Placement of Virtual Containers on NUMA systems: A Practical and Comprehensive Model

Justin Funston[*], Maxime Lorrillere[*], Alexandra Fedorova[*], Baptiste Lepers[†]
David Vengerov[‡], Jean-Pierre Lozi[‡], Vivien Quéma[§]
[*]*University of British Columbia,* [†]*EPFL,* [‡]*Oracle,* [§]*IMAG*

Abstract

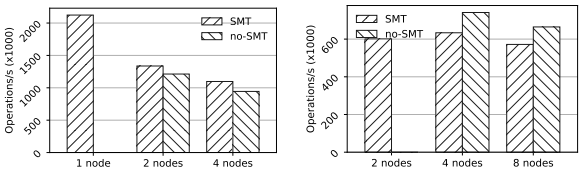
Our work addresses the problem of placement of threads, or virtual cores, onto physical cores in a multicore NUMA system. Different placements result in varying degrees of contention for shared resources, so choosing the right placement can have a large effect on performance. Prior work has studied this problem, but either addressed hardware with specific properties, leaving us unable to generalize the models to other systems, or modeled much simpler effects than the actual performance in different placements.

Our contribution is a general framework for reasoning about workload placement on machines with shared resources. It enables us to build an accurate performance model for any machine with a hierarchy of known shared resources *automatically*, with only minimal input from the user. Using our methodology, data center operators can minimize the number of NUMA (CPU+memory) nodes allocated for an application or a service, while ensuring that it meets performance objectives.

1 Introduction

We address the problem of placing a virtual container on a multicore NUMA system. Hardware resources allocated to a container determine how well it performs and how much energy it consumes. Roughly speaking, there are two decisions affecting which hardware resources are allocated to a container. First, the user decides how many cores, memory and perhaps other resources the container requires. Second, the system software decides, when launching the container, how to map the container's virtual cores onto physical cores. Our work proposes a solution for automatically making this second decision.

The placement of virtual cores onto physical cores can have a large and unpredictable effect on performance. Consider the following experiment, where we use MongoDB's WiredTiger key-value store [3] running a B-tree search using 16 threads. We run this application in a `lxc` container on an Intel NUMA system and on an AMD NUMA system (Fig. 2 provides their overview). Suppose our goal is to maximize the throughput. How should we place this container? Assuming that each virtual core gets its own physical core, how do we place virtual cores onto NUMA nodes? Do we squeeze them onto as few nodes as possible?, do we spread them evenly across all nodes?, or do we use a middle ground?



(a) WiredTiger, Intel (b) WiredTiger, AMD

Figure 1: Throughput of the WiredTiger key-value store on two NUMA systems.

As Fig. 1 shows the right answer can vary greatly from one system to the next. On the Intel system, the application performs significantly better when all of its threads run on a single node. On the AMD system, four nodes are better than two, only if we do not use SMT, but using eight nodes does not buy you better performance[1].

There is a number of factors responsible for this dissimilar behaviour. When all threads are squeezed into a single NUMA node, they experience more resource sharing: on the Intel system they have no choice but to share the SMT pipeline and the L3 cache. Resource sharing can be contentious (where threads compete for cache space and hardware queues [34]) or cooperative (where threads pre-fetch data for each other [27]). Furthermore, with all the threads running on a single NUMA node,

[1]A single-node configuration for AMD is not shown: we used 16 virtual cores, so we could not fit them onto a single node (with 8 cores) while ensuring that each virtual core gets its own physical core.

cross-thread communication has a lower latency, because it occurs via the L3 cache, as opposed to a slower cross-chip interconnect. Apparently, the benefits of faster communication and cooperative resource sharing outweigh the cost of resource contention on the Intel system, but not on the AMD system.

The best-performing placement for a container, therefore, is difficult to predict. The problem becomes even harder if the goal is to not only pick the best-performing placement, but to achieve a trade-off between the number of used nodes and performance. Our work proposes a solution that works as follows:

Step 1: The user provides a simple abstract specification of the shared resources present on the target hardware. Identifying the right level of abstraction was key to being able to automatically construct models for different target systems. The abstraction we propose in this work is called *scheduling concerns* (§4).

Step 2: Using shared resource specification, our algorithm generates, for a given container size, a list of *important placements* – placements that will likely yield different performance on the target hardware[2]. This step is crucial for automatically training a model: while the total number of potential placements is measured in billions (making training infeasible), the number of important placements is only a couple dozen. We introduce the concept of important placements and propose the algorithms for automatically generating them (§4).

Step 3: Using our script, the user trains a machine learning model for the target hardware and the target number of vCPUs (§5). Unlike prior work, we do not rely on hardware performance events (HPE) as model features. Manual selection of the right HPEs puts too much burden on the user. Automatic selection turned out to be impractical on modern machines with 1000s of HPEs. Instead, our model uses as inputs actual performance measurements obtained in two different placements. This approach makes our model-building methodology easier to port across hardware, reduces the training time and achieves higher accuracy, compared to using HPEs.

Step 4: The scheduler runs the virtual container in two different placements, for a couple of seconds in each, feeds their performance into the model and obtains a vector of predicted performance values in each important placement. Using this vector, the system decides what placement to use and remaps the virtual cores. Since the container needs to run in two placements, its memory may need to be migrated if these placements do not use the same NUMA nodes. We improve on memory migration in Linux and evaluate its overhead in §7.

Our solution enables determining the best placement for a *specific virtual container*, but not how to interleave different containers on the same NUMA node. Some data center operators we spoke to do not interleave containers, others do, so we leave that decision up to the operator. Going back to the scenario in Fig.1, using our tools the scheduling system can quickly decide that on the Intel host it is sufficient to provide a single NUMA node for the key-value store in order to maximize its throughput. Then the remaining nodes can be used to host other containers. We believe that our techniques can be used to build scheduling systems that pack virtual containers onto physical hardware more efficiently.

Our main contribution is ***abstractions and methodology for constructing accurate and portable models for predicting performance of a container in various placements on NUMA systems, regardless of what shared resources are present***. We evaluate it by automatically generating performance models for two different hardware systems and measuring their accuracy using cross-validation (§6). We also present a use case demonstrating how the model could be used in practice (§7).

