Ginseng : Market-Driven LLC Allocation

Liran Funaro Orna Agmon Ben-Yehuda Assaf Schuster
Technion—Israel Institute of Technology
{fonaro,ladypine, assaf}@cs.technion.ac.il

Abstract

Cloud providers must dynamically allocate their physical resources to the right client to maximize the benefit that they can get out of given hardware. Cache Allocation Technology (CAT) makes it possible for the provider to allocate last level cache to virtual machines to prevent cache pollution. The provider can also allocate the cache to optimize client benefit. But how should it optimize client benefit, when it does not even know what the client plans to do?

We present an auction-based mechanism that dynamically allocates cache while optimizing client benefit and improving hardware utilization. We evaluate our mechanism on benchmarks from the Phoronix Test Suite. Experimental results show that *Ginseng* for cache allocation improved clients' aggregated benefit by up to $\times 42.8$ compared with state-of-the-art static and dynamic algorithms.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources wrapped as an infrastructure, i.e., a guest virtual machine (VM), to their clients. To compete in the tough market of cloud computing, providers must improve their clients' quality-of-service (QoS) while maintaining competitive pricing and reducing per-client management cost. Thus, better hardware utilization is necessary. New Intel technology that supports last level cache (LLC) allocation allows better cache utilization via cache partitioning.

Providers can utilize this new technology to guarantee clients' performance requirements by preventing applications from polluting each other's cache [27], as Intel intended [20]. Moreover, they can accommodate more clients' performance requirements by granting more LLC to those who benefit from it and preventing access to those who do not. This increases the provider's ability to consolidate the physical host.

Without any client performance information, the provider can only optimize guest performance according to host-known metrics, such as instructions-per-second (IPC), LLC hit-rate, LLC reads-count, and so forth [15, 36, 57]. The host does not know what the client's real benefit from more cache is, nor can it compare benefits that different clients draw from cache. For example, higher IPC does not necessarily indicate better performance, as the guest VM might just be polling on a spin-lock more quickly.

Moreover, a client may be willing to settle for poorer performance in exchange for a lower payment. This might be the case, for example, when the guest VM of a performance-demanding client is running maintenance work in between important workloads every few seconds. Nevertheless, lacking the guests' current workload information, the host will try to improve its performance despite the lack of benefit to the client. This, in turn, may hinder the performance of other guests. It is therefore in the interest of both provider and client that clients pay only for the fine grained LLC they need, when they need it [1, 4, 11, 43]. Client satisfaction will thus be improved, as clients can pay less for the same performance but only when it is really needed, while providers will be able to improve hardware utilization.

However, real-world public cloud clients are *selfish*, *rational* economic entities. They will not let the provider know precisely how much benefit each quantity of cache ways would bring to it. They are *black-boxes*, and as such, unlike *white-box* clients [11, 18, 19, 41], they will not share their *true* private information with their provider unless it is in their own best interest to do so. For example, if the host allocates cache ways to guests who will derive the greatest benefit from it, each guest will claim that it has the most to gain from additional cache ways. Likewise, if the host allocates cache ways to guests who perform poorly, each guest will claim poor performance.

Even passive *black-box* measurements taken by the provider can be manipulated [15, 36, 57]. For example, a guest can fake cache misses by adding random instructions in the code that access random—non-cached—memory addresses. Such instructions will not delay the out-of-order-execution (OOOE) CPU as they are independent of the other instructions, and they will induce many cache-misses.

In this paper we address the problem of how cloud providers should allocate cache among selfish black-box clients in light of the new cache allocation technology.

Our contribution towards a solution to this problem is *Ginseng* [2] for *cache allocation*, a market-driven auction system that maximizes the aggregated benefit of the guests in terms of the economic value they attribute to the desired allocation, using game-theoretic principles. This approach encourages even a *selfish* guest to bid for cache according to its *true* benefit. *Ginseng* was first introduced for memory allocation [2], and a similar, auction-based approach was used before for bandwidth allocation [33].

We evaluate *Ginseng* on benchmarks from the Phoronix Test Suite [32], which we classify according to their benefit from the cache. We show that *Ginseng* improves the aggregated economic benefit of guests from cache by up to $\times 42.8$ compared to the prevalent method used by today's cloud providers.

Our second contribution is an evaluation of the attributes which differentiate dynamic cache allocation from other resources: **(1)** As opposed to memory, cache does not have to be exclusively allocated and can be shared effectively. However, mutual trust is required to allow benefit for the sharing participants. **(2)** Unlike bandwidth—but much like memory—cache has to be warmed up before the guest can benefit from it. However—unlike memory pages—cache must be allocated consecutively, which induces more cache way transfers when the allocation is changed. Furthermore, Intel's allocation mechanism might fail to enforce frequent transfers of cache ownership. Thus, dynamic cache allocation might incur performance overhead. We address the question of when it is beneficial to share cache and when exclusive allocation is preferable, and we measure and analyze the overhead of frequently changing the allocation.

2 System Architecture

Ginseng is a market-driven cloud mechanism that allocates resources to guest virtual-machines by means of an auction. It is implemented for cloud hosts running the KVM hypervisor [30] but can work seamlessly on any other hypervisor. *Ginseng* for *cache allocation* controls the cache ways allocated to each guest using the *cache-driver* described in §3.

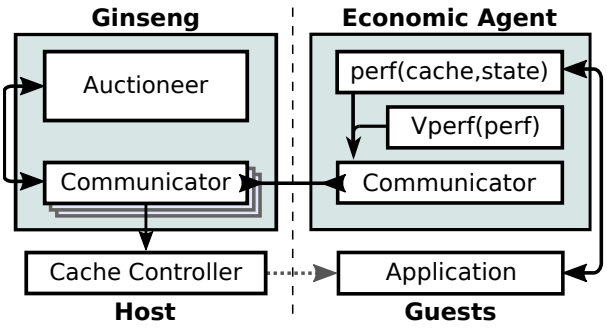


Figure 1: Ginseng system architecture

Ginseng has a host component and a guest component, as depicted in Figure 1. The host's Auctioneer uses the Vickrey-Clarke-Groves (VCG) auction [7, 13, 54], to be described in §4. The host's communicators communicate with the guests using the auction protocol, detailed in §5. It also instructs the cache controller how to allocate cache ways among the guests. The guest's economic agent bids on behalf of the client by stating a valuation for each number of cache ways. The guest component we implemented bids with a true valuation, as that is the best strategy for the guest [7, 13, 54], but *Ginseng* does not enforce any restrictions on the implementation and strategy of the guest's economic agent.

3 Cache Architecture

Intel's cache, and LLC in particular, stores data in granularity of cache lines that typically vary from 64 to 256 bytes, depending on the machine. The cache is organized in ways, each of which is a hash table, where the key is a hash value of the line's memory address and the value is the content of the cache line. Way locations that are designated to be filled by lines with the same keys are called a set.

When reading from a memory address or writing to it, the CPU first computes the address's set by using the hash function. Then the line is stored in this set on one of the cache ways. If the entire set is full, the least recently used (LRU) line in the set will be evicted and replaced.

When an application uses the cache exclusively, it will evict its own least recently used data. However, when several applications use the same cache, one might evict the other's cache lines and influence its performance. To prevent this, Intel's new cache allocation technology (CAT) allows cache partitioning. The API defines the notion of classes-of-service (COS), which determine a set of cache ways. When a hardware thread is assigned to a COS, it is only allowed to store new cache lines in the ways determined by the COS. However, the COS does not limit reading from any of the ways in the cache.

Intel's API requires that the selected ways in each COS be consecutive. The API does not impose exclusivity, so a cache way can be used by more than one COS.

We experimented with new hardware (Haswell) that supports the new cache allocation technology. It supports only four COSes and requires a minimum allocation of two cache ways for each COS. However, more advanced architectures such as Broadwell will support a minimum of one cache-way allocation and 16 COSes [21].

