A General Model for Placement of Workloads on Multicore NUMA Systems.

Paper **#196**

12 pages, excluding bibliography

The problem of mapping threads, or virtual cores, to physical cores on multicore systems has been studied for over a decade. Despite this effort, there is still no method that will help us decide in real time and for arbitrary workloads the relative impact of different mappings on performance. Prior work has made large strides in this field, but these solutions addressed a limited set of concerns (e.g., only shared caches and memory controllers, or only asymmetric interconnects), assuming hardware with specific properties and leaving us unable to generalize the model to other systems.

Our contribution is an abstract machine model that enables us to automatically build a performance prediction model for any machine with a hierarchy of shared resources. In the process of developing the methodology for building predictive models we discovered pitfalls of using hardware performance counters, a *de facto* technique embraced by the community for decades. Our new methodology does not rely on hardware counters at the expense of trying a handful of additional workload mappings (out of many possible) at runtime.

Using this methodology data center operators can decide on the smallest number of NUMA (CPU+memory) nodes to use for the target application or service (which we assume to be encapsulated into a virtual container so as to match the reality of the modern cloud systems such as AWS), while still meeting performance goals. More broadly, the methodology empowers them to efficiently "pack" virtual containers on the physical hardware in a data center.

1. Introduction

Data center operators balance two objectives: providing a satisfactory experience for the customer and efficiently allocating the hardware that runs customer virtual instances or operator services. We believe that hardware provisioning should not be a guessing game. No one should have to allocate more hardware than needed just out of the fear that providing less could violate a service-level objective.

Unfortunately, mapping threads, or virtual cores, to physical cores on multicore NUMA systems is still like playing a guessing game. Modern **N**on-**U**niform **M**emory **A**ccess systems consist of several nodes that share various resources. Cores within a node may share SMT pipelines and caches. Nodes themselves share an interconnect, which may be asymmetric, providing higher bandwidth for some links than for others and for some directions of communication than for others. See Figure 2 for examples.

The challenge of placing workloads on a NUMA system has to do with very complex effects taking place when the workload's threads access shared resources. Consider Figure 1(a), which shows performance of a WiredTiger key-value store[1] on a NUMA Intel machine. This workload, which runs a BTree search using 24 threads, runs almost twice as fast when all of the threads are placed on a single node, as opposed to being spread across two or more nodes. On Figure 1(b), on the other hand, we see that the same workload configured with 16 threads on an AMD NUMA system runs much faster when it has four nodes at its disposal, rather than two[2]. On the Intel system, this application prefers having all of its threads co-located on a single NUMA node and using SMT because it enjoys lower communication latencies this way. On the AMD machine, on the other hand, it suffers from contention for shared resources when all of its threads are crowded on a small number of nodes.

These observations are not new and they have been studied in the research community for more than a decade [8–10, 12, 13, 15–20, 23, 26, 30–33]. Yet, as we elaborate in §2, we still do not have a method for deciding at runtime how a particular placement – a mapping of virtual CPUs to physical

[1] WiredTiger is used as the backing store for MongoDB [3].

[2] With eight cores/node we could not fit all threads on a single node without creating contention for CPU cycles.

cores[3] – will affect performance of an unknown workload, let alone a model for reasoning about workload placement on a variety of different machines.

The contributions of our work can be summarized as follows: ***We propose a general methodology for reasoning about workload placement on machines with shared resources***. Our methodology is not tied to a particular machine, specific shared resources or certain hardware counters. We abstract a multicore machine as a collection of shared resources (scheduling concerns), where each concern produces a *score* indicating how many threads are using the resource in a given placement. Vectors of scores uniquely identify placements that differ with respect to how threads share the resources. This abstraction enables us to dramatically reduce the number of distinct placements to dozens, from billions of all possible, which makes the problem of making performance predictions tractable. It also enables reasoning about a variety of hardware with shared resources: a previously unmodeled shared resource can be added to the model as a new scheduling concern and the model is updated automatically. *We do not have to manually redesign the model for every new machine.*

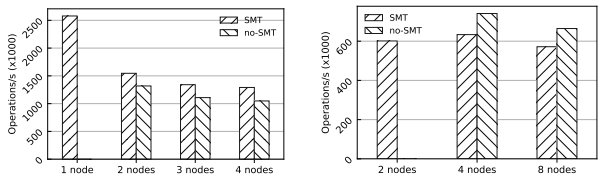
Based on the abstract machine representation, ***we develop a methodology for automatically building a model that predicts performance of a given workload in any distinct placement on a multicore machine*** relative to performance in a baseline placement observed at runtime. With the resulting model users can make decisions such as "give application X as few NUMA nodes as possible while making sure that its throughput remains above Y operations per second".