2 Background and Related Work

Workload placement on multicore systems has been explored for over a decade. Early studies examined contention between single-threaded applications for a specific resource, such as the SMT instruction pipeline [25, 14], or shared caches and memory controllers [34, 12, 21, 31]. Later work extended the techniques to multi-threaded workloads and to additional resource combinations, such as SMT and shared caches [33], memory controllers and the shared interconnect [9, 18]. While laying a crucial foundation for our work, these prior techniques did not provide a general solution for reasoning about such systems. For instance, while the work of Zhuravlev et al. [34] showed us how to avoid interference for shared memory controllers and the work of Lepers et al. [18] showed us how to place applications on machines with asymmetric interconnects, we still do not know how to build a model that combines both concerns.

Techniques used in prior work did not allow for automatic combination of several models. Every model required manual design: careful selection of hardware performance events [34, 17, 18, 9, 12] or even manual crafting of artificial "probe" workloads or "Rulers" [31, 33] that must be run side-by-side with the target workloads to determine their sensitivity to contention.

Dwyer used an automated model-building methodology, where automatically selected features (from all HPEs available on the machine) were fed into a variety of machine-learning models [11]. However, the model predicted a rather simple outcome: a performance degradation when a target workload was co-scheduled with an interfering one, and not the performance in different place-

[2] §4 defines the *important placement*.

ments. Consistent with our finding that HPEs observed in a single placement are poor model features, Dwyer's study reported rather poor prediction accuracy.

A recent system, Pandia [15] not only accurately predicts performance of different workload placements on a multicore NUMA machine, but also predicts how an application would perform with different numbers of threads. Unforunately, to make predictions, Pandia requires performance observations of six runs with different thread counts, which is difficult to do online because most real applications cannot easily reconfigure their thread count on demand. Despite addressing many limitations of previous work, fundamentally Pandia still relies on the machine-specific modelling methodology that prevents easily transferring results to other systems. Pandia's authors capture factors that contribute to performance, such as cache contention, latency of communication, and load balancing, in a set of machine-specific equations. If the model had to be adapted to another machine, the equations would have to be manually reformulated.

We believe that investing that much effort into designing new models for every new type of hardware puts an unreasonable burden on system engineers. Instead, we sought a future-proof methodology that uses easily available information about a machine's shared resources and automatically builds an accurate performance model.

There are two recent studies that address a different problem, but use techniques that could be adopted in our work. CherryPick [4] handles cloud configuration options, like CPU count, amount of RAM, disk speed, and network speed, but not multicore resources. CherryPick uses Bayesian optimization to minimize the number of search configurations needed and achieves high accuracy despite the non-linearity of performance. The Bayesian optimization approach could potentially work well with our goal and resources, and is a possible avenue of future work. PARIS [30] is similar to CherryPick in that it handles the same type of resources and its goal is to help cloud customers choose the correct cloud configuration. It does not abstract or handle multi-core resources.

3 Assumptions and Limitations

Identically scored placements yield identical performance. As we explain in §4, a placement is identified by the degree of sharing for each hardware resource, to which we refer as the *score*. A vector of scores identifies a placement. Placements with identical score vectors are deemed to yield identical performance for a given workload. This assumes that our machine model must be informed about all shared resources that might affect performance. Most solutions in this space also assume awareness of all shared resources. A radically different approach would be a statistical technique that searches for an optimally performing placement by trying a sufficient number of random placements [23]. Unfortunately, the best known techniques require trying *thousands* of placements and assume that performance in all placements fits a Generalized Pareto distribution — an assumption that does not hold in our case.

A workload is encapsulated in a virtual container. Data centers use virtualization for a variety of reasons, so this assumption dovetails with our target environment. Managed cloud environments present their offerings as a menu of virtual instances with a fixed number of vCPUs per instance (see [1], for example). As a result, we can feasibly train a separate model for each type of hardware and each vCPU count; we do not have to incorporate the effects of varying number of threads into the model, which would make it more complicated. We are not addressing the problem of finding the optimal number of threads or vCPUs for the workload; for that, users can resort to other tools [15, 26].

A NUMA node is a unit of resource allocation. Our solution predicts the performance of a container in all important placements, provided that the target container does not share NUMA nodes with other containers. Unused NUMA nodes can be safely used to run other containers without interference as long as those nodes do not share the interconnect – a condition that can be automatically checked using the machine specification[3]. Suppose that the "best" number of NUMA nodes chosen for a container gives us more physical cores than the container needs. Then the remaining cores would be left idle if no other containers used them. Some data center operators find this acceptable, reasoning that the cost of leaving cores idle is negligible relative to missing performance targets; others contend that maximizing the utilization of physical cores is very important. Our solution does not dictate the decision. If the operator chose to interleave containers on NUMA nodes, our modeling techniques would need to be extended to provide performance predictions under interleaving. Another alternative would be to only interleave with "safe" containers, e.g., those with low CPU utilization or otherwise known to cause negligible interference. We leave the exploration of these scenarios to future work.

We consider only balanced placements. A balanced placement is one where the number of vCPUs is evenly divisible by any number of shared resource units considered for placement. For instance, if we have shared L3 caches on the system, we will only consider placements where the number of vCPUs sharing each L3 cache is equal. Uneven sharing can cause unpredictable performance effects on the workload, for example by creating

[3] Experimental results confirming this statement are available [13].

stragglers, so we choose to not model these effects.