Intel already added support for CAT to Linux kernel (tip) via the cgroups interface. However, at the time we experimented with the hardware, the modification was only available as a patch and was not stable enough. Therefore, we implemented our own user-level driver that allows control over the COSes and their assignment to CPUs. We implemented it in Python by writing directly to the model specific registers (MSR) using rdmsr/wrmsr utilities for Linux. The driver code is available at `https://bitbucket.org/fonaro/cat-driver`.

3.1 Restricting LLC Access: The Pit

Preventing LLC access to some guests will allow us to allocate more cache ways to others who might benefit more from it. However, the cache allocation API does not allow LLC access to be restricted to specific guests. Instead, we assign them to a single COS we denote the *pit*. We allocate the *pit* the minimum number of cache ways allowed by the hardware (two for our hardware).

4 Cache Auction

Ginseng allocates cache efficiently because it uses a game-theoretic mechanism to elicit the guests' true benefit from cache. The host conducts rounds of cache auctions to adapt the allocation according to the guests' changing needs.

In *Ginseng*, each guest has a different, changing, private (secret) valuation of cache, which is expressed in dollars per second for each cache way allocation. Each way will be allocated exclusively. The guest derives its valuation by combining two private functions: performance as a function of cache ways (in performance units per second) and valuation of performance (in dollars per performance unit). By taking into account resource allocation and monetary worth, *Ginseng* is able to compare valuations, while the actual performance requirements are defined and controlled by the client [14].

As in *Ginseng* for memory allocation [2], we define the aggregate benefit from a cache allocation to all guests—their satisfaction from the 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 cache they receive in the allocation.

Ginseng for *cache allocation* uses the VCG auction, which maximizes social welfare by encouraging even selfish participants with conflicting economic interests to inform the auctioneer of their true valuation of the auctioned goods. In VCG auctions, this is done by charging each participant for the damage it inflicts on other participants' social welfare, rather than directly for the goods it wins. VCG auctions are used, for example, in Facebook's repeated auctions [38], as well as in other settings.

The guest's valuation for each allocation of cache can be affected by its expected performance given its current state. However, it can also be affected by variables unrelated to performance. For example, if the guest is a service provider without any traffic, it may value any number of cache ways as contributing zero to its utility. We denote the guest's valuation by

$$V(cache, state) = V_{perf}(perf(cache, state)) \,,$$

where $V_{perf}(perf)$ describes the value derived by the client for a given level of performance for a given guest, and $perf(cache, state)$ describes the performance the guest can achieve given its current state and a certain number of cache ways. $V_{perf}(perf)$ is private for each client; it is based on economic considerations and business logic.

For example, two clients run a market forecasting algorithm and need to evaluate 1,000 stocks on average to find a group of stocks that are expected to yield 10% profit. They can measure their performance in evaluated stocks per hour. The first client is willing to invest \$10K. For this client, $V_{perf}(perf) = \frac{\$1}{\text{stock}} \cdot perf$. The second client, however, is only willing to invest \$1K. For this client, $V_{perf}(perf) = \frac{\$0.1}{\text{stock}} \cdot perf$. Both clients will need to know $perf(cache, state)$: how many stocks they can evaluate per hour when given various numbers of cache ways and under the current conditions (e.g., server load).

In our experiments, we use an offline mapping of performance as a function of cache and the current server load. We found this to be sufficiently accurate, as we demonstrate in §8.2. But performance can also be measured online, as demonstrated in a number of works [3, 15, 36, 52, 57, 58], and as might be required in real-world scenarios.

5 Auction Protocol

In *Ginseng*, each client pays a constant hourly fee for its guest VM while it is assigned to the *pit*. In each auction round, each guest can bid for exclusive cache ways. After each round, *Ginseng* calculates a new cache allocation, and guests exclusively rent the cache ways they won until the next round ends.

The constant fee is not affected by the auction results. It guarantees the lion's share of the host's revenues, so that the host can utilize the auction to maximize social welfare, thereby attracting more guests.

Clients with hard performance requirements can verify the availability of exclusive cache ways by prepaying for them (and thereby removing them from the cache ways that are up for rent). Supporting these clients will be easier in future hardware with more COSes. These clients are not included in our experiments, which were performed on Haswell. Clients with very low performance requirements are expected to pay in advance only the constant fee and bid with low valuations or not at all, so that they rarely pay for cache ways and manage to stay within their budget. Clients in between those extremes are expected to choose a flexible payment scheme that meets their needs.

Here follows the description of an auction round, along with a numeric example. In the example, as well as in the experiments that follow, the host's clients are service providers with their own customers.

Initialization. Each guest is assigned to the *pit* as it enters the system.

Auction Announcement. The host informs each guest of the number of available cache ways, the server load (i.e., the number of active VMs) and the auction's closing time, after which bids are not accepted. In our example, the physical machine has 20 cache ways, two of which are dedicated to the *pit*, so the host announces an auction for 18 cache ways.

Bidding. Interested guests bid for cache ways. A bid is composed of a price per hour for each number of exclusive cache ways that the guest is willing to rent. In our example, 10 guests choose not to bid in this round, and 2 guests have strict performance requirements: Guest 1 is willing to pay \$1 per hour when allocated 10 or more cache ways and \$0 per hour for fewer cache ways. Guest 2 is willing to pay \$5 per hour for allocation of 14 or more cache ways and \$0 per hour for fewer cache ways.

Bid Collection. The host asynchronously collects guest bids as soon as the auction is announced. It considers the most recent bid from each guest, dismissing earlier bids. Guests that do not bid lose the auction automatically, and are assigned to the *pit*.

Allocation and Payments. The host computes the allocation and payments according to the VCG auction rules, using a specially designed algorithm described in §6. For each guest, it computes how much cache it won and at what price. The payment rule guarantees that the guest will not pay a price that exceeds its bid. The guest's account is charged accordingly (and accurately, by the second). In the example, guest 1 loses, is assigned to the *pit* and pays nothing; guest 2 wins all of the cache ways and pays \$1 per hour.

Informing Guests and Assigning Cache Ways. The host informs each guest of the auction results that are relevant to it: its cache allocation and payment. Then, the host takes cache ways from those who lost them and gives them to those who won, by updating their COSes as necessary.

6 Auction Rules

Every auction has an allocation rule—who gets the goods?—and a payment rule—how much do they pay? To determine who gets the goods, the VCG algorithm calculates the optimal allocation of cache ways: the one that maximizes social welfare—client satisfaction—as described in §4. To determine the optimal allocation, the VCG auction solves a constrained multi-unit allocation problem, as detailed in §6.1. To determine how each client pays, the VCG auction computes the damage it inflicts on other guests, as detailed in §6.2. After explaining the auction rules, we discuss their run-time complexity and provide an example showing how they are executed. Then, we provide a correctness proof.

6.1 Allocation Rule

To find the optimal allocation—the one that maximizes the social welfare—*Ginseng* must consider all the allocations for the number of guests, the number of cache ways available, the size of the *pit*, and the maximum number of classes-of-service (COS) available. Since the number of possible allocations is exponential in the number of guests and cache ways, iterating over them is impractical. Therefore, we introduce a simple algorithm that finds the optimal allocation in polynomial time.

In each iteration, the algorithm reduces the valuation functions of two guests to a combined valuation function. For any number of cache ways and COSes, the combined valuation function finds the allocation which maximized the aggregated valuation of the guests it combines, and returns the aggregated valuation. The iterative algorithm continues until a single combined valuation function remains. This combined function returns the maximal aggregated valuation of all the guests, which is the social welfare. The optimal allocation is then reconstructed from additional data, which is stored during the construction of the combined valuation functions.

6.2 Payment Rule

The payments follow the VCG exclusion compensation principle, as formulated in [2]. Let a_k denote player's k cache allocation, and let a'_k denote the number of cache ways that would have been allocated to guest k in an auction in which guest i did not participate and the rest of the guests bid as they bid in the current auction.

Then guest i is charged a price p_i, computed as follows:

$$p_i = \sum_{k \neq i} V_k(a'_k) - V_k(a_k) \ .$$

The payment reflects the damage that guest i's bid inflicted on other guests.

6.3 Complexity

Let N denote the number of bidding guests. Let W denote the total number of cache ways and let C denote the total number of COSes.