We show that a widely accepted methodology of using hardware counters for building performance models is not as robust as previously thought. We develop an alternative model building strategy that requires no information from hardware counters for performance predictions[4]. For robust predictions we require trying at most three different placements at runtime (and often fewer that that), out of many possible. The placements are tried without interrupting the workload. With this methodology, performance for most workloads is predicted within 2% of the actual.

Could this problem be solved using a simpler shortcut? Could we conservatively allocate ample number of NUMA nodes for "important" workloads that are subject to service-level objectives? The problem is that if we did that we would end up wasting hardware. For example, the workload in Figure 1(b) runs just as well, even slightly better, on four nodes as it does on eight, so by allocating eight nodes to every virtual instance running this workload, we would use twice as much hardware without any performance benefit.

[3] In §2 we explain why we frame the problem as mapping virtual CPUs to physical cores, as opposed to mapping threads.

[4] Cycle counter would still be used to measure performance and the instruction counter would be used if we use instructions per cycle as the performance measure.



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

Figure 1. Throughput of the WiredTiger key-value store on two NUMA systems according to the number of nodes. On 1-node and 2-node placements we must use SMT to avoid placing more than one virtual CPU per hardware context.

How about avoiding large NUMA systems altogether? Many applications are implemented as a collection of processes communicating over a network, as opposed to a collection of threads communicating over shared memory, so instead of relying on large NUMA systems, we could use a larger number of small NUMA systems to run these applications. Actually, using a smaller number of larger systems can be 25-33% more power efficient relative to using twice as many systems of half the size (based on our own experiments, vendor specifications [2], and previous studies [6]). Similarly, larger densely packed systems would use less space than and require less cooling than a larger number of smaller hosts[5] All in all, using large NUMA systems can deliver economies of scale, but to use these systems effectively we need better tools.

The rest of the paper is organized as follows. §2 compares our work with prior research and states our assumptions about the operating environment. §3 describes the abstract hardware model and §4 presents the model building methodology and evaluates it on two different machines. §5 provides a few example use cases for the model with database and data analytics applications. §6 summarizes the work.

2. Background and Related Work

2.1 State of the Art

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 [15, 26], or shared caches and memory controllers [14, 23, 30, 32]. Later work extended the techniques to multithreaded workloads and to additional resource combinations, such as SMT and shared caches [31], memory controllers and the shared interconnect [12, 20]. 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. [32] showed us how to avoid interference for shared memory controllers and the work of Lepers et al. [20] showed us how to place applications on machines

[5] Our argument does not take into account reliability trade-offs.

with asymmetric interconnects, we still do not know how to build a scheduler that combines both concerns.

Techniques used in prior work did not allow for automatic combination of several models. Every contention model elaborates manual design: careful selection of hardware counters [12, 14, 18, 20, 32] or even manual coding or artificial 'probe' workloads or 'Rulers' [30, 31] that must be run side-by-side with the target workloads to determine their sensitivity to contention.

A very recent system Pandia [17] made significant advances in this problem domain. Pandia accurately predicts relative performance of different workload placements on multicore NUMA machines. It can also predict relative performance of an application with different numbers of threads, but such predictions in Pandia require 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 and producing remarkably accurate performance predictions, 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, for example one with an asymmetric interconnect, 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.