4 Abstract machine model

A major obstacle to a solution to the placement problem is the sheer number of possible placements. For 16 virtual cores on a 64 core system, the number of possible placements is the combinations of 16 objects chosen from a set of 64, which is on the order of 10^{14}. It is essential to exploit the symmetry in the system to reduce the number of placements to a manageable number. By this we mean that for most types of shared resources it does not matter *which* shared resources are being used but *how much* of the shared resources is available to the workload. For instance, for the workload in Fig. 1 on the AMD system, it does not matter which L3 cache it uses, all that matters is whether it has two, four or eight L3 caches at its disposal.[4]

We tackle this with the concept of ***scheduling concerns***. A single scheduling concern is responsible for a single hardware resource, or an inseparable set of hardware resources that affect the performance of vCPU placements. The primary purpose of a scheduling concern is to provide a numerical ***score*** when given a vCPU placement. The score represents the *static* utilization of the particular resource, meaning that it only depends on the vCPU placement, not the dynamic behavior of a workload. A simple example is an "L2 cache" resource. If in a given placement all the virtual cores share a single L2 cache, the score for the L2 cache scheduling concern will be equal to one. If in another placement the cores are spread over two L2 caches, the score will be equal to two, and so on. So, two placements might use completely different NUMA nodes and physical cores, but if they use the same number of L2 caches then they will both have the same L2 cache score. From the vantage point of the L2 cache, these placements will be *identical* in terms of performance. For non-symmetrical resources, such as the cross-chip interconnect on some systems, instead of counting how many links are used by a placement in order to obtain the score, we would add up the total available bandwidth of all links used by a placement. A vector of numeric scores for all scheduling concerns uniquely identifies each placement that is distinct with respect to sharing of resources. Placements with identical vectors are deemed identical with respect to resource sharing, so we can discard the duplicates when training our model. ***By considering only the placements with distinct score vectors, we substantially reduce the space of relevant placements and make the problem tractable.***

[4]The exception is asymmetric resources, for instance if one NUMA node is positioned closer to the system NIC than others; our model allows caputring this asymmetry.

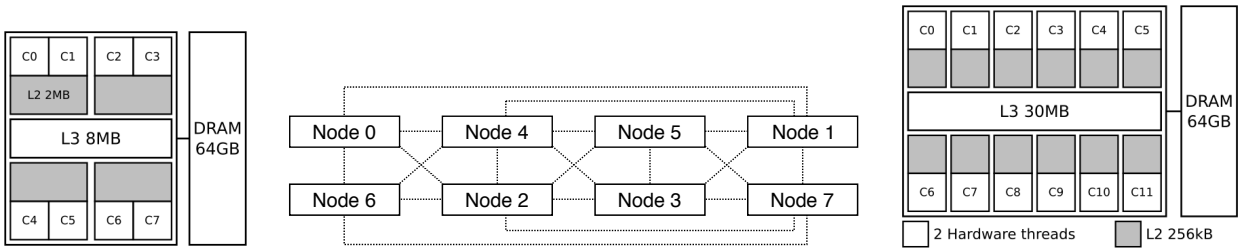
There are two additional pieces of information a scheduling concern needs in order to identify the important placements. The first is whether the concern's score is proportional to the user's cost, which is the case for resources like NUMA nodes because fewer nodes (lower score) means more containers can be packed onto a system. If a lower score for a resource only meant worse performance, we could simply discard placements with a lower score for that resource (all other scores being equal) from our list of important placements. But since we want users to be able to make cost-performance trade-offs, placements with lower scores but potentially lower cost could still be relevant. The second piece of information needed by a scheduling concern is whether the resource encompassed by a concern can ever have an inverse relationship with performance. For some resources, like the L2 cache, a higher score is usually better, but for some workloads such as those showing cooperative cache sharing, a smaller score (using fewer L2 caches) may actually improve performance. For other resources, like the shared interconnect described below, a lower score will never improve performance and would not result in a lower cost for the user, so we can safely ignore placements with lower scores when all else is equal.

In practice, a single scheduling concern may cover multiple shared resources because some resources are inseparable with respect to thread placement. Threads sharing a physical core via SMT typically share a cache, the instruction front-end, and functional units. In cases like this, a single scheduling concern is still sufficient.

Our AMD system (Fig. 2) has multiple NUMA nodes, an asymmetric interconnect, and a form of SMT. For this system we developed the scheduling concerns shown in Table 1. For the L2/SMT and L3 concerns, the score for a particular placement can be calculated directly from information provided by the operating system. The OS also provides information on the interconnect topology, but it is simpler and more accurate to measure the aggregate bandwidth with a benchmark (e.g. *stream* [20]) for each possible combination of nodes.

For example, for a 16-vCPU container in an eight-node placement without SMT the score vector for the AMD system is [16, 8, 35000], because this placement uses 16 L2 caches (16 hardware threads, one per cache), eight L3 caches (8 nodes) and has an IC bandwidth of 35GB/s. For the same placement, but with SMT, the score vector would be [8, 8, 35000], because on each node two hardware threads would be collocated on the same L2 cache, so we would use half the L2 caches than in the case without SMT.

Each concern is relatively easy to implement, and can be developed independently. Since it does not require a performance expert, we envision the specification of concerns being provided as part of system BIOS. ***Over-***



(a) AMD Opteron 6272 node (b) AMD interconnect (c) Intel Xeon E7-4830 v3 node

Figure 2: The two systems used in our study. The first is a quad AMD Opteron 6272. It has eight NUMA nodes (schematically shown in Figure 2a) connected with an asymmetric interconnect (Figure 2b) and a total of 64 cores. Pairs of cores share the instruction front-end, L2 cache, and floating point units. The second system is a quad Intel Xeon E7-4830 v3 with four NUMA nodes (Figure 2c) and 96 hardware threads (12 physical cores per node with SMT). The interconnect (not shown) is symmetric.

Concern	Score	Resources	Cost?	Inverse Perf Possible?
L2/SMT	Number of L2 caches in use	L2 cache, instruction fetch and decode, and floating point units	Y	Y
L3	Number of L3 caches in use	L3 cache, memory controller, and bandwidth to DRAM	Y	Y
Interconnect	Aggregate bandwidth between nodes in use	Interconnect bandwidth	N	N

Table 1: Scheduling concerns used on our AMD test system (shown in Figure 2).

all, we found scheduling concerns to be a powerful abstraction that enables encoding shared resources on a variety of hardware and makes the model easy to port to new hardware.

Next, from the concerns and hardware topology we need to derive the *important placements*. An important placement must have a score that ensures it satisfies three properties: **(1)** conform to our balanced assumption, **(2)** be feasible: i.e., not assign more than one vCPU to a single hardware thread, and **(3)** not be superseded by a strictly better placement.