$V_{combined}(w,c)$ is obtained by comparing $O(w \cdot c)$ allocations and summing the two valuations for each allocation. That is, for $W \cdot C$ values, the time complexity is $O\left(W^2C^2\right)$. After $N-1$ reductions we will have one combined valuation. So the total time complexity of the allocation algorithm is $O\left(W^2 \cdot C^2 \cdot N\right)$.

To compute the payment for a guest that is allocated any cache ways, the allocation algorithm needs to be computed again without this guest. Since the number of winning guests is bounded by C, in each auction round the allocation procedure is called up to $\min(C,N)+1$ times, and the time complexity of the total allocation and payment calculation is $O\left(W^2 \cdot C^2 \cdot N \cdot \min(C,N)\right)$.

The relatively small number of guests, ways and COSes on a physical machine result in a reasonable run time for the algorithm: less than one second using a single hardware thread, even in experiments with 24 guests.

6.4 Proof

Definition 1. *For each guest i, we denote its valuation for an allocation of goods a as $V_i(a)$, which is monotonically rising.*

Definition 2. *We denote a group of guests as G.*

Definition 3. *We denote an allocation of ways for each guest $i \in G$ as $A_G = \{a_i\}_{i \in G}$.*

Definition 4. *We denote the aggregated valuation of the group G with the allocation A_G as $V(A_G) = \Sigma_{a_i \in A_G} V_i(a_i)$*

Definition 5. *We denote the sum of goods allocated to the group G with the allocation A_G as $S(A_G) = \Sigma_{a_i \in A_G} a_i$*

Definition 6. *We denote the number of COSes that needed to apply A_G as $C_G = |\{\, a_i \in A_G | \, a_i > 0\}|$*

Definition 7. *For subgroup $X \subseteq G$, we denote $X(A_G) = \{a_i\}_{a_i \in A_G \wedge i \in X}$*

Definition 8. *To shorten, for group $X \subseteq G$, lets denote $V_X(A_G) = V(X(A_G))$, and $S_X(A_G) = S(X(A_G))$*

Definition 9. *We denote $A_G^{w,c}$ as the allocation that maximizes the aggregated valuation of the guests in G for w ways and c COSes.*

Lemma 1. *There exists an allocation $\hat{A}_G^{w,c}$ such that $S\left(\hat{A}_G^{w,c}\right) = w$, $C\left(\hat{A}_G^{w,c}\right) = c$ and $V\left(\hat{A}_G^{w,c}\right) = V\left(A_G^{w,c}\right)$*

Proof. Lets consider $\hat{A}_G^{w,c} = A_G^{w,c}$. If $S\left(A_G^{w,c}\right) < w$ we can add more ways to one of the guests. If $C\left(A_G^{w,c}\right) < c$ we can "use" more COSes. Hence, since the valuation are monotonically rising and more COSes will not change the allocation, $V\left(A_G^{w,c}\right) \leq V\left(\hat{A}_G^{w,c}\right)$. However, since $A_G^{w,c}$ is the optimal allocation then $V\left(A_G^{w,c}\right) \geq V\left(\hat{A}_G^{w,c}\right)$ $\Longrightarrow V\left(A_G^{w,c}\right) = V\left(\hat{A}_G^{w,c}\right)$ □

Definition 10. *According to Lemma 1 we can always consider $S\left(A_G^{w,c}\right) = w$ and $C\left(A_G^{w,c}\right) = c$.*

Lemma 2. *For any subgroup $X \subseteq G$, where $w_X = S_X\left(A_G^{w,c}\right)$ and $c_X = C_X\left(A_G^{w,c}\right)$, then $V_X\left(A_G^{w,c}\right) = V\left(A_X^{w_X,c_X}\right)$.*

Proof. Lets consider the claim is false, then since $A_X^{w_X,c_X}$ is optimal, $V_X\left(A_G^{w,c}\right) < V\left(A_X^{w_X,c_X}\right)$. Thus, we can replace $X\left(A_G^{w,c}\right)$ in the original allocation, with the optimal allocation $A_X^{w_X,c_X}$ and create a new allocation $\hat{A}_G^{w,c}$. Hence, $V\left(A_G^{w,c}\right) = V_X\left(A_G^{w,c}\right) + V_Y\left(A_G^{w,c}\right) < V\left(A_X^{w_X,c_X}\right) + V_Y\left(A_G^{w,c}\right) = V\left(\hat{A}_G^{w,c}\right)$. Contradictory to the optimality of the allocation. □

Theorem 3. *For a group of guests G, amount of ways W and amount of COSes C, the algorithm finds $A_G^{W,C}$ when V_i for each player is monotonically rising.*

Proof. Base case $|G| = 2$: For each number of goods w and COSes c:

Case 1. *If $c \geq 2$, when player 1 allocated a_1, player 2 will most benefit from the maximum goods left available (monotonically rising valuation). Thus, for each possible allocation of goods to player 1 a_1, the best allocation for player 2 is $a_2 = w - a_1$, hence will maximize their valuation sum. If so, then iterating over all the options where $a_1 + a_2 = c$ must find the allocation that maximize the valuation sum between the 2 players.*

Case 2. *If $c = 1$, then it is enough to consider just two options: player 1 gets all the ways or player 2 get all the ways.*

Case 3. *If $c = 0$, there is only one option: both player get nothing.*

Inductive hypothesis: Suppose the theorem holds for all size of G up to some k, $k \geq 2$. Let $|G| = k+1$.

Lets consider any two subgroups: X and Y where $X \cup Y = G$, $X \cap Y = \emptyset$, $|X| > 1$ and $|Y| > 1$. $S_X\left(A_G^{w,c}\right) + S_Y\left(A_G^{w,c}\right) \leq w$ and $C_X\left(A_G^{w,c}\right) + C_Y\left(A_G^{w,c}\right) \leq c$ as they both together must accommodate $A_G^{w,c}$.

According to Definition 10, we can consider: $S_X\left(A_G^{w,c}\right) + S_Y\left(A_G^{w,c}\right) = w$ and $C_X\left(A_G^{w,c}\right) + C_Y\left(A_G^{w,c}\right) = c$.

Lets denote $w_X = S_X\left(A_G^{w,c}\right)$, $w_Y = S_Y\left(A_G^{w,c}\right)$ and $c_X = C_X\left(A_G^{w,c}\right)$, $c_Y = C_Y\left(A_G^{w,c}\right) \Rightarrow w_X + w_Y = w$ and $c_X + c_Y = c$. Thus, according to Definition 2, $V_X\left(A_G^{w,c}\right) = V\left(A_X^{w_X,c_X}\right)$ and $V_Y\left(A_G^{w,c}\right) = V\left(A_Y^{w_Y,c_Y}\right) \Longrightarrow V\left(A_X^{w_X,c_X}\right) + V\left(A_Y^{w_Y,c_Y}\right) = V\left(A_G^{w,c}\right)$.

Hence, if we iterate over all the options where $w_X + w_Y = w$ and $c_X + c_Y = c$, and find $A_X^{w_X,c_X}, A_Y^{w_Y,c_Y}$ for each of them, we must encounter an allocation with the above aggregates valuation.

So, the theorem holds for $|G| = k+1$. By the principle of mathematical induction, the theorem holds for every size of G. □

7 Experimental Setup

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

7.1 Machine Setup

We used a machine with two Intel(R) Xeon(R) E5-2658 v3 @ 2.20GHz CPUs with a 30MB, 20-way LLC that supports CAT. Each CPU had 12 cores with hyper-threading enabled, for a total of 48 hardware threads. One CPU was dedicated to the host and the other to the guests. As many guests as possible were each pinned to two exclusive hardware threads that resided on the same core. In experiments with more than 12 guests, some were pinned to one hardware thread each. This let us manage cache allocation per hardware thread and not per VM's process. The machine had 32GB of RAM per socket. Each VM got 1GB of RAM, pinned to memory from the same node. The host ran Ubuntu Linux with kernel 4.0.9-040009-generic #201507212131, and the guests ran 3.2.0-29-generic-#46-Ubuntu. Each application ran exclusively on a virtual machine (VM); hence we refer from now on to an application and the guest VM running it interchangeably.

