Ginseng: Market-Driven Memory Allocation

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

[1]Technion – Israel Institute of Technology   [2]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 $6.2\times$–$15.8\times$ improvement, which is 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

General Terms   Virtualization, Economics, Performance

Keywords   Virtual Machines, Operating Systems, KVM, Memory Overcommit, Oversubscription

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 use in cloud computing platforms today [19, 25, 27, 36, 41, 55]. 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." [12]. 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 [8, 34, 55]. 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 [3, 42]. They would like to pay only for the physical memory they need, when they need it [1, 5, 18]. 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-overcommitted 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 both clients, whose satisfaction is improved, and providers, whose hardware utilization is improved.

Previous physical memory allocation schemes assumed that guest virtual machines are run by fully cooperative clients, who let the host know precisely what each guest is doing, how much benefit additional memory would bring to it, and the importance of the workload to the client [19, 25, 27, 41]. Gingko, for example, assumed that guests are cooperative and that the host knows the benefit additional memory would bring each guest [19, 27]. Heo et al. [25] solved a control problem over performance level objectives, by measuring guest performance inside the guests, again assuming guest cooperation. Nathuji et al. [41] sold performance instead of resources, again by measuring guest performance inside the guests. All 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 [3] 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 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.

The host can allocate memory on the basis of passive black-box or grey-box measurements [29, 34, 36, 55, 60, 61]. For example, Jones, Arpaci-Dusseau, and Arpaci-Dusseau [29] monitored I/O and inferred major page faults, and Zhao and Wang [61] monitored use of physical pages to balance the guests' need for physical memory. 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, 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?

We make the following three contributions. **Our first contribution** 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). 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.

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 im-

plementation of a single-resource Resource-as-a-Service (RaaS) cloud [3]. Ginseng is the first cloud platform to optimize overall client satisfaction for black box guests. 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. Many applications in use today are static-memory applications, but elastic-memory applications are starting to become more common. It is easy to convert Java- or database-dependent static-memory applications to being dynamic using environments such as Salomie et al.'s database [44] or a Java runtime with balloons [19, 27, 44] or CRAMM [58]. Other examples of dynamic-memory applications include dynamic heap adjustment for garbage-collected environments [21, 26, 59], applications making use of the Linux mempressure control group [40, 63] and CloudSigma's Burst Pricing [1]. **Our third and final contribution** is two elastic-memory benchmark applications: an **elastic-memory version of Memcached**, a widely-used key-value storage cloud application, and **MemoryConsumer**, an elastic memory benchmark we developed.

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 [31] with Litke's Memory Overcommit Manager MOM [34]. It controls the exact amount of physical memory allocated to each guest via the libvirt abstraction using balloon drivers [8, 34, 55].

The *balloon driver*, first presented by Waldspurger [55], 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. Ginseng does not specifically depend on any balloon implementation; it only requires that the host supports some underlying mechanism for balancing physical memory between guests.

Ginseng's system architecture is depicted in Figure 1. Ginseng has a host component and a guest component. The
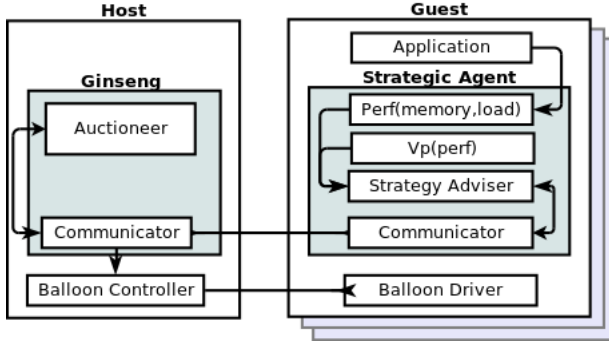
Figure 1: Ginseng system architecture

host component includes the *Auctioneer*, which runs the MPSP auction that is presented in **Section 3** and described in detail in **Section 5**. The auctioneer and the guests asynchronously communicate information for the purpose of the auction using their respective *Communicators*, according to the protocol specified in **Section 4**. The host's communicator is also responsible for instructing the Balloon Controller how to allocate memory between different guests. The Balloon controller inflates and deflates the balloon drivers inside the guests. In the guest, the *Strategy Adviser* is the brains of the guest's economic learning agent, which acts on behalf of the client. The client is free to choose its logic, provided it speaks the MPSP protocol. Our implementation of an adviser is described in **Section 6**.

