare for 7 guests (MemoryConsumer). In the Memcached experiment the relative peak is flat because the first guest's valuation is much higher than the valuations of the rest of the guests. In both experiment sets, Ginseng achieves 83%–100% of the optimal social welfare. The sharp decline in Ginseng's social welfare for 13 guests comes when Ginseng no longer has enough free memory to answer even the needs of the most valuable guest. This improved social welfare does not come at the cost of overall aggregate performance: it is roughly equivalent to the performance of the better performing competitors, as can be seen in Figure 6.

8.2 Influence of Off-Line Profiling

In our experiments we used performance graphs that were measured in advance in a controlled environment. In real life, such data should be collected on-line, considering both data accumulation and data freshness in view of changing environment conditions. Since the accuracy of the best online methods is bounded by the accuracy of hindsight, we can bound the influence of refraining from on-line evaluation on the performance graphs. In Figure 7 we compare our benchmarks' predicted performance (deduced from measured load and memory quantities using the functions in Figure 2, which were measured without memory overcommitment) with performance values measured during Ginseng experiments for the same loads and memory quantities. The experimental values were collected after the memory usage stabilized. The comparison shows that the profiled data is accurate enough, as can be seen when comparing Ginseng's results in the full experiments to its results with simulated guests in Figure 5.

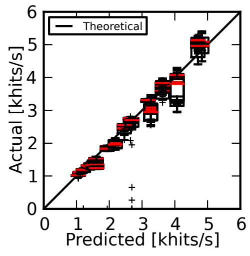
9. Discussion: Host Revenue and Collusion

Ginseng does not maximize host revenue directly. Instead, it assumes that the host charges an admittance fee for the seed virtual machine and maximizes the aggregate client satisfaction (the social welfare). Maximizing social welfare improves host revenues indirectly because better-satisfied guests are willing to pay higher admittance fees. Likewise, improving each cloud host's hardware (memory) utilization should allow the provider to run more guests on each host. To ensure that guests cover power-related operational costs, the host can introduce a dummy bidder that bids with a unit price that equals these costs.

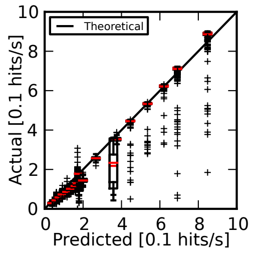
Our guests reach a steady state using indirect interactions that result from their on-line strategy. More sophisticated guests may collude and negotiate to ease their way into a steady state of their choice. While collusion can hasten reaching a steady-state, it may also result in a non-optimal resource allocation. However, collusion which involves bidding with non-truthful unit prices is risky if bidders can join or leave and bids can change. The risk can be increased by randomly limiting the auctionable memory [1].

10. Related Work

Grey-Box and Black-Box Techniques. Magenheimer [25] used the guests' own performance statistics to guide overcommitment. Zhao et al. [40] balanced memory between VMS on the basis of on-the-fly intermittently-built miss-rate curves. Waldspurger [37] randomly sampled pages to esti-



(a) memcached



(b) MemoryConsumer

Figure 7. Comparison of predicted performance values (according to the profile graphs, given load and memory allocation) with measured performance.

mate the quantity of unused guest memory, to guide page reclaim. These methods can be circumvented by a selfish guest, and like white-box methods, ignore the client's valuation of performance. Gong, Gu and Wilkes [11] and Shen et al. [30] used learning algorithms to predict guest resource requirements.

Market Driven Resource Allocation Drexler and Miller suggested auctioning memory chunks to reach a market clearing price [10]. Waldspurger et al. used multiple concurrent sealed-bid, second price auctions to auction processor time slices [35]. Waldspurger [36] allocated resources in proportion to tickets that had to be allocated by a centralized know-all control. In a cloud, no know-all control can allocate tickets to separate economic entities. However, real clouds do not need to allocate tickets—the real money that they use has intrinsic value.

Resource Allocation For Monotonically Rising, Concave Valuations. Maillé and Tuffin [26] extended the PSP [21] to multi-bids, increasing the auction's complexity to instantly reach equilibrium. 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.) In contrast, the MPSP auction leaves the control over the currently desired resource allocation to the guest, who best knows its own current and future needs.

Chase et al. [6] allocated CPU time assuming client valuations of the resource are fully known, concave, and monotonically increasing. Urgaonkar, Shenoy, and Roscoe [33] overbooked bandwidth and CPU given full profiling data.

Auctions With Non-concave Valuations. Bae et al. [4] supported a single bidder with a non-concave valuation function. Dobzinski and Nisan [9] presented truthful polynomial time approximation algorithms for multi-unit auctions with not-necessarily-concave k-minded valuations. They only assumed that the valuations are non-decreasing (because they allow shedding of unneeded goods). Our bidding language of forbidden ranges creates more efficient allocations, because it allows undesired memory to be auctioned to guests who value it more, instead of disposing of it. Ginseng is based on a divisible good auction, and not on *bundles* in a *multi-unit* auction. Hence, its fine-grained allocation accuracy does not increase its algorithmic complexity.