7.2 Workloads

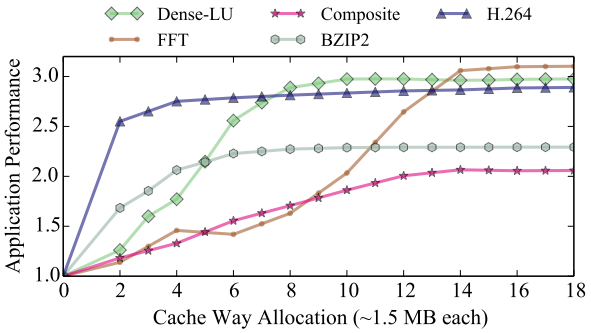
The Phoronix Test Suite [32] includes over 100 benchmarks for a variety of applications. We chose a sample of 10 applications with varying cache utilization, along with their associated benchmarks: ***BZIP2*** (1.5.0) uses parallel compression on a 256MB file. ***H.264*** (2.0.0) encodes a video to H.264 format on the CPU. ***HMMer*** (1.1.0) searches the Pfam database for profile hidden Markov models. ***Gcrypt*** (1.0.3) uses the CAMELLIA256-ECB cipher. ***OpenSSL*** (1.9.0) uses open-source SSL implementation with 4096-bit RSA. Five of the applications were taken from the ***SciMark 2.0 suite*** [45] (1.2.0), which is included in the Phoronix suite but exists also as a stand-alone: ***Fast Fourier Transform*** performs a one-dimensional forward transform of complex numbers. ***Dense LU Matrix Factorization*** computes the LU factorization of a dense matrix using partial pivoting. ***Monte-Carlo*** approximates the value of pi by using random point selection on a circle. ***Jacobi Successive Over-Relaxation*** performs Jacobi successive over-relaxation on a grid. ***Composite-Scimark*** is comprised of several SciMark 2.0 benchmarks.

7.2.1 Classifying the Applications

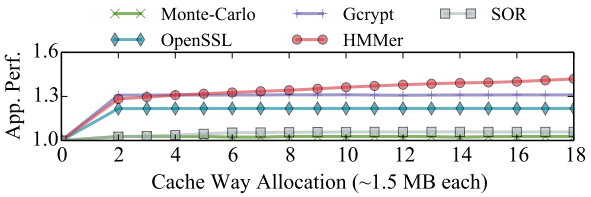
We used the benchmarks to classify the above applications and demonstrate how they perform under different cache allotments and partitioning.

Cache-utilizer applications perform better when allocated more cache. The performance of such applications is depicted in Figure 2a.

Cache-neutral applications cannot utilize the cache to obtain better performance. The performance of such applications is depicted in Figure 2b. However, they might experience minor improvement as compared to being assigned to the *pit*.



(a) Cache-utilizer application performance increases with more ways.

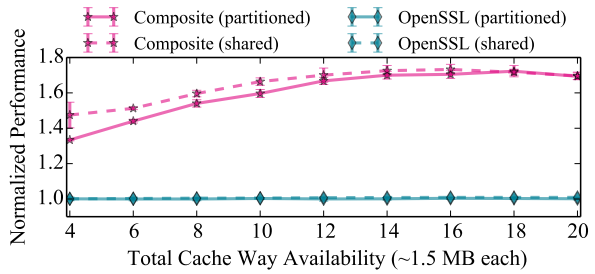


(b) Cache-neutral application performance is indifferent to cache ways.

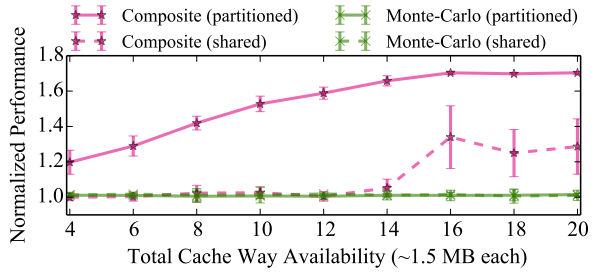
Figure 2: Performance of various applications, normalized by their performance in the *pit*. Measured with 11 other guests assigned to the *pit*. All of the applications were allocated two hardware threads.

Cache-polluter applications are cache-neutral applications that pollute the cache in a way that will harm cache-utilizer performance when cache is shared with the polluter (Figure 3).

It is likely that partitioning the cache can benefit cache-utilizer applications by protecting them from



(a) *Composite-Scimark* (cache-utilizer) performance improves when sharing cache with *OpenSSL* (cache-neutral). Therefore, we consider *OpenSSL* to be a non-polluter.



(b) *Composite-Scimark* (cache-utilizer) performance drops when sharing cache with *Monte-Carlo* (cache-neutral, does not gain from extra cache). Therefore, we consider *Monte-Carlo* to be cache-polluter.

Figure 3: Cache-utilizer application performance drops when sharing cache with a cache-polluter application. In each of the above scenarios, we ran one cache-utilizer and 7 cache-neutral applications simultaneously. We assigned all of the cache to all of the guests in the shared scenario. In the partitioned scenario, we assigned 2 cache ways to all of the cache-neutral applications together and the rest of the available cache was allocated to the cache-utilizer. The performance was normalized to the minimum measurement in all the experiments.

cache polluters without affecting cache-neutral applications. Furthermore, the provider may need to decide how to allocate the cache between several cache utilizers.

7.2.2 Living with Offline Profiling

Offline profiling is error-prone due to the dynamic nature of the cloud. For example, a cache-utilizer may depend on memory bandwidth. That is, if an application can benefit from faster access to the memory via cache, it will likely suffer when memory access time increases due to low memory bandwidth. Memory bandwidth isolation mechanisms have been researched [22, 24, 40, 42], but are not yet available in commercial hardware [36]. Thus, we are compelled to accept the available memory bandwidth as dependent on the number of guests in the cloud. In a real cloud, the client might want to receive information from the host about its available (or expected) memory bandwidth and take it into account when deriving its valuation. In our *Ginseng* experiments, we consider the number of guests in the system to be the only factor influencing memory bandwidth and report it to each guest. The guest uses this information from the host as a factor in its valuation, employing its offline performance profiling for environments with various numbers of guests (Figure 4).

7.2.3 Valuations

The experimental scenario consists of cloud guests who are themselves service providers. Each guest serves one of its customers at a time. Each guest's customer shares performance metrics but has different performance requirements. Thus, when customers change, this implies a change in the guest's valuation function. The valuation function is formulated as the profiled performance function, normalized to the range $[0..1]$, and multiplied by a scale factor that represents the amount the guest's customer is willing to pay for the performance. The scale factor depends on the performance: if it is below the customer's required performance, then the scale factor will be lower. Formally, we can express this as:

$$valuation(a) = s(perf(a)) \cdot \frac{perf(a) - min\_perf}{max\_perf - min\_perf},$$

where a denotes the cache way allocation and s denotes the scale factor. The *pit* is free of charge, and therefore $valuation(0) = 0$.

We characterize three customer types by their scale factors: A ***low-valuation customer*** has a constant scale factor $s = 0.05$. Such a client is unconcerned with performance or unwilling to pay to improve it. An ***medium-valuation customer*** has a scale factor $s = 1$ when meeting its performance requirements, and $s = 0.05$ otherwise. A ***high-valuation customer*** has a scale factor $s = 3$ when meeting its performance requirements, and $s = 0.05$ otherwise. The performance requirements of this type of customer are higher. See, for example, the valuation functions of a customer running *Composite-Scimark* (Figure 5).