3. Memory Auctions

Ginseng allocates memory efficiently because its guests bid for the memory they want in a specially-designed auction. We begin by describing how Ginseng auctions memory.

In Ginseng, each guest has a different, changing, private (secret) *valuation* for memory. Simply put, 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 [11, 20, 53] 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, 35].

Various auction mechanisms, some of which resemble the VCG family, have been proposed for *divisible* resources, in particular for *bandwidth sharing* [30, 32, 37]. For prac-

tical 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 [32]. 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. The memory valuation function, which describes how much the guest is willing to pay for different memory quantities, is a function of the load the guest is under, the performance gain or loss it expects from less or more memory given that load, and the value of (less or more) performance to the guest. Formally, $V(mem, load) = V_p(perf(mem, load))$, where $V_p(perf)$ refers to the valuation of performance as described below, and $perf(mem, load)$ 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. Performance can be mapped offline as function of memory and load, and done by Hines et al. [27] and Gordon et al. [19]. As we demonstrate in Section 8.2, the performance predictions made in this paper according to the offline measurements were accurate enough. However, real-world performance may depend on many variable conditions such as program phase and bottlenecks in resources other than memory. To this end, performance can be measured online as several works demonstrate [60–62]. 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 in which it operates.

$V_p(perf)$, the guest's owner's (i.e., the client's) valuation of performance function, describes the value 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. It is computed mostly on the basis of economic considerations and business logic.

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

and under the current conditions (e.g., load). The guests do not need to measure the *perf*(·) function for the full range of possible memory values. It is enough that these guests measure it only for the working points that they would like to consider bidding for.
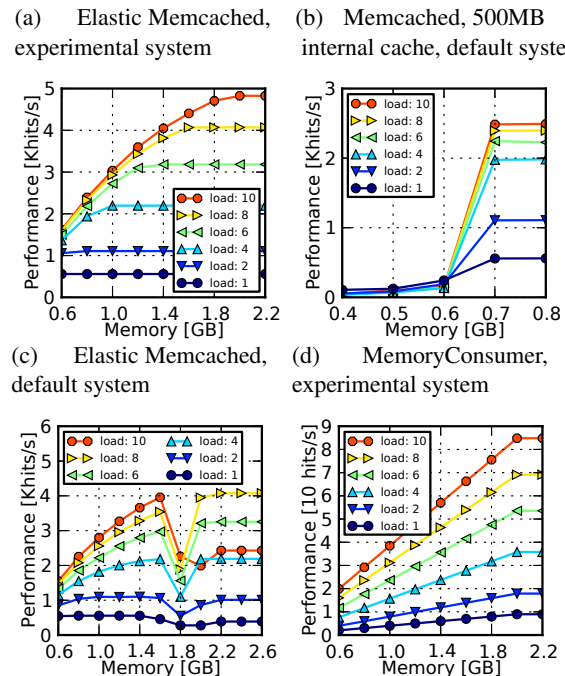
If either of these functions is non-concave or not monotonically rising, the composed function may be non-concave or not monotonically rising as well. The PSP auction optimally allocates a divisible resource if and only if all the valuation functions are monotonically rising and concave. Other bandwidth auctions also rely on the monotonically rising concave property of the valuation functions.

Guest performance *perf*(*mem*, *load*), however, 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 2). 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 [49]. An application that knows how much memory it can use and can change its heap size on-the-fly will have a more concave graph. We modified *memcached* so that it can be instructed to change its heap size on-the-fly by releasing and allocating some of the memory used for its internal cache. Our *elastic memcached* has a concave, monotonically rising performance graph in the same experimental environment (Figure 2). However, in a default system configuration, its performance graph is neither concave nor monotonically rising (Figure 2), due to a network bottleneck that was prevented in the experimental environment. But a real production system guest cannot fine-tune its setup parameters and re-design its software on-the-fly; it has to make the best of the performance graphs it measures for the current setup, which might look like Figure 2. To do that, we designed Ginseng to allow valuation-of-memory graphs *V*(*mem*, *load*) such as Figure 2 that are neither concave nor monotonically rising.

Auction protocols that assume monotonically rising concave valuation functions either interpret a bid of unit price and quantity (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

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.