11. Conclusion

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. Ginseng achieves an order of magnitude of improvement in the social welfare function when compared with alternative cloud memory allocation methods.

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, guests that perform the same function for different purposes, such as a test server vs. a production server, can be distinguished by their economic valuation functions.

The MPSP auction is suitable for memory auctioning, but is not limited to this purpose. When used for the allocation of another divisible resource, e.g., bandwidth, whose valuation functions are concave and monotonically rising, it is as efficient as the PSP auction. Hence, Ginseng is not just a memory auctioning platform, but rather the first concrete step towards the Resource-as-a-Service (RaaS) cloud [2]. 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.

12. Acknowledgments

We thank Sharon Kessler, Abel Gordon, Michael Hines, Nadav Amit, Gala Yadgar, Yaron Singer, Hadas Shachnai and Erez Petrank for fruitful discussions. This research was partially supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 605/12) and by the Prof. A. Pazi IAEC-UPBC joint research foundation.

References

[1] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE Conf. on Cloud Computing Technology and Science (CloudCom)*, 2011.

[2] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The resource-as-a-service (raas) cloud. In *USENIX Conf. on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] J. Bae, E. Beigman, R. Berry, M. L. Honig, and R. Vohra. An efficient auction for non concave valuations. In *Int'l Meeting of the Society for Social Choice and Welfare*, 2008.

[5] M. Cary, A. Das, B. Edelman, I. Giotis, K. Heimerl, A. R. Karlin, C. Mathieu, and M. Schwarz. Greedy bidding strategies for keyword auctions. In *ACM Conf. on Electronic Commerce (EC)*, pages 262–271, 2007.

[6] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

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

[8] G. D'Alesandre. Updated app engine pricing faq! Web site, June 2011. `http://tinyurl.com/D-Alesandre`.

[9] S. Dobzinski and N. Nisan. Mechanisms for multi-unit auctions. *J. of Artificial Intelligence Research*, 37:85–98, 2010.

[10] K. E. Drexler and M. S. Miller. Incentive engineering for computational resource management. In *The Ecology of Computation*, pages 231–266. Elsevier Science Publishers, 1988.

[11] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Int'l Conf. on Network and Service Management (CNSM)*, pages 9–16, 2010.

[12] A. Gordon, M. Hines, D. Da Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *Runtime Environments/Systems, Layering, & Virtualized Environments Workshop (ASPLOS RESOLVE)*, 2011.

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

[14] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *Int'l Symposium on Code Generation and Optimization (CGO)*, pages 325–340, 2007.

[15] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IFIP/IEEE Symposium on Integrated Management (IM)*, 2009.

[16] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Int'l Symposium on Memory Management (ISMM)*, 2011.

[17] M. Hines, A. Gordon, M. Silva, D. D. Silva, K. D. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *IEEE Conf. on Cloud Computing Technology and Science (CloudCom)*, 2011.

[18] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2006.

[19] F. Kelly. Charging and rate control for elastic traffic. *European Trans. on Telecommunications*, 8:33–37, 1997.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symposium (OLS)*, pages 225–230, 2007.

[21] A. Lazar and N. Semret. Design and analysis of the progressive second price auction for network bandwidth sharing. *Telecommunication Systems—Special issue on Network Economics*, 20:255–263, 1999.

[22] M. Levy and S. Solomon. New evidence for the power-law distribution of wealth. *Physica A*, 242:90–94, 1997.

[23] A. G. Litke. Memory overcommitment manager. website, 2011. `https://github.com/aglitke/mom`.

[24] B. Lucier, R. Paes Leme, and E. Tardos. On revenue in the generalized second price auction. In *Int'l World Wide Web Conf. (WWW)*, 2012.

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

[26] P. Maillé and B. Tuffin. Multi-bid auctions for bandwidth allocation in communication networks. In *IEEE INFOCOM*, 2004.

[27] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *ACM SIGOPS European Conf. on Computer Systems (EuroSys)*, 2010.

[28] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of amazon EC2. In *USENIX Conf. on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[29] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *ACM SIGOPS European Conf. on Computer Systems (EuroSys)*, pages 337–350, 2013.

[30] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[31] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Int'l Symposium on Memory Management (ISMM)*, 2004.

[32] W. Souma. Universal structure of the personal income distribution. *Fractals*, 9(04):463–470, 2001.

[33] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol.*, 9(1), 2009.

[34] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *J. of Finance*, 16(1), 1961.

[35] C. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: a distributed computational economy. *IEEE Trans. on Software Engineering*, 18(2):103–117, 1992.

[36] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, 1995.

[37] C. A. Waldspurger. Memory resource management in Vmware ESX server. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2002.

[38] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 103–116, 2006.

[39] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 21–30, 2009.

[40] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *USENIX Annual Technical Conf. (ATC)*, 2011.

[41] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2004.