Given a score s and the number of vCPUs v, the balance property is encoded as $v \bmod s = 0$, and the feasibility property is encoded as $v/s \leq Capacity$, where capacity is the number of hardware threads available in a single instance of the resource if applicable: e.g. there are eight hardware threads per L3 cache on our AMD test system. We also define the *Count* of a concern as the total number of that resource on the system, so our AMD test system has an *L2Count* of 32 for example. The first step in generating important placements is generating the possible scores that satisfy the balance and feasibility requirements individually. This is done for each scheduling concern that can affect cost or have an inverse relationship with performance. For our AMD test system this step is shown in Algorithm 1.

Now that we have all balanced and feasible placements, and before filtering the duplicates, we need to enumerate all possible placements whose performance the scheduler might want to predict if more than one container were running on the system. For example, suppose that after placing one container onto two nodes on the system, the scheduler might want to place other containers on the remaining nodes, so it should be able to predict the performance on any combination of those nodes. Therefore, we must keep track of possible placements on those remaining nodes in order to properly train the model. The packings are generated with a recursive method shown in Algorithm 2. On our AMD system, we use the L3 scores because the L3 scheduling concern corresponds to NUMA nodes, and NUMA nodes are our unit of resource allocation (see §3).

Next, as shown in Algorithm 3, packings that are duplicates and packings that are not Pareto-efficient with respect to the interconnect score are filtered out (since the interconnect concern does not affect cost and cannot have an inverse relationship with performance). Because the L2 and L3 scores can affect cost or have an inverse relationship with performance, placements are not filtered based on them.

As an example of a Pareto-efficient packing, on our AMD system we need to keep the 4-node placement that

Algorithm 1 Generating possible L2 and L3 scores

```
L3Scores = List()
for i ← 1, L3Count do
    if v/i ≤ L3Capacity ∧ v mod i = 0 then
        L3Scores.append(i)
    end if
end for
L2Scores = List()
for i ← 1, L2Count do
    if v/i ≤ L2Capacity ∧ v mod i = 0 then
        L2Scores.append(i)
    end if
end for
return L3Scores, L2Scores
```

Algorithm 2 Generating packings of placements

```
Packings = List()
procedure GENPACK(L3Scores, NodesLeft, Current)
    for all L3S in L3Scores do
        if L3S > len(NodesLeft) then
            continue
        end if
        for all n in Combinations(NodesLeft, L3S) do
            Remaining = NodesLeft - n
            NewPacking = Current.append(n)
            if len(Remaining) > 0 then
                GenPack(L3Scores, Remaining,
                        NewPacking)
            else
                Packings.append(NewPacking)
            end if
        end for
    end for
end procedure
return Packings
```

uses nodes $\{2,3,4,5\}$ because it is the 4-node placement with the highest interconnect score. Therefore the placement using nodes $\{0,1,6,7\}$ is also an important placement and will be kept because it is the placement that can be packed with the best 4-node placement. Continuing, suppose that we consider a 4-node placement that uses nodes $\{0,1,4,5\}$. If we were to use this placement at runtime, the remaining set of four nodes, potentially used for another workload, is $\{2,3,6,7\}$. Both of these placements have poor interconnect scores, in part because there is a two-hop distance between nodes $\{0,5\}$ and nodes $\{3,6\}$. Instead, we can pack the machine with a better combination of 4-node placements: $\{0,2,4,6\}$ and $\{1,3,5,7\}$. Using this observation, the vectors for placements $\{0,2,4,6\}$ and $\{1,3,5,7\}$ will be kept over the worse pair of 4-node placements.

Algorithm 3 Generating important placements

```
Nodes = range(0, L3Count)
Packings = GenPack(L3Scores, Nodes, List())
Remove duplicates from Packings
for all (a,b) in Permutations(Packings, 2) do
    if L3 Scores in a ≠ L3 Scores in b then
        continue
    end if
    aIC = Sorted interconnect scores of a placements
    bIC = Sorted interconnect scores of b placements
    ToRemove = True
    for i in range(0, len(aIC)) do
        if aIC[i] > bIC[i] then
            ToRemove = False
        end if
    end for
    if ToRemove then
        Remove a from Packings
    end if
end for
ImportantPlacements = List()
for all Placements p in Packings do
    n ← L2Count/L3Count
    L3S = L3 Score of p
    for all L2S in L2Scores do
        if n · L3S ≥ L2S then
            ImportantPlacements.append(p)
        end if
    end for
end for
return ImportantPlacements
```

After this process is complete, we are left with the important placements. For our AMD system we have 13 of them: two 8-node placements (one sharing L2 caches and one not), three 2-node placements (with the best and second-best interconnect score, and one placement used to pack when specific 4-node placements are used), and eight 4-node placements (half sharing L2 caches, half not, and various interconnect scores relevant for packing). Our Intel test system (Fig. 2), on the other hand, only uses an L2/SMT concern and an L3 concern. With 24 virtual cores per container, it has seven important placements which are all of the placements that satisfy the balance and feasibility constraints: a one node placement sharing L2 caches, two 2-node placements, two 3-node placements, and two 4-node placements.

5 Performance Predictions

Automatic model-building techniques learn how to map a set of features describing data to a predicted outcome.

The outcome we would like to model is a vector of performance values in all important placements, relative to a baseline placement. For example, if there are three important placements, and the performance in the second and third is 20% and 30% better than that in the first baseline placement, the performance vector will be: $[1.0, 0.8, 0.7]$. Our data elements are executions of workloads in different placements, and the features are some metrics describing the execution.

Model-building methodology and feature selection. To build a model, we use a multi-output *Random Forest* regressor (RF). RF is a machine learning technique known for its ability to learn non-linear functions with very little or no tuning. More complex techniques, like deep neural networks, can yield slightly higher accuracy, but require substantial tuning and are prone to overfitting, especially if not given the "right" features.

Any modelling technique requires predictive input features. Feature selection turned out to be a challenge. In the past, to model performance on multicore systems, researchers used hardware performance events as inputs to the model. In most cases, the HPEs were selected manually[5], which required substantial insight into the intricacies of hardware architecture and its effects on software. Our goal was to make model training automatic, so manual HPE selection was not an option. Automatic selection, on the other hand, turned out to be impractical.