(a) Elastic Memcached, experimental system

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

(c) Elastic Memcached, default system

(d) MemoryConsumer, experimental system

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 until the next auction round, when 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 not to make more profit off the extra memory rental, and use this mechanism solely to optimize social welfare, thereby attracting more guests. The price of base memory is not affected by the prices paid for extra memory.

Ultra high-end guests 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 budget-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 their 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 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, sorted in ascending order. We denote the desired ranges by $[r_j, q_j]$ for $j = 1 \ldots m_i$. 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 Figure 2. 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 Figure 2.

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 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 6.3.

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 [63]. 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 the sake of 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 divis-

ible good allocation by sorting the guests lexically by three qualities: first by their bid unit-price, second (to break ties) by their current holdings [61] and last by a random shuffle. 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 all of g's requested quantity. 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 allocation. In the first one, guest g gets a memory quantity that is large enough to cover all of R. In the second one, g gets a memory quantity that is small enough such that it does not get any memory within 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 [32]. 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:

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

According to this payment rule, when guest i is charged $p'_i q'_i$, it actually pays for the damage that its bid made to the benefit of the rest of the 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.

5.4 Single Round Example

Consider a system with 6 GB of physical memory and two guests. The first guest bids a unit price of 2 for between 3GBs and 4GBs of memory ($p = 2, r_1 = 3, q_1 = 4$) and the second guest bids a unit price of 1 for between 3GBs and 5GBs of memory ($p = 1, r_1 = 3, q_1 = 5$). In the first stage, we sort the guests by price, and try to allocate 4GB to guest 1 and 2GB to guest 2. This is an invalid allocation, because the second guest gets a quantity that falls in its forbidden range (anything less than 3GBs). We therefore examine two constrained systems instead. (1) The second guest gets no more than the minimum of the forbidden range, which is 0. In this case, the overall allocation is 4GB to the first guest, with a social welfare of 8. (2) The second guest is allocated at least as much as the maximum of its forbidden range, i.e., at least 3GBs. Then the rest of the free memory is allocated by the order of prices, so the first guest gets the other remaining 3GBs. The social welfare in this case is 9, and this is the chosen allocation.

According to the payment rule given in Equation 1, the guests pay $p'_1 = \frac{1}{3} \left(1 \left[5 - 3 \right] \right) = \frac{2}{3}$ and $p'_2 = \frac{1}{3} \left(2 \left[4 - 3 \right] \right) = \frac{2}{3}$, because in each other's absence they each would have gotten all of the memory they wanted.

6. Guest Strategy

So far, we discussed 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 for it? In an exact VCG auction, the guests would be expected to inform the host about their valuation for different memory quantities. However, the reduced MPSP bidding language lightens the computational burden off the host and leaves the choice of memory quantity with the guest. Multi-bid auctions are further discussed in Section 10.

In this section we present an example 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 [9].

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, with the help of on-line algorithms (Section 6.3).
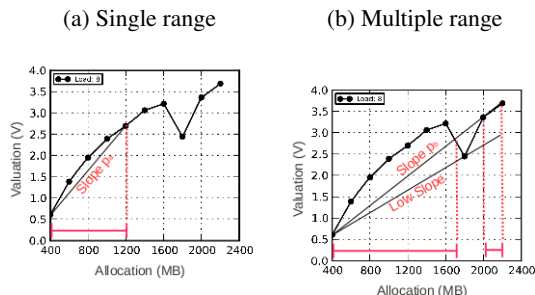
6.1 Choosing a Bid Price

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 ($p(q)$). 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 $p(q) = \frac{V(base+q)-V(base)}{q}$ is the best strategy, no matter what the other guests do. By bidding lower than $p(q)$, 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 $p(q)$ is still the best strategy for q regardless of other guests' bids: $p(q)$ is the guest's minimal valuation for the range because the unit valuation drops with the quantity. See, for example, Figure 3, 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 that of 1200MB.

In the rest of the cases, the valuation function is non-concave or not even monotonically rising. In such functions, the mean unit-valuation $p(q)$ may rise locally with quantity: For example, in Figure 3, $p(2200MB)$ is higher $p(1800MB)$. This means that simply bidding for 2200MB with a unit-price of $p(2200MB)$ may result in getting a memory quantity for which the guest is not willing to pay as 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 $p(2200MB)$.