2.2 Assumptions and Limitations

A workload is encapsulated in a virtual container. This assumption sits well with many data centers that use virtualization for a variety of reasons. In our case, it makes the problem easier to solve than if we viewed a workload as an amorphous collection of threads. With a thread-centric view, the degree of concurrency can unpredictably change, either because of application logic or because of OS scheduling decisions. When it changes, performance can be affected because of intra-application scaling issues or because of the change in pressure on shared resources. Combining both effects in a single model if cumbersome: we concluded that for robustly accurate predictions that do not require offline runs we need to train a separate model for each feasible number of vCPUs in a container. Managed 'cloud' environments present their offerings as a menu of virtual instances with a fixed number of virtual cores per instance. For example, AWS offers a dozen instances with the number of different core counts limited to ten [1]. So we can feasibly train a separate model for each machine and each virtual core count. We are *not* interested in finding the optimal number of threads or virtual CPUs for the workload; for that, users can leverage other tools[17, 27].

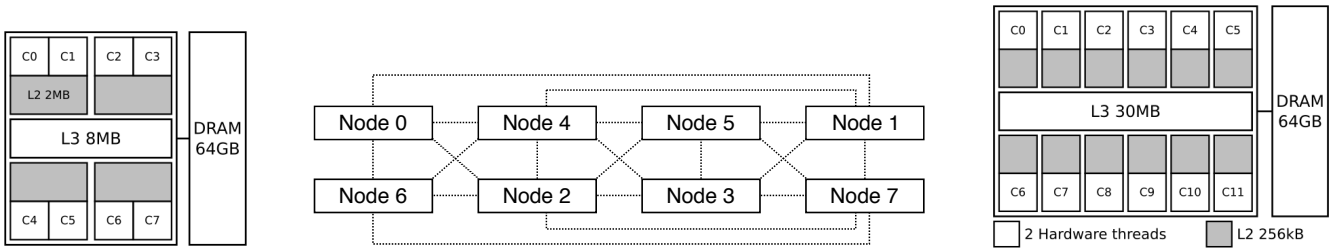
A NUMA node is a unit of resource allocation. We assume that CPU cores are allocated to containers in units of entire NUMA nodes. That is, given a virtual container the goal is to decide across how many NUMA nodes to spread the container; we don't co-locate different containers on the same node. The implication is that a particular placement may leave unused physical cores. For example, if we decide to spread a container with 16 vCPUs across four NUMA nodes on a machine with eight cores per node, four cores on each node will be left unused. Back-filling unused cores with virtual CPUs from other containers is not necessary on modern hardware, cores are *not* the scarce and expensive physical resource: memory is. We rely on this observation to simplify the modelling problem. If we had to model contention among different containers on the same NUMA node, the problem would be a lot more difficult. On the other hand, if multiple containers on the same machine are mapped to different NUMA nodes, they can co-exist without interference if the nodes used for different containers do not share interconnect links; a placement not difficult to effect. We also confirmed this property experimentally.

We consider only balanced placements. A balanced placement is one where the number of virtual cores 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 virtual CPUs sharing each L3 cache is equal. Uneven sharing can cause unpredictable performance effects on the workload, for example by creating stragglers, so we choose to not model these effects. Since we are already assuming that resources will be allocated to containers in multiples of NUMA nodes and that leaving physical cores unused is not detrimental, the balance assumption is not very limiting.

3. Abstract machine model

A major obstacle to a solution to the virtual core 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, and having a way to quantify this would allow us to eliminate the vast majority of placements by only considering those that are actually relevant with respect to performance.



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

Figure 2. We used two main systems for running experiments and evaluating our methods. The first is a quad AMD Opteron 6272 system with 512GB of RAM. It has eight NUMA nodes connected with an asymmetric interconnect (see Figure 2(b)) and 64 cores. Pairs of cores share the instruction front-end, L2 cache, and floating point units. Figure 2(a) shows memory specification of a single node. The second system is a quad Intel Xeon E7-4830 v3 with four NUMA nodes and 96 hardware threads (12 physical cores per node with SMT). Figure 2(c) shows details about each of its nodes.

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 thread placements. The primary purpose of a scheduling concern is to provide a numerical score when given a thread placement as input. The score represents the utilization of the particular resource and it only depends on the vCPU placement, not the dynamic behavior of a workload. A simple example