Modern machines have many hundreds of HPEs, some more than 1000 [32]. Automatic feature selection would measure *all* HPEs during training and use feature selection to identify the best predictors. Only four HPEs can be measured at a time, because there is usually only four hardware counter registers, so measuring all the HPEs for the entire training set can take weeks (66 days on our Intel machine), even if we use sampling.

In an effort to find an acceptable compromise, we first used a combination of the manual and automatic approaches. We started with a set of plausible features (41 HPEs on the Intel test system and 25 the AMD) covering cache, memory, TLB, interconnect, and pipeline behaviour, which are metrics commonly used in similar work. We then used *Sequential Forward Selection* [10, 16] (SFS) to pick the best ones. The final RF model would take a vector of selected HPEs observed in a single baseline placement as the input and produce the performance vector as the output. Even after this rigorous feature selection process, we were not happy with the accuracy (see §6).

Finally, we designed a solution that is more robust, requires little training time, and is largely automatic. Instead of relying on HPEs describing various architectural events, we rely on *observations of actual performance in two different configurations from the set of important placements*. Performance can be measured using instructions per cycle (IPC), transactions per second, or any other application-specific metric – the only requirement is that it must be possible to obtain this metric online. Specifying the performance metric for a container is the only manual part of the process. Beyond that, the training process automatically finds the two of the important placements that give the highest accuracy when used as inputs to the model. The final model takes as inputs the performance observations in these two placements and outputs the predicted performance vector. A separate model is trained for each number of vCPUs used in virtual containers.

The downside of this approach is that at runtime we have to run the container in two placements instead of one before obtaining the predictions, but the advantage is that the predictions are more accurate and we do not have to use the time-consuming feature selection process for each new target hardware.

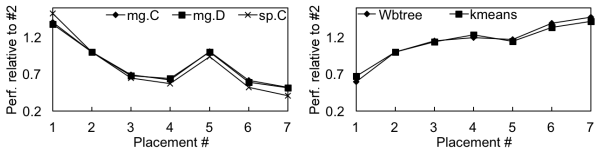


Figure 3: Performance relative to the baseline placement (#2) for workloads in two example clusters on Intel.

Why do performance observations have good predictive ability? Empirical evidence suggests that workloads naturally fall into several categories, according to the shapes of their performance vectors. Figure 3 shows two example categories on the Intel system. As we can see, the vectors within the category are almost identical, but the vectors in different categories are very distinct.

To generate these categories we used k-means clustering. To automatically determine the best value for k, we select the k that maximizes the average Silhouette coefficient [24, 2] over all data points, which is the standard practice in the field. This clustering method produced six categories on our systems (full results are reported in [13]). This suggests that workloads may naturally form distinct categories depending on their performance trends. For example, workloads that are not memory intensive and are not adversely affected by sharing SMT contexts could belong to the same category (where thread placement does not matter). Another category could be one where using fewer NUMA nodes and fewer physical cores greatly hurts performance, and so on. Then there is no surprise that a ML model could quickly narrow down the category, and hence the shape of the per-

[5]Dwyer's [11] and Zellweger's [32] works are the only exceptions known to us.

formance vector, from two observations of performance.

6 Evaluation

In this section we focus on evaluating the accuracy of predictions. Since our training method does not require automatic feature selection, training the model takes seconds. The algorithms used to determine important placements also run in a matter of seconds. The inference time is negligible (milliseconds).

We had three model variants to compare: the first one used as inputs the actual performance measurements observed in two important placements, the second used only the HPEs observed in a single placement, the third used both. The third variant did not improve accuracy over the first one, so we do not include the data for it.

The set of applications we experimented with are drawn from the NAS Parallel Benchmark suite [6], Parsec suite [7], the Metis map-reduce benchmarks [19], and BLAST [5]. Also included are the Linux kernel compile gcc benchmark, two Spark graph workloads, TPC-C [28] and TPC-H [29] on Postgres and a WiredTiger [3] BTree benchmark. Workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system (Fig. 2). Within containers, the number of application threads is set so as to achieve >70% CPU utilization on each core, typical of what is done in practice.

Figure 4 show the actual and predicted performance for each workload for important placements on the AMD and Intel systems. The x-axis shows the IDs of the important placements, numbered 1–13 on the AMD system and 1–7 on the Intel system. The y-axis shows the performance in the placements relative to the baseline. Placement #1 was used as the baseline for the AMD system, and placement #2 for the Intel system – the baseline placement can be any of the two placements whose performance is required as the input to the model.

The results are per-application cross-validated. For example, when training the model that will be used for predicting a Spark workload neither the data from *spark-cc* (a Spark connected components algorithm run on the LiveJournal database) nor *spark-pr-lj* (a PageRank algorithm run on the LiveJournal database) is included in the training. We cross-validated every workload, but for space constraints we omit the results for most of the NAS and Parsec benchmarks. They are qualitatively similar to others and we are happy to provide them upon request.

Overall the accuracy when using only the actual performance measurements as model features is high. The predicted performance is within 4.4% of actual on average on the AMD system, and within 6.6% on Intel. A couple of exceptions are the cases where the training set did not include any workloads that behaved similarly to the predicted benchmark, for example *kmeans* on the AMD system, which was the only benchmark in our training set that preferred SMT, or *canneal* on Intel.

Prediction accuracy when using only the HPEs from a single placement was a lot less reliable. On the AMD system it produced good results overall, but the accuracy was still noticeably worse compared to the model variant that relied only on actual performance measurement. On Intel the model relying only on HPEs produced many poor predictions. It completely missed the performance trend for *ft.C* and *freqmine*, produced errors of over 40% for *kmeans* and *WTbtree*, and is noticeably worse for several other workloads.