The forbidden ranges mechanism allows the guest to bid $p(q)$ without a risk of operating at a loss. However, the guest may have something to gain by bidding with a unit-price that is less than $p(q)$. 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 enables the host to give it a partial allocation in more cases, when the alternative might be not getting any memory at all. In Figure 3, the lowest bid-price worth exploring is labeled as "Low Slope": it eliminates any need of forbidden ranges.

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

(a) Single range   (b) Multiple range

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 it makes. The 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 $p(q_m)$. In our experiments, a steady state is typically reached after 3 auction rounds.

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 assesses its utility from the next auction by estimating the quantity of memory it will get q_{est}, which is estimated for simplicity as q_m if $p > p_{min}$, and as 0 otherwise. p_{min} is the lowest price the guest can offer and still have a chance of getting any memory at all. The guest's estimation of p_{min} is discussed in Section 6.3.

The utility estimation also requires an estimation of the unit price to be paid for the allotted memory amount, p_{est}. The guest's utility is its valuation of the memory it gets minus the charge. The guest estimates its utility from bidding (p, q_m) by dividing it to two components: (1) its estimate of the valuation improvement from winning the memory it expects to win and (2) its estimate for the charge. For concave valuation functions $V(\cdot)$, the estimated utility is maximized when $p(q_m) = p_{min}$. In such cases, the guest needs only estimate and predict p_{min} to bid optimally. For other (non-concave) functions, to find the memory quantity that maximizes the estimated utility the guest must evaluate the estimated charge. To this end it assesses the estimated memory quantity it will get q_{est} as described above and the estimated unit price it will pay according to Section 6.3. If several values of q_m maximize the estimated utility, the

guest prefers to bid with higher p values, which improve its chances of winning the auction.

6.3 Predicting Guest Utility

In this subsection we describe the learning algorithms used by the guest to predict its utility. The guest evaluates p_{min} for the current round on the basis of ten recent borderline bids that are announced by the host. The price to be paid, p', depends mainly on losing bids. To predict p', the guest maintains a historical table of (p', q') pairs, and uses it as a basic estimate for p_{est}. The p_{est} estimate is further bounded from above by the highest losing bid price in the last auction round.

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 that are 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)* [34] 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 it can be easily circumvented by a malicious guest.

Workloads. To experiment with overcommitment trade-offs, 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 when the balloon is inflated, the system would not run out of memory.

Elastic memcached is a version of memcached that can change its heap size on-the-fly to respond to guest memory changes. When ordered, memcached can free some internal-cache slabs (the less-needed ones, according to its internal statistics), 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 bottlenecks in resources other than memory. As with memcached, we chose a range of 1950 cells, so that performance graphs would saturate at 2GB.

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. [27]. We gradually increased and decreased the physical memory in small steps, waiting in each step for the performance to stabilize. For memcached we waited and measured the performance for 200 seconds, and for MemoryConsumer for 60 seconds. The $perf(mem, load)$ graphs can be seen in Figure 2 for the elastic Memcached and Figure 2 for MemoryConsumer.

Load. We defined "load" for memcached and MemoryConsumer as the number of concurrent requests being made. We used *coordinated dynamic loads*, where 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 hits per second of different experiments will be comparable. The load values and their exchange timing were chosen to increase the diversity among the guests, as expected in a real system, where guests' loads change independently of other guests. Guests with different loads also have different memory valuation functions, and are thus more diverse, as in a real system.

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 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`. To reduce measurement noise, we disabled EIST, NUMA, and C-STATE in the BIOS and Kernel Samepage Merging

---

[1] Elastic-memcached is available from ANON. INFO..

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	$base$	2

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

(KSM) [4] 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 $1 \ldots N$. When the host also drove the load for memcached, memslap processes were randomly (uniformly) pinned to threads $(N + 1) \ldots 15$.

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.

Reducing Indirect Overcommitment. Bare metal operating systems keep some memory free, in case of sudden memory pressure. In a virtualized system, the hypervisor can indirectly overcommit this memory by giving it to other operating systems while it is not in use; the hypervisor 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 the tunable knob 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 settings (vm.overcommit_memory=0) that are intended for a bare-metal OS. In a VM with the default settings, an application which needs 600MB would have required on our system 300MB more, which it could not make use of. These 300MB would only be available for use by other virtual machines.