In each experiment, each guest serves 10 customers with different valuations, one after the other. We emulated that by giving each guest a pool of valuations with four customer-type distributions. The distributions are denoted as triplets of high-valuation, medium-valuation and low-valuation customers. We experimented with the following distributions: (**1,1,8**), (**1,2,7**), (**0,5,5**), and (**3,3,4**). For each guest we employed a different, randomly shuffled and unique order on those valuation sets. Hence, when we repeated an experiment but with more guests, a guest that participated in both experiments had the same valuation order in both. This gives us an idea

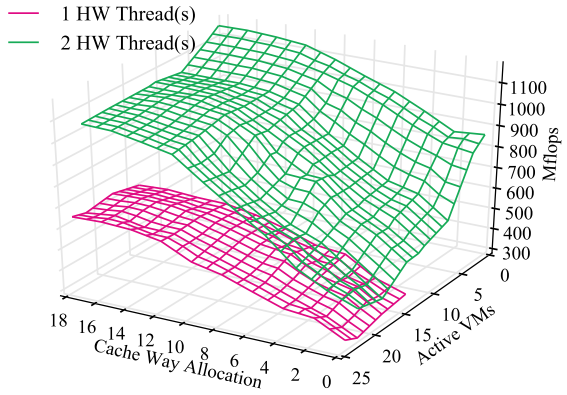
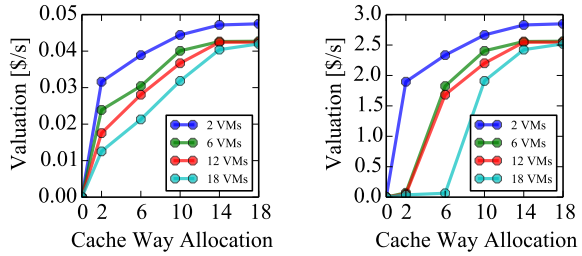


Figure 4: Example of performance profiling: *Composite-Scimark* under different server loads (i.e., active VMs) and with different number of allocated hardware threads.



(a) Low-valuation customer (b) High-valuation customer

Figure 5: *Composite-Scimark* valuation function for different server-loads (i.e., active VMs) and when allocated 2 hardware threads. Note the different scale in the vertical axes.

of what we could achieve if we consolidate more guests on the same physical host.

7.3 Alternative Cache Allocation Methods

We compared *Ginseng* with the following cache allocation methods:

Shared-cache allocation, where all of the guests share the entire LLC. This was the prevalent method prior to the introduction of CAT.

Uniform-static allocation, where each guest is allocated a fixed and equal number of cache ways, as many as the hardware allows. In our hardware there are 4 COSes, so for 4 clients or fewer the cache was divided equally. For more clients, three clients received six cache ways each, and the rest of the clients were assigned to the *pit*.

Performance-maximizing allocation, where the guests' allocation maximizes the overall performance of all of the applications. To this end, we employed *Ginseng*'s optimization algorithm to maximize the aggregate performance by using a constant scale factor $s = 1$ for all the guests' valuations. We did not compare to this method when the experiment had more than one type of application, as the aggregated performance of different applications is meaningless. This allocation is in practice a static allocation, as there is no provider-observable difference in the application's behavior during the experiment.

Ideal-static allocation, where all the future client valuations are known in advance, and the static allocation that maximizes the social welfare is chosen. It serves as an upper bound for all the static allocations.

7.4 Time Scales

Ginseng's responsiveness to guest valuation changes improves with more frequent auctions. Hence, an auction round is conducted every 10 seconds. In each round, the host collects guest bids for 3 seconds, and computes the optimal allocation and payments for at most 3 seconds (in practice it takes well under one second). Then the host notifies the guests of their new allocation and payments and applies the new allocation. However, to gather enough performance measurements for our experiments, we changed the guest's valuation every 5 minutes in dynamic allocation experiments. In static allocation experiments, where valuation changes did not affect the guests' state, 30 seconds were enough.

8 Evaluation

Our experiments were designed to answer the following questions: (1) Which cache allocation method results in guests who are most satisfied (i.e., have the highest social welfare)? (2) How accurate is off-line profiling of guest performance? (3) What are the limitations of a *Ginseng*-based cloud?

The data presented in this paper is based on 1,416 experiments, each lasting 10-50 minutes.

8.1 Comparing Social Welfare

We evaluated the social welfare achieved by Ginseng vs. each of the four other methods listed in §7.3, for all of the workloads and for workloads mixtures (neutrals, utilizers and a mixture of both). We varied the number of guests running the relevant applications. In the mixed workload experiments we cyclically chose the new workload from the set.

The social welfare was calculated from the measured performance of each application using its guest's valuation function. *Ginseng* achieves much better social welfare than the other allocation methods for the tested workloads, as seen in Figure 6. It improves social welfare for *Dense LU Matrix Factorization* by up to $\times$42.8 compared to shared-cache and by up to $\times$26.3 compared to ideal-static. For *Fast Fourier Transform* and *Composite-Scimark*, *Ginseng* improves social wel-

fare between $\times 1.7$ to $\times 17.1$ compared to shared-cache and ideal-static. For a heterogeneous cloud with cache-utilizers, *Ginseng* improves social welfare by up to $\times 13.7$ compared to other allocation methods.

As seen in Figures 7a,7b, *Ginseng* increases the social welfare for an increasing number of up to 12 guests, because more high-valuation and medium-valuation customers can be served simultaneously. However, other methods, including performance-maximizing, improve the social welfare very little or not at all with more guests because they disregard client valuation changes.

For more than 12 guests, hardware threads become a bottleneck, and some guests only get one hardware thread; hence the social welfare gradually declines (Figure 7b). However, under *Ginseng*, some applications can compensate for fewer hardware threads with additional ways, so that *Ginseng* can maintain high social welfare while increasing server consolidation (Figure 7a).

Nevertheless, other allocation methods can still produce results closer to *Ginseng* for some specific scenarios. For example, when all guests run cache-neutral applications (Figure 7c), the applications are less likely to suffer from being consigned to the *pit* than when some guests run cache-utilizer applications. Although their performance does not depend strongly on cache allocation, their performance in the *pit* deteriorates when more guests are assigned to it. Thus, as the number of guests in the cloud increases, it becomes increasingly important to allocate cache ways to the right guests, as opposed to assigning them to the *pit*.

Shared-cache can produce better results than *Ginseng* when all guests use applications with a small memory working-set (Figure 7d). In such a case, cache-misses are rare (e.g., a solid 80% hit ratio for 12 *H.264* with shared-cache). Thus, because none of the applications access the memory frequently, an application is expected to consume the maximum memory bandwidth when it does. Hence, memory bandwidth will not be a bottleneck in this case. However, when memory bandwidth is low due to frequent memory access by other applications, even a rare memory access can dramatically affect on performance. This is illustrated in Figure 9a, where two applications are *H.264* and 10 applications are *Monte-Carlo*. *H.264* uses a small memory working-set but relies on prefetching to improve performance. When the cache is not shared, all the prefetched data remains in the cache, resulting in better performance. When the two *H.264* applications share the cache, and the 10 *Monte-Carlo* applications are assigned to the *pit*, some of the *H.264* data might be evacuated from the cache. Because the *Monte-Carlo* applications access the memory very frequently, any memory access by the *H.264* applications will result in sharply decreased performance of the latter.

8.2 Influence of Off-Line Profiling

We experimented with off-line performance profiling data (e.g., Figure 4) that was measured in a controlled environment. However, in a live environment, profiling data should be collected on-line, so that it remains fresh under changing conditions. To retrospectively justify the use of off-line profiling in our experiments, we measured the deviation of actual performance from performance predicted by the off-line profiling (for the conditions at the time).

In Figure 8, we see that the deviation from the expected performance was under 10% in most cases. Moreover, the median deviation for all the applications was under 1%, and 95% of the measurements deviate from the predicated performance by less than 12%. The accuracy of the profiling is reflected in the small difference between *Ginseng* and the simulation (Figure 7), and shows that a more accurate profiler would achieve only a minor improvement.