An example of why HPEs observed in a single placement could have poor predictive power, and one of the reasons why the Intel system produced worse predictions, is predicting the effect of inter-thread communication latency. There is a huge latency difference for communication between a single-node placement and placements including more than one node. For some applications, reduced inter-thread communication latency when all threads are running on a single node has a major performance impact, as is the case for *WTbtree*. Separating the sensitivity to latency from overall memory intensiveness (which can be measured by the cache miss rate) is difficult to do with HPEs. Similarly, it is also very difficult to determine if a workload's working set will fit in a given number of L3 caches by only measuring HPEs on a single placement. ***We conclude that using actual performance observations as model features is likely to produce higher accuracy, in addition to being a more practical method of training the model, than using selected architectural events.***

7 Using the model in practice

There are many ways in which data center operators can use our model. To illustrate one potential use case we set up a scenario, where the user would like to pack as many instances of a given virtual container into a physical server while respecting a performance target. For demonstration of the complete solution we implemented its prototype (covering steps 1-4 in §1). To assess the overheads, we measure the costs of container migration.

We use virtual containers of three types: WiredTiger running a B-tree search workload, Postgres running TPC-H, and Spark running PageRank on a LiveJournal database. For clarity, we present the results of homogeneous configurations, where many containers of the same type are packed into each system. Performance results with our model in heterogeneous configurations can be inferred from these figures, because different containers collocated on the same system do not interfere with our approach. Actual data can be provided upon request.

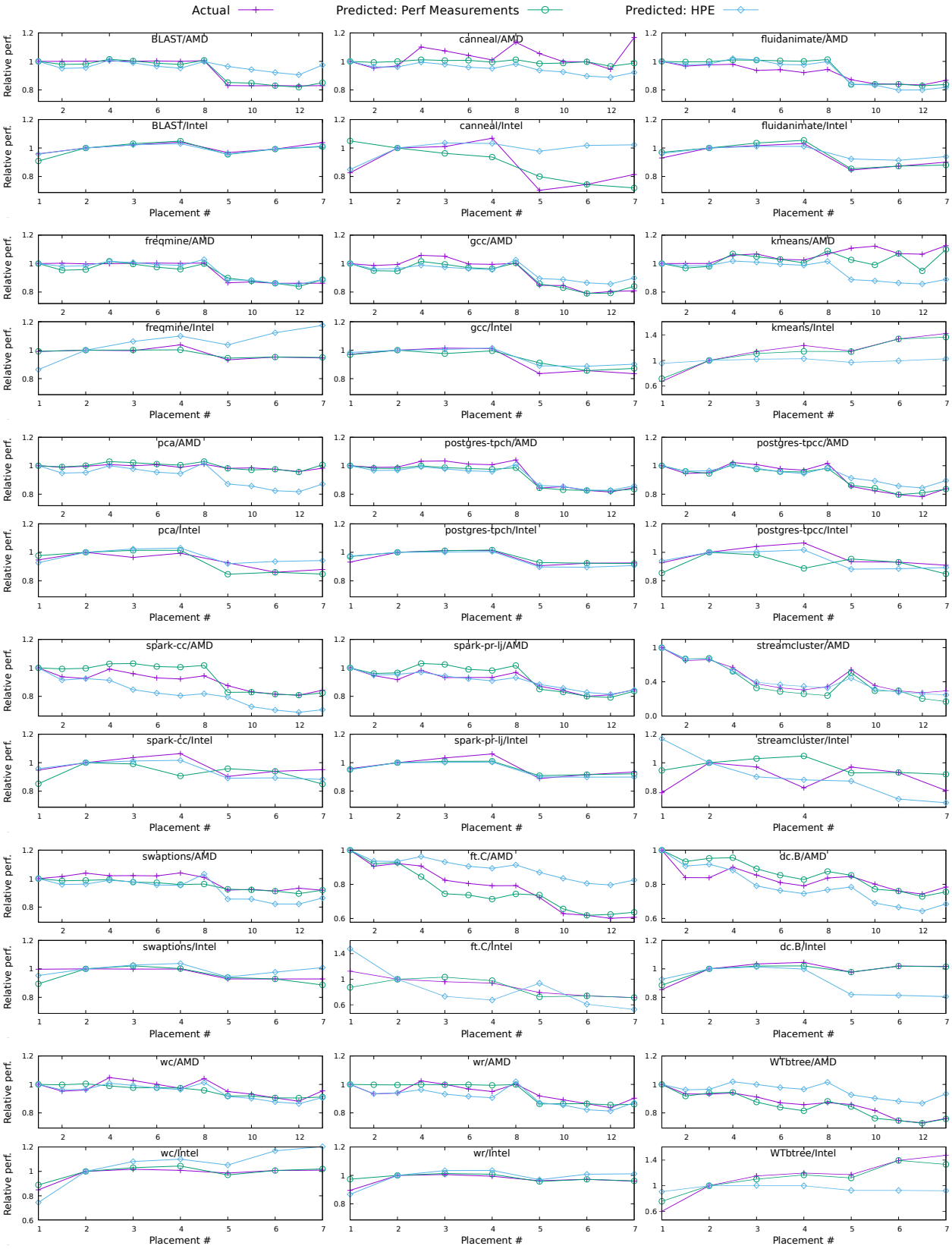
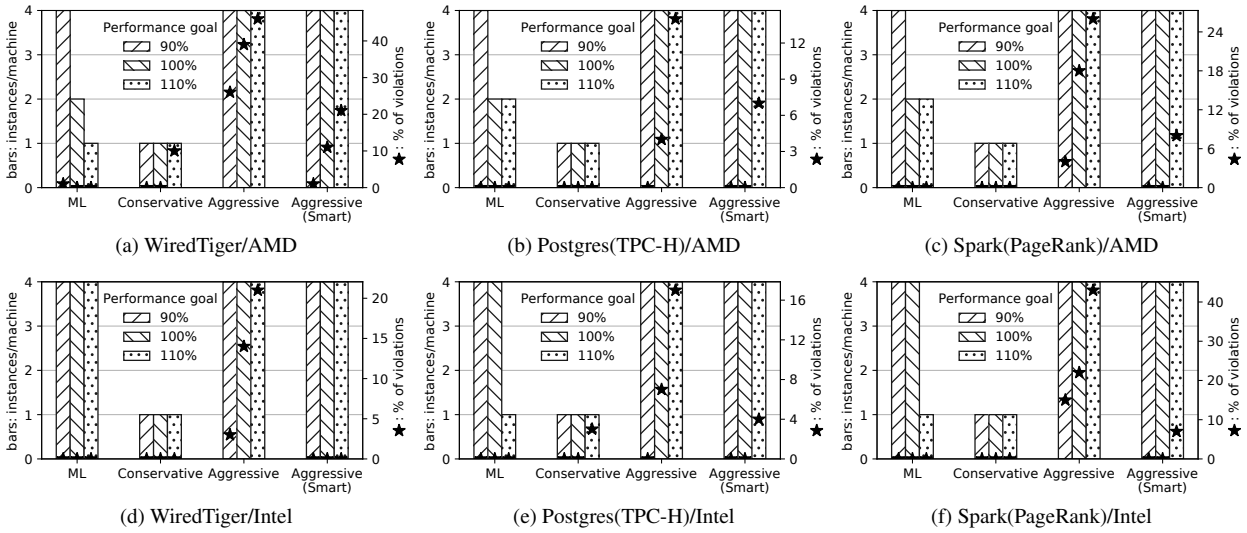