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 the performance resulting from 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 guest major faults were insignificant (indicating guest thrashing hardly happened), and to verify that the performance measurement method was getting enough time 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-warming was not affected.
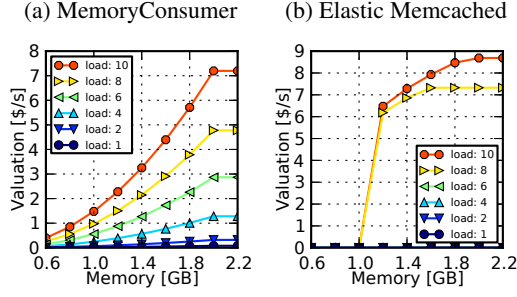
8. Performance Evaluation

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 begin by evaluating 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 ex-

Figure 4: Valuation functions for different loads

(a) MemoryConsumer   (b) Elastic Memcached

periments. 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 $V_p(perf)$ that make the resulting composed valuation function $V(mem)$ non-concave.

In the first experiment set (MemoryConsumer), each guest i's valuation function is defined as $V_i(mem) = f_i \cdot (perf(mem))^2$, where the f_i values were drawn from the *Pareto distribution*, a widely used model for income and asset distributions [50]. We used a Pareto index of 1.1, which is reasonable for income distributions [51], and a lower bound of $10^{-4} \frac{\$}{Khit}$.
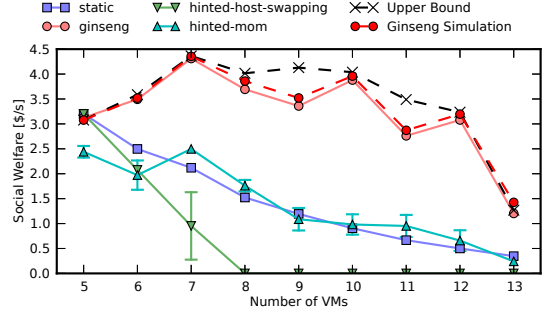
The "square of performance" valuation function is characteristic of on-line games and social networks, where the memory requirements are proportional to the number of users, and the income is proportional to user interactions, which are proportional to the square of the number of users. The composed valuation function is illustrated in Figure 4.

In the second experiment set (elastic memcached), each guest i's valuation function is defined as $V(mem) = f_i \cdot perf(mem)$, where the f_i values were distributed according to a *Pareto distribution* with a Pareto index of 1.36 (according to Levy and Solomon's wealth distribution [33]), bounded in the range $[10^{-4}, 100] \frac{\$}{Khit}$. The bounding represents the fact that on-line trading does not span the whole range of human transactions: some are too cheap or too expensive to be made on-line. The highest coefficient was set as:
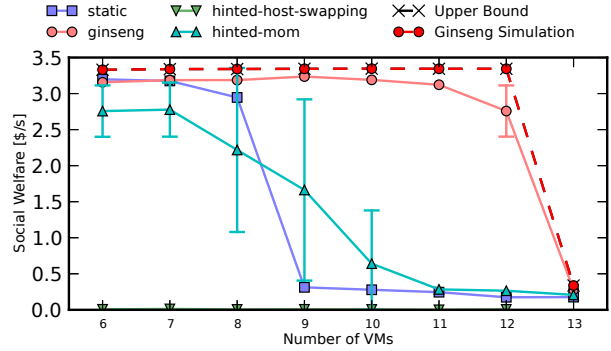
$$f_1 = \begin{cases} 0.1 \frac{\$}{Khit} & perf(mem) < 3.4 \frac{Khit}{s} \\ 1.8 \frac{\$}{Khit} & otherwise. \end{cases} \quad (2)$$

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

We calculated the social welfare for each experiment using each VM's measured performance 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

(a) MemoryConsumer, valuation is a square of performance
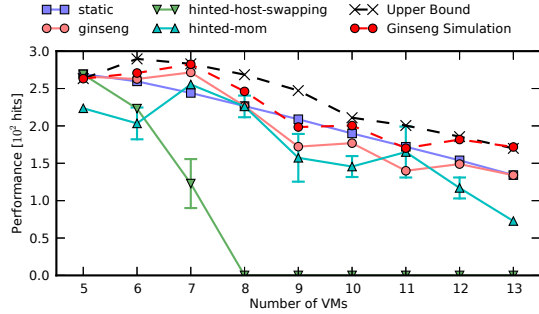