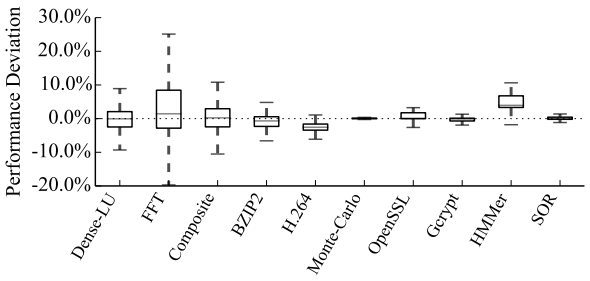
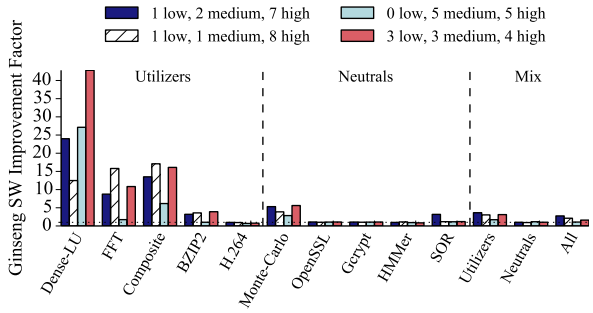


Figure 8: Expected performance deviation for all applications in all of our experiments. The *pit* measurements are excluded as the performance is expected to fluctuate when sharing a small number of cache ways.

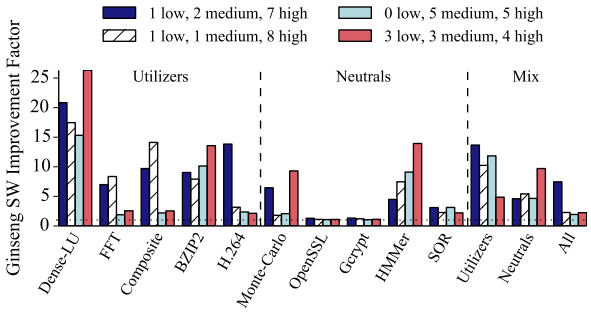
8.3 To Share or Not To Share

We have already seen cases where partitioning the cache can benefit cache-utilizer applications without affecting cache-neutral ones. However, in some cases, a partitioning that includes limited sharing could greatly improve overall performance.

We consider two possible simple partitioning schemes where we reserve two cache ways for the *pit*. ***Hard-partitioning*** allocates a set of exclusive cache ways to each guest. A guest that values cache more than others will be allocated more cache ways. ***Soft-partitioning*** allocates all the cache ways to the guest that values cache the most. The guest that values cache second-most gets a subset of the previous guest's ways, and so forth. For simplicity, we only let guests bid for the right to use fixed COSes (for example: $COS_1 = [1..2]$ (pit), $COS_2 = [3..20], COS_3 = [3..15], COS_4 = [3..10]$). Guests will need to consider how they value these COSes, as-

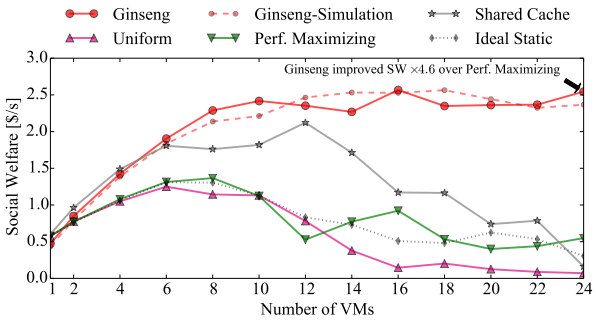


(a) *Ginseng* improvement factor over shared-cache allocation method.

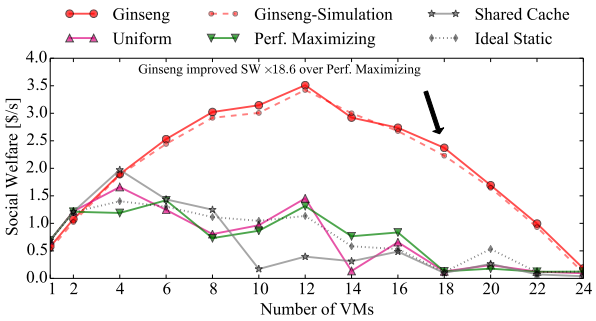


(b) *Ginseng* improvement factor over ideal-static allocation method.

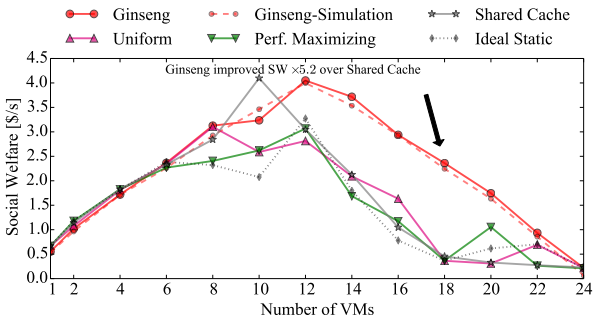
Figure 6: Maximum improvement factor of *Ginseng* compared to the shared-cache and ideal-static methods with different assumptions on the number of high, medium, and low valuation customers. The maximum is over any number of guests with the application, or mixture of applications.



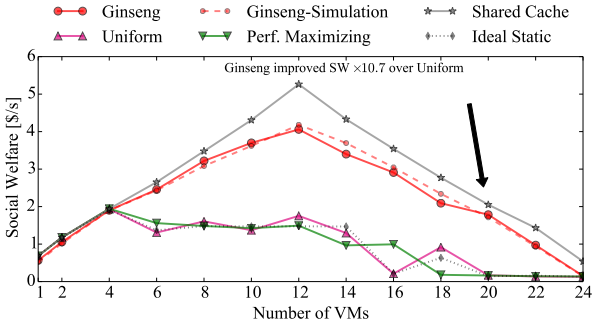
(a) All guests run *Fast Fourier Transform* with 1 high-valuation customer, 1 medium-valuation customer and 8 low-valuation customers. *Ginseng* improves social welfare by up to ×4.7 over performance-maximizing and by up to ×15.8 over shared-cache.



(b) All guests run *Dense LU Matrix Factorization* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. *Ginseng* improves social welfare by up to ×18.6 over performance-maximizing and by up to ×24 over shared-cache.



(c) All guests run *Monte-Carlo* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. *Ginseng* outperforms other allocation methods as server consolidation is increased, even for cache-neutral applications.



(d) All guests run *H.264* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. This is the only case where shared-cache outperforms *Ginseng* for any number of guests.

Figure 7: Social welfare under different cache allocation methods as a function of the number of guests. The dashed lines indicate an experiment where the clients perform in an identical way to the profiler (artificial clients). Cache-utilizer application can greatly benefit from *Ginseng*. Cache-neutral applications can still enjoy the benefits of *Ginseng*, albeit to a lesser extent. Applications with a small memory working set will prefer sharing the cache with others like it.

suming other COSes may be occupied by at most one application per COS.

As we have seen, guests can successfully estimate their expected performance for a given allocation of exclusive cache (i.e., hard-partitioning). However, it is harder to valuate a given soft-partitioning allocation when the cache is shared with an unknown guest, as is common in the cloud. Even if the neighbor guest is known, the performance and valuation still depend on additional dimensions (quantity and share level) that further complicate the bidding and optimization process for guest and host alike.

Ginseng uses hard-partitioning due to its simplicity and accuracy of estimation. In this section we assess the benefit guests might have achieved from soft-partitioning. To simplify, we tested several pairs of cache-utilizer applications, and the *pit* contained 10 *Monte-Carlo* applications that served as cache-polluters. We measured the performance of each pair for all possible cache allocations in the hard-partitioning and soft-partitioning allocation schemes. Then, we compared each pair's performance in these settings. We used each application's measured performance, normalized to its performance when assigned to the *pit*, as its valuation function, and experimented with different ratios of scale factor between each pair's valuations.

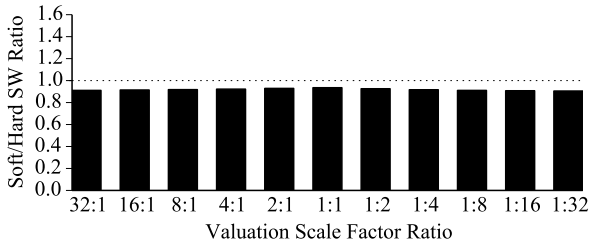
Although soft-partitioning sometimes yields better social welfare than hard partitioning (Figure 9b), it usually improves it by no more than 10% (Figures 10 and 9a), or even degrades it.