Figure 4: Accuracy of predictions.



(a) WiredTiger/AMD (b) Postgres(TPC-H)/AMD (c) Spark(PageRank)/AMD

(d) WiredTiger/Intel (e) Postgres(TPC-H)/Intel (f) Spark(PageRank)/Intel

Figure 5: Instances per machine (left y-axis, higher is better) and % performance goal violation (right y-axis, lower is better).

The performance goal can be specified in terms of an application-level metric such as transactions per second or a generic metric such as instructions-per-cycle. The placement policy is agnostic to the metric used and only requires that the application make this metric available at runtime. For simplicity, we set the performance goals to correspond to 90%, 100% and 110% of the performance observed in the baseline placement.

We compare four hypothetical container placement policies. The first policy, referred to as **ML**, is based on our techniques. It decides how many nodes to allocate to the container based on performance observations in two placements and the model presented in the previous section. It runs the workload in two placements during the first few seconds of the execution without interrupting the workload, and then migrates it into the best predicted placement. To separate various aspects of performance, the results shown here do not include the migration overhead; it is studied separately in the next section. The second policy, **Conservative**, is a naïve policy that allocates the entire machine to each instance, allowing only one instance per machine. The third policy, **Aggressive**, is another simple policy that fills the system with as many instances as possible, maximizing machine utilization at the risk of performance violations. For example, our AMD system allows up to four 16-core instances and our Intel system up to four 24-core instances. Neither Conservative nor Aggressive pin vCPUs to cores, allowing Linux to perform the mapping in the way it wishes, and possibly creating unneeded contention. We also evaluate a more sophisticated fourth policy, **Smart-Aggressive**. This policy is similar to Aggressive, except each instance is pinned to the best minimum set of nodes, which we define as having the highest interconnect bandwidth. This policy requires an analysis of the interconnect topology in order to find the correct set of nodes.

We could not make a fair comparison to any other method presented in earlier work. As we explained in §2, most earlier models targeted very different systems and most did not predict performance vectors, so we could not apply them directly.

We evaluate the policies by measuring how many instances of the same workload they were able to pack per machine (higher is better) and the degree of violation of the performance goal as the percent of the target (lower is better). All workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system. Figure 5 show the results for the three container types. The bars show the number of instances packed (left y-axis), while the "stars" shows the deviation from the target performance goal, expressed as percentage (right y-axis).

The ML policy always meets the performance goal while in most cases packing more instances per machine than the conservative scheduler. The conservative policy almost always packs fewer instances per machine than ML, but also, surprisingly, may cause performance target violations, because Linux may map vCPUs unevenly to shared resources, causing unnecessary contention.

The aggressive policy packs a maximum possible number of containers per machine, at the cost of performance target violations, up to 46% with WiredTiger on AMD, and 43% with Spark on Intel. It is surprising that even when the aggressive policy packs the same num-

ber of containers per machine as the model-based policy, it still often reports a higher violation percent. That is because this policy allows virtual containers to share NUMA nodes. Smart-aggressive addresses this shortcoming, but even that policy can cause performance violations (e.g., 20% for WiredTiger on AMD), because it does not take into account all ways in which workload placement might affect performance.

Memory migration overhead. Memory migration in Linux is known to be inefficient [18]. Since we need to measure the performance of workloads in two or three configurations, fast memory migration is needed in that phase to reduce overheads. Lepers et al. [18] propose a method that freezes the application and migrates pages with concurrent worker threads. We improve on this by migrating the page cache and reducing locking overhead. Table 2 shows migration times for the workloads of §5. We observed similar results on the Intel system. Note that page cache migration time is counted with our method only since Default Linux doesn't support it – and yet, it can be a large part of migration overhead (93% with BLAST, 75% with TPC-C and 62% on TPC-H). We are able to migrate a large amount of memory in a few seconds, usually one order of magnitude faster than Default Linux (38× faster for Spark). Linux is especially inefficient for workloads with many processes such as TPC-C, since it has per-task overhead linked to updating the `cpuset` at each migration.

A drawback of our method is that it requires freezing the container during migration in order to reduce contention on some critical kernel locks. It is therefore suitable for non-latency-sensitive workloads. For latency-sensitive workloads, we have the option of not freezing the container and to instead throttle the bandwidth given to the migration process so as to reduce the impact on the running application. Thus, the migration takes more time but with a smaller impact on the running container. Using this method, the overhead of migration for the WiredTiger workload[6] is between 3% and 6%, and the migration takes 60 seconds. In comparison, Linux takes 43.8 seconds, has a overhead of 20% at best and completely freezes the applications for several seconds. It also does not migrate the page cache.

Overall, we observe that the migration overhead is proportional to the amount of memory used by the container, except in cases with extremely high thread counts. Using the container's memory footprint, the user can estimate whether the migration cost warrants an online deployment of the placement algorithm, or if it is preferable to use it offline for placement of recurring jobs.

[6]We picked WiredTiger for this evaluation since other the other workloads we use don't report the evolution of performance during the execution.