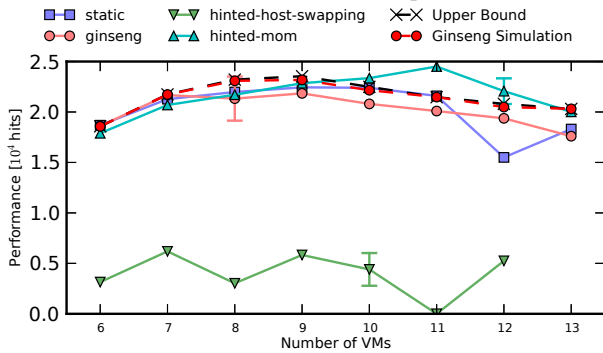
(b) Memcached, first guest valuation is piecewise linear

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.

their predicted performance (neglecting cache warmup, for example). The tighter bound results from a simulation of Ginseng itself. The looser bound results from a white-box on-line simulation, that 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 presented.

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 $15.8\times$ for memcached and up to $6.2\times$ for MemoryConsumer, compared with both black-box approaches (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 rest. 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

(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.

enough free memory to answer even the needs of the most valuable guest.

As we saw above, Ginseng provides much better social welfare than alternative memory allocation method. But does it do so at the cost of reducing overall aggregate performance? As can be seen in Figure 6, Ginseng provides aggregate performance that is roughly equivalent to the performance of the better methods, namely hinted-MOM and static division, while providing an order of magnitude better social welfare.

8.2 Impact of Off-Line Profiling

In our experiments we used performance graphs that were measured in advance in a controlled environment. In real life, artificial intelligence methods should be used to collect such data on-the-fly, considering both data accumulation and data freshness in view of changing environment conditions. Since the accuracy of the best on-the-fly methods is bounded by the accuracy of hindsight, we can bound the impact of refraining from on-the-fly evaluation on the performance graphs. In Figure 7 we compare our benchmarks' predicted performance (according to measured load

and memory quantities, and using Figure 2) with performance values measured during Ginseng experiments for the same loads and memory quantities. The experimental values were collected after the memory usage stabilized (more than T_{memory} after a memory change). 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 attempt to 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.

The guests we implemented reach a steady state using indirect negotiations that result from their on-line strategy (in Section 6.3). More sophisticated guests may directly collude and negotiate to ease their way into a steady state of their choice [6]. They can complete the deal among themselves by subletting memory to each other or by making side-payments. Such guests might bid differently than their true valuation of the memory, or coordinate the memory quantities they bid for such that they only bid together for as much as the host can offer, thus reducing and even eliminating all charges.

The bright side of collusion is that it shortens the time it takes the system to converge to a steady state. In particular, when the colluding guests only change their desired quantities and keep bidding a true unit-price, the allocation still optimizes the social welfare.

If the host cares about its revenue from the extra memory rental, e.g., due to power-related operational costs, it can set a minimum (reserve) price for the extra memory. This can easily be implemented by adding a dummy bidder that bids on behalf of the host for all the memory with a unit-price that equals the operational costs. The dummy bid will prevent the host from renting the extra memory at a loss, and will also limit the gain that guests can achieve at the expense of other guests.

The dark side of collusion is that if guests do not bid with their true valuations, the allocation will not necessarily be optimal. However, bidding with a unit-price which is not the true valuation carries the risks of losing the auction or working at a loss, as described in Section 6.1. Colluding to bid unit prices which are not true valuations is beneficial when the other guest's bids and the auctionable memory are known in advance. In a cloud platform guests may join the auction between rounds, or the valuation of existing guests might change, and a new bid might be made below a ficti-
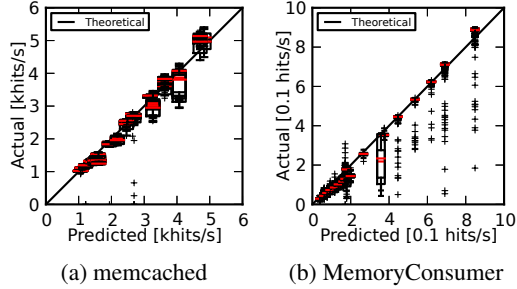
Figure 7: Comparison of predicted performance values (according to the profile graphs, given load and memory allocation) with measured performance.

tious high bid, forcing the colluding guest to pay more than the memory is worth to it. The host can further increase the risk involved in such bids by introducing uncertainty to the supply, for example, by randomly limiting the auctionable memory [2]. This will usually not greatly hinder the operation of non-colluding guests, but it will make the planning of collusion much harder.