8.4 Dynamic Allocation Overhead

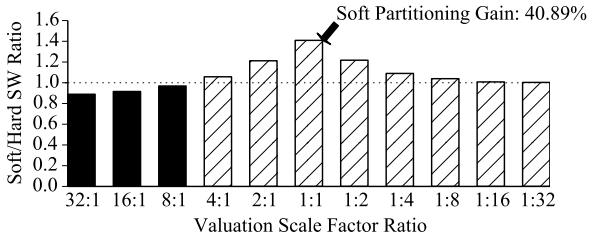
Transferred cache ways require a warm up period. Moreover, they are likely to contain the previous application's data. If the previous application is a cache-utilizer, it is likely to access this data soon, and have this data marked as most-recently-used (MRU). This creates competition for the other application. If it accesses its own data too slowly, it may end up evicting that data from its previously owned ways to store new data in the cache. It will thus take longer (possibly forever) for the second application to benefit from additional cache ways. We refer to such a scenario as *cache leakage*.

Furthermore, any allocation change is constrained by the need to preserve the consecutiveness of ways in a COS. For example, let the initial allocation be $COS_1 = [1..4]$, $COS_2 = [5..6]$ and $COS_3 = [7..10]$. To transfer a way from COS_3 to COS_1, COS_2 must also change. The least disruptive transfer moves two ways: way 7 to COS_2 and way 5 to COS_1. Compared with the required transfer of a single way, this consecutiveness-constrained transfer doubles the *cache leakage* effect.

We measured how dynamic allocation changes affect application performance. In each experiment, a guest machine ran one of the workloads listed in §7.2. At the same time, the host natively ran an application that repeatedly touches all its data, in parallel, using 8 hardware threads and by utilizing the CPU's out-of-order-execution (OOOE) mechanism. We designed this application to ensure that its data fits perfectly in its allocated cache ways. By keeping its cache lines marked as MRU, it amplifies the *cache leakage*, and thus represents a worst-case scenario.



(a) Both applications are *H.264*. Hard-partitioning yields better social welfare than soft-partitioning.



(b) Both applications are *Fast Fourier Transform*. The maximal improvement for soft-partitioning over hard-partitioning is achieved when the applications' scale factors are equal.

Figure 9: Social welfare under hard vs. soft-partitioning. Striped columns indicate better social welfare under soft-partitioning. Black indicates the opposite.

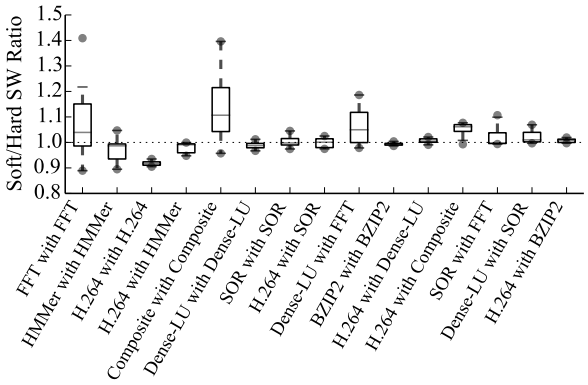


Figure 10: Social welfare improvement under soft-partitioning compared with hard-partitioning for various application pairs. Boxes show the middle 50% of the values over different valuation scale factor ratios. Whiskers mark extreme values.

Each experiment ran for 10 minutes. In each experiment both applications were allocated a basic set of ways. Another set was transferred between the applications every [10..60] seconds. The numbers of basic and transferred ways were in the range [2..10].

In the baseline experiments the cache ways were transferred once, from the application, to ensure that the application's performance was not affected by the *cache leakage*. Half the performance measurements in these experiments were high and half were low.

In the experiments with the frequent transfer intervals, there is a similar performance distribution, whose values varied by up to 4% from the baseline values (high values were lower, low values were higher). The mean performance over the duration of the experiment varied from the baseline by up to 1.1% for all of the workloads. Mean performance values did not depend on the transfer frequency: the effect of a single transfer is negligible, and when there are many intermittent leaks, those that benefit an application will compensate for those that harm it.

9 Related Work

Market Driven Resource Allocation. Lazar and Semret auctioned bandwidth [33]. Agmon Ben-Yehuda et al. introduced Ginseng as a memory auctioning platform [2]. Drexler and Miller [8] and Maillé and Tuffin [39] suggested an auction to compute a market clearing price for memory and bandwidth, respectively. Waldspurger et al. auctioned processor time slices [55].

Cache Partitioning [50]. Many hardware solutions detect cache pollution by non-reused data and prevent its future insertion, or apply partitioning to prevent the application from interfering with other applications [9, 10, 23,26,35,44,46]; others rely on the user or OS to allocate the cache, like CAT does [6,28,34,49]. However, CAT is the first hardware implementation of such a mechanism in commodity hardware.

A cache-polluter can prevent caching of specific data by using Intel's non-temporal store instruction. Cache can be partitioned in software using page-coloring [53] to prevent cache pollution: by the program [5, 52], by the OS [15, 37, 58], and under virtualized environments [25, 48, 56]. Some works proposed to guarantee the applications' performance demands via LLC management [14,16,17,36,41,47,51]; these works require the guests to reveal their performance requirements without any incentive to do so.

Shared Cache Performance Interference. VM *Performance interference* when sharing LLC [12,29,31] was analyzed and predicted. Such methods can help guests estimate their performance on shared cache and allow a biddable soft-partitioning scheme.

10 Conclusions and Future Work

Ginseng efficiently allocates cache to selfish black-box guests while maximizing their aggregate benefit. *Ginseng* can also benefit private clouds, where it distinguishes between guests that perform the same function for different purposes, such as a test server vs. a production server. Cache-*Ginseng* is the first economically-based cache allocation method, and cache is the second resource implemented in the *Ginseng* framework. Cache-*Ginseng* works by hard-partitioning the cache in short intervals according to a VCG auction in which the guests have an incentive to bid their true valuation of the cache.

The guests utilize their cache fast enough to allow such rapid changes in the allocation without any substantial effect on their performance. *Ginseng* achieves up to $\times$42.8 improvement in social welfare when compared with alternative cache allocation methods. Shared cache allocation may improve on these results. Formulating a bidding and valuation language for shared cache remains as future work.

Although the VCG auction has a high computational complexity, it is a suitable cache auction because of its coarse allocation granularity. Similarly, it can be efficiently used to allocate other small numbered multi-unit resources whose valuation functions are monotonically rising: CPUs, for example. Thus, *Ginseng* is not only a platform for auctioning cache and memory, but also a concrete step toward the Resource-as-a-Service (RaaS) cloud [1], in which all resources, not just cache and memory, will be bought and sold on-the-fly. Extending *Ginseng* to additional resources and to their concurrent allocation remains as future work.

11 Acknowledgments

We thank Sharon Kessler, Nadav Amit, Avi Mendelson, Vikas Shivappa, Priya Autee, Edwin Verplanke and Matt Fleming for fruitful discussions. This work was partially funded by the Hasso Platner Institute, by the Professor A. Pazi Joint Research Foundation and by the Israeli Ministry of Science. We thank Intel for loaning the hardware that facilitated the research.

References

[1] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (raas) cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, p. 12.

[2] AGMON BEN-YEHUDA, O., POSENER, E., BEN-YEHUDA, M., SCHUSTER, A., AND MU'ALEM, A. Ginseng: Market-driven

memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2014), VEE '14, ACM, pp. 41–52.

[3] ALBONESI, D. H. Selective cache ways: on-demand cache resource allocation. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (1999), IEEE, pp. 248–259.

[4] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[5] BUGNION, E., ANDERSON, J. M., MOWRY, T. C., ROSENBLUM, M., AND LAM, M. S. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 244–255.

[6] CHIOU, D., JAIN, P., RUDOLPH, L., AND DEVADAS, S. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference* (New York, NY, USA, 2000), DAC '00, ACM, pp. 416–419.

[7] CLARKE, E. Multipart pricing of public goods. *Public Choice 11*, 1 (Sept. 1971), 17–33.