Benchmark	Memory (GB)	Fast Migration (s)	Default Linux (s)
BLAST	18.5	3.0	5.9
canneal	1.1	0.3	3.9
fluidanimate	0.7	0.3	2.3
freqmine	1.3	0.3	4.2
gcc	1.4	0.3	2.8
kmeans	7.2	1.5	6.5
pca	12.0	2.8	10.0
postgres-tpch	26.8	5.8	117.1
postgres-tpcc	37.7	14.9	431.0
spark-cc	17.0	3.7	139.9
spark-pr-lj	17.1	3.8	137.0
streamcluster	0.1	0.1	0.4
swaptions	0.01	0.1	0.0
ft.C	5.0	1.3	19.4
dc.B	27.3	5.4	51.7
wc	15.4	3.4	19.5
wr	17.1	3.6	18.9
WTbtree	36.3	6.3	43.8

Table 2: Migration performance on the AMD system, compared to the default Linux migration method. The amount of memory includes processes' memory and the page cache associated with the container.

8 Conclusion

Modern multicore systems have a complex hierarchy of shared resources and performance can vary wildly depending on how virtual CPUs are mapped to hardware contexts. Operators waste resources and money by using conservative and sub-optimal placement policies. We have shown a solution to this problem using a methodology to abstract a system's shared resources, identify important placements, and predict their performance. Our method can lead to very significant advantages in machine utilization while keeping performance guarantees.

CPU architecture is continually changing, often by sharing resources between cores in new ways, in order to continue scaling the core count. AMD's newly introduced Zen architecture [8] has L3 cache sharing separate from sharing the memory controller. Intel's Haswell-E architecture has asymmetric links between NUMA nodes through its cluster-on-die feature [22], which has unique performance implications different from other asymmetric architectures. The flexibility of our methods means that they can be used on systems like these or future architectures without significant retooling by an expert.

References

[1] Amazon EC2 Instance Types. `https://aws.amazon.com/`

ec2/instance-types.

[2] Selecting the number of clusters with silhouette analysis on KMeans clustering. http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html.

[3] The WiredTiger key-value store. http://www.wiredtiger.com.

[4] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 469–482.

[5] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. Basic local alignment search tool. *Journal of molecular biology 215*, 3 (1990), 403–410.

[6] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, L., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. The Nas Parallel Benchmarks. *IJHPCA 5*, 3 (1991), 63–73.

[7] BIENIA, C., AND LI, K. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (2009).

[8] CLARK, M. A New, High Performance x86 Core Design from AMD. In *Hot Chips: A Symposium on High Performance Chips* (2016).

[9] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic management: A holistic approach to memory placement on numa systems. *SIGPLAN Not. 48*, 4 (Mar. 2013), 381–394.

[10] DRAPER, N. R., AND SMITH, H. *Applied regression analysis*. John Wiley & Sons, 1966.

[11] DWYER, T., FEDOROVA, A., BLAGODUROV, S., ROTH, M., GAUD, F., AND PEI, J. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 83:1–83:11.

[12] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques* (Washington, DC, USA, 2007), PACT '07, IEEE Computer Society, pp. 25–38.

[13] FUNSTON, J. R. *A model for thread and memory placement on NUMA systems*. PhD Dissertation, University of British Columbia https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0363032, 2018.

[14] FUNSTON, J. R., EL MAGHRAOUI, K., JANN, J., PATTNAIK, P., AND FEDOROVA, A. An smt-selection metric to improve multithreaded applications' performance. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2012), IPDPS '12, IEEE Computer Society, pp. 1388–1399.

[15] GOODMAN, D., VARISTEAS, G., AND HARRIS, T. Pandia: Comprehensive contention-sensitive thread placement. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 254–269.

[16] JOHN, G. H., KOHAVI, R., AND PFLEGER, K. Irrelevant features and the subset selection problem. In *Machine learning: proceedings of the eleventh international conference* (1994), pp. 121–129.

[17] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. Using os observations to improve performance in multicore systems. *IEEE Micro 28*, 3 (May 2008), 54–66.

[18] LEPERS, B., QUÉMA, V., AND FEDOROVA, A. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Berkeley, CA, USA, 2015), USENIX ATC '15, USENIX Association, pp. 277–289.

[19] MAO, Y., MORRIS, R., AND KAASHOEK, F. Optimizing MapReduce for multicore architectures. Tech. rep., 2010.

[20] MCCALPIN, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[21] MERKEL, A., STOESS, J., AND BELLOSA, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 153–166.

[22] MOLKA, D., HACKENBERG, D., SCHÖNE, R., AND NAGEL, W. E. Cache coherence protocol and memory performance of the intel haswell-ep architecture. In *Parallel Processing (ICPP), 2015 44th International Conference on* (2015), IEEE, pp. 739–748.

[23] RADOJKOVIC, P., CARPENTER, P. M., MORETÓ, M., CAKAREVIC, V., VERDÚ, J., PAJUELO, A., CAZORLA, F. J., NEMIROVSKY, M., AND VALERO, M. Thread assignment in multicore/multithreaded processors: A statistical approach. *IEEE Trans. Computers 65*, 1 (2016), 256–269.

[24] ROUSSEEUW, P. J. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics 20* (1987), 53–65.

[25] SNAVELY, A., AND TULLSEN, D. M. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), ASPLOS IX, ACM, pp. 234–244.

[26] SRIDHARAN, S., GUPTA, G., AND SOHI, G. S. Adaptive, efficient, parallel execution of parallel programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 169–180.

[27] TAM, D., AZIMI, R., AND STUMM, M. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 47–58.

[28] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[29] TRANSACTION PROCESSING PERFORMANCE COUNCIL. TPC Benchmark H. http://www.tpc.org/tpch/, 2014.

[30] YADWADKAR, N. J., HARIHARAN, B., GONZALEZ, J. E., SMITH, B., AND KATZ, R. H. Selecting the *best* VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017* (2017), pp. 452–465.

[31] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 607–618.

[32] ZELLWEGER, G., LIN, D., AND ROSCOE, T. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2016), APSys '16, ACM, pp. 14:1–14:9.

[33] ZHANG, Y., LAURENZANO, M. A., MARS, J., AND TANG, L. Smite: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE Computer Society, pp. 406–418.

[34] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 129–142.