10.   Related Work

White-Box Memory Overcommitment. Heo et al. [25] balanced memory allocations according to desired performance levels. Under memory pressure they divided the memory according to a fair share policy. Nathuji, Kansal and Ghaffarkhah [41]'s guests specified several performance and payment levels and the host chose which level to fulfill. This approach guarantees the host demand for any excess production power it has. Our approach is guest oriented, leaving the designation of the current required resource amount in the hands of the guest. In Ginkgo, Hines et al. [27] and Gordon et al. [19] used optimization with constraint satisfaction to optimize a general social welfare function of the guests' performance. These works assume guest cooperation, while we analyze the guest as a non-cooperative, selfish agent. Our work is the first work on memory allocation which assumes non-cooperative guests.

   Grey-Box and Black-Box Techniques. Magenheimer [36] used the guests' own performance statistics to guide overcommitment. Jones, Arpaci-Dusseau, and Arpaci-Dusseau [29] inferred information about the unified buffer cache and virtual memory by monitoring I/O and inferring major page faults. Zhao and Wang [61] monitored use of physical pages, and Zhao et al. [60] balanced memory between VMS on the basis of on-the-fly intermittently-built miss-rate curves. Waldspurger [55] randomly sampled pages to estimate 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. Gupta et al. [22] did not require any guest cooperation for their content based page sharing. Wood et al. [57] allocated guests to physical hosts according to their memory

contents. Gong, Gu and Wilkes [18] and Shen et al. [48] used learning algorithms to predict guest resource requirements.

   Sekar and Maniatis [47] argued that all resource use must be accurately attributed to the guests who use it so that it can be billed. In contrast, Ginseng lays the burden of metering on the client.

   Guest Hint Techniques. Schwidefsky et al. [46] used guest hints to improve host swapping. Miłoś et al. [39] incentivized guests to supply sharing hints by counting a shared page as a fraction of a non-shared page. Like Ginseng, their method can be applied to non-cooperative guests.

   Resource Allocation with *Funny Money* *Funny money* was used in shared systems to control resource allocation. However, when using funny money, the problem is attaching real value to it. For example, Waldspurger [54] used a proportionally fair allocation using tickets, which stood for shares. Tickets had to be allocated by a centralized know-all control. In a cloud there is no know-all control that can allocate tickets to separate economic entities. However, such a control is not needed in a cloud which already charges clients real money, which has intrinsic value.

   General Resource Allocation For Monotonically Rising, Concave Valuations. Kelly [30] 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 influence on the price they pay). Popa et al. [43] traded off proportional fairness with starvation prevention. Johari and Tsitsiklis [28] computed the price of anarchy of Kelly's auction, and Sanghavi and Hajek [45] improved the auction in this respect.

   Maillé and Tuffin [37] 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. 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.) 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. Maillé and Tuffin also showed that the PSP's social welfare converges to theirs [38].

Chase et al. [10] 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 [16].

Urgaonkar, Shenoy, and Roscoe [52] overbooked bandwidth and CPU cycles given full profiling information but did not address memory.

Unlike bandwidth and CPU auctions, our memory auction is oriented toward minimizing transfer of ownership. Unlike divisible good auctions, it supports non-concave valuation functions.

Ghodsi et al. [17], Dolev et al. [14] and Gutman and Nisan [23] considered allocating multiple resources to strategic guests whose private information is the relative resource quantities they require. In contrast, Ginseng compares valuations of different strategic clients.

Drexler and Miller suggested auctioning memory chunks to reach a market clearing price [15]. Waldspurger et al. used multiple concurrent sealed-bid, second price auctions to auction processor time slices [56].