[8] DREXLER, K. E., AND MILLER, M. S. Incentive engineering for computational resource management. *The ecology of Computation 2* (1988), 231–266.

[9] DYBDAHL, H., AND STENSTRÖM, P. Enhancing last-level cache performance by block bypassing and early miss determination. In *Advances in Computer Systems Architecture*, C. Jesshope and C. Egan, Eds., vol. 4186 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 52–66.

[10] GONZÁLEZ, A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (New York, NY, USA, 2014), ACM, pp. 217–226.

[11] GORDON, A., HINES, M., DA SILVA, D., BEN-YEHUDA, M., SILVA, M., AND LIZARRAGA, G. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. *Proc. RESoLVE* (2011).

[12] GOVINDAN, S., LIU, J., KANSAL, A., AND SIVASUBRAMANIAM, A. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM.

[13] GROVES, T. Incentives in teams. *Econometrica: Journal of the Econometric Society* (1973), 617–631.

[14] GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. Quality of service shared cache management in chip multiprocessor architecture. *ACM Trans. Archit. Code Optim. 7*, 3 (Dec. 2010).

[15] GUO, R., LIAO, X., JIN, H., YUE, J., AND TAN, G. Nightwatch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX Association, pp. 307–318.

[16] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*, M. van Steen and M. Henning, Eds., vol. 4290 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 342–362.

[17] HASENPLAUGH, W., AHUJA, P. S., JALEEL, A., STEELY, S., AND EMER, J. The gradient-based cache partitioning algorithm. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012).

[18] HEO, J., ZHU, X., PADALA, P., AND WANG, Z. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Integrated Network Management, 2009. IM '09. IFIP/IEEE International Symposium on* (June 2009), IEEE, pp. 630–637.

[19] HINES, M. R., GORDON, A., SILVA, M., DA SILVA, D., RYU, K., AND BEN-YEHUDA, M. Applications know best: Performance-driven memory overcommit with ginkgo. In *2011 IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)* (Nov. 2011), IEEE, pp. 130–137.

[20] INTEL. Improving real-time performance by utilizing cache allocation technology. Website, Apr. 2015.

[21] INTEL OPEN SOURCE.ORG. Cache monitoring technology, memory bandwidth monitoring, cache allocation technology & code and data prioritization. https://01.org/packet-processing/cache-monitoring-technology-memory-bandwidth-monitoring-cache-allocation-technology-code-and-data, Jan. 2016.

[22] IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), vol. 35 of *SIGMETRICS '07*, ACM, pp. 25–36.

[23] JAIN, P., DEVADAS, S., AND RUDOLPH, L. Controlling cache pollution in prefetching with software-assisted cache replacement. *Comptation Structures Group, Laboratory for Computer Science CSG Memo 462* (2001).

[24] JEONG, M. K., EREZ, M., SUDANTHI, C., AND PAVER, N. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference* (New York, NY, USA, 2012), DAC '12, ACM, pp. 850–855.

[25] JIN, X., CHEN, H., WANG, X., WANG, Z., WEN, X., LUO, Y., AND LI, X. A simple cache partitioning approach in a virtualized environment. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on* (Aug. 2009), IEEE, pp. 519–524.

[26] JOHNSON, T., CONNORS, D., MERTEN, M., AND HWU, W. Run-time cache bypassing. *IEEE Transactions on Computers 48*, 12 (Dec. 1999), 1338–1354.

[27] KANG, H., AND WONG, J. L. To hardware prefetch or not to prefetch?: A virtualized environment study and core binding approach. *SIGPLAN Not. 48*, 4 (Mar. 2013), 357–368.

[28] KASERIDIS, D., STUECHELI, J., AND JOHN, L. K. Bank-aware dynamic cache partitioning for multicore architectures. In *2009 International Conference on Parallel Processing (ICPP)* (Sept. 2009), IEEE, pp. 18–25.

[29] KASTURE, H., AND SANCHEZ, D. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 729–742.

[30] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[31] KOH, Y., KNAUERHASE, R., BRETT, P., BOWMAN, M., WEN, Z., AND PU, C. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software* (Apr. 2007), IEEE, pp. 200–209.

[32] LARABEL, M., AND TIPPETT, M. Phoronix test suite. Website, 2011.

[33] LAZAR, A., SEMRET, N., AND OTHERS. Design, analysis and simulation of the progressive second price auction for network bandwidth sharing. *Columbia University (April 1998)* (1998).

[34] LEE, H., CHO, S., AND CHILDERS, B. R. Cloudcache: Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (Feb. 2011), IEEE, pp. 219–230.

[35] LIN, W.-F., AND REINHARDT, S. K. Predicting last-touch references under optimal replacement. *Ann Arbor 1001* (2002), 48109–2122.

[36] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ACM, pp. 450–462.

[37] LU, Q., LIN, J., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on* (2009), IEEE, pp. 246–257.

[38] LUCIER, B., LEME, R. P., AND TARDOS, E. On revenue in the generalized second price auction. In *Proceedings of the 21st International Conference on World Wide Web* (New York, NY, USA, 2012), WWW '12, ACM, pp. 361–370.

[39] MAILLÉ, P., AND TUFFIN, B. Multi-bid auctions for bandwidth allocation in communication networks. In *IEEE INFOCOM* (2004).

[40] MURALIDHARA, S. P., SUBRAMANIAN, L., MUTLU, O., KANDEMIR, M., AND MOSCIBRODA, T. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44, ACM, pp. 374–385.

[41] NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 237–250.

[42] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair queuing memory systems. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on* (Washington, DC, USA, Dec. 2006), IEEE, pp. 208–222.

[43] OU, Z., ZHUANG, H., NURMINEN, J. K., YLÄ-JÄÄSKI, A., AND HUI, P. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2012).

[44] PIQUET, T., ROCHECOUSTE, O., AND SEZNEC, A. Exploiting single-usage for effective memory management. In *Asia-Pacific Computer Systems Architecture Conference* (2007), Springer, pp. 90–101.

[45] POZO, R., AND MILLER, B. Scimark 2.0. *URL: http://math. nist. gov/scimark2* (2000).

[46] QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2007), ISCA '07, ACM, pp. 381–391.

[47] RAFIQUE, N., LIM, W. T., AND THOTTETHODI, M. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2006), PACT '06, ACM, pp. 2–12.

[48] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2009), CCSW '09, ACM, pp. 77–84.

[49] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: Scalable and efficient fine-grain cache partitioning. *SIGARCH Comput. Archit. News 39*, 3 (June 2011), 57–68.

[50] SCOLARI, A., SIRONI, F., SCIUTO, D., AND SANTAMBROGIO, M. D. A survey on recent hardware and software-level cache management techniques. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on* (Aug. 2014), IEEE, pp. 242–247.

[51] SHARIFI, A., SRIKANTAIAH, S., MISHRA, A. K., KANDEMIR, M., AND DAS, C. R. Mete: Meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2011), SIGMETRICS '11, ACM, pp. 13–24.

[52] SOARES, L., TAM, D., AND STUMM, M. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, Nov. 2008), MICRO 41, IEEE Computer Society, pp. 258–269.

[53] TAYLOR, G., DAVIES, P., AND FARMWALD, M. The tlb slice—a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1990), ISCA '90, ACM, pp. 355–363.

[54] VICKREY, W. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance 16*, 1 (Mar. 1961), 8–37.

[55] WALDSPURGER, C. A., HOGG, T., HUBERMAN, B. A., KEPHART, J. O., AND STORN, W. S. Spawn: a distributed computational economy. *Software Engineering, IEEE Transactions on 18*, 2 (Feb. 1992), 103–117.

[56] WANG, X., WEN, X., LI, Y., LUO, Y., LI, X., AND WANG, Z. A dynamic cache partitioning mechanism under virtualization environment. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on* (June 2012), IEEE, pp. 1907–1911.

[57] YE, C., BROCK, J., DING, C., AND JIN, H. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. 1–15.

[58] ZHANG, X., DWARKADAS, S., AND SHEN, K. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 89–102.