Auctions With Non-concave Valuations. Bae et al. [7] supported a single bidder with a non-concave valuation function. Dobzinski and Nisan [13] 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. Our bidding language of forbidden ranges creates more efficient allocations than free disposal, because it allows undesired memory to be auctioned to guests who value it more, instead of disposing of it. Had Ginseng been implemented on the basis of *bundles* in a *multi-unit* auction, the memory would have been divided to units. The clients would have bid for bundles of such units. The host would have had to trade off the accuracy of the final allocation with the complexity of the auction by controlling the bundle size. As the number of units grew, the final allocation would be more accurate, but the auction's complexity would grow. In contrast, the MPSP auction is of a continuous resource, and thus its fine-grained allocation accuracy does not increase its algorithmic complexity.

11. 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. Ginseng achieves an order of magnitude of improvement in the social welfare 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, 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 [3]. 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.

References

[1] Cloudsigma price schedules: Burst pricing. `http://www.cloudsigma.com/en/pricing/price-schedules`.

[2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 spot instance pricing. In *IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.

[3] 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.

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

[5] 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.

[6] Lawrence M. Ausubel and Paul Milgrom. *Combinatorial auctions*, chapter 1. The lovely but lonely Vickrey auction, pages 17–40. MIT Press, 2006.

[7] 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.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[9] Matthew Cary, Aparna Das, Ben Edelman, Ioannis Giotis, Kurtis Heimerl, Anna R. Karlin, Claire Mathieu, and Michael Schwarz. Greedy bidding strategies for keyword auctions. In *ACM conference on Electronic Commerce (EC)*, pages 262–271. ACM, 2007.

[10] 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.

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

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

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

[14] 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.

[15] K. Eric Drexler and Mark S. Miller. Incentive engineering for computational resource management. In B.A. Huberman, editor, *The ecology of computation*, pages 231–266. Elsevier Science Publishers, North-Holland, Amsterdam, 1988.

[16] 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.

[17] 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.

[18] 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.

[19] 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 *Runtime Environments/Systems, Layering, and Virtualized Environments (ASPLOS RESoLVE) workshop*, 2011.

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

[21] Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski. Isla vista heap sizing: Using feedback to avoid paging. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO*, pages 325–340, 2007.

[22] 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.

[23] 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.

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

[25] Jin Heo, Xiaoyun Zhu, Pradeep Padala, and Zhikui Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IFIP/IEEE Symposium on Integrated Management (IM)*, 2009.

[26] Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan E. Bard. Waste not, want not: resource-based garbage collection in a shared environment. In *Proceedings of the international symposium on Memory management (ISMM)*, 2011.

[27] 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.

[28] Ramesh Johari and John N. Tsitsiklis. Efficiency loss in a network resource allocation game. *Mathematics of Operations Research*, 29(3):407–435, 2004.

[29] 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.

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

[31] 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.

[32] 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.

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

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

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

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

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

[38] 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.

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

[40] Kim Minchan. [PATCH v2] memcg: Add memory.pressure_level events. http://tinyurl.com/KimMinchan, February 2013.

[41] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2010.

[42] 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.

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

[44] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 337–350. ACM, 2013.

[45] Sujay Sanghavi and Bruce Hajek. Optimal allocation of a divisible good to strategic buyers. In *IEEE Conference on Decision and Control (CDC)*, 2004.

[46] 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.

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

[48] 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.

[49] Sunil Soman, Chandra Krintz, and David F. Bacon. Dynamic selection of application-specific garbage collectors. In *4th International Symposium on Memory Management (ISMM)*, 2004.

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

[51] Wataru Souma. Physics of personal income. `http://arxiv.org/pdf/cond-mat/0202388`, 2002.

[52] 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.

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

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

[55] 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.

[56] Carl.A. Waldspurger, Tad. Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: a distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, 1992.

[57] 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.

[58] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Cramm: virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, 2006.

[59] Chengliang Zhang, Kirk Kelsey, Xipeng Shen, Chen Ding, Matthew Hertz, and Mitsunori Ogihara. Program-level adaptive memory management. In *Proceedings of the 5th international symposium on Memory management (ISMM)*, 2006.

[60] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. Low cost working set size tracking. In *USENIX Annual Technical Conference (ATC)*, 2011.

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

[62] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.

[63] Bartlomiej Zolnierkiewicz. The mempressure control group. `http://lwn.net/Articles/531077/`, January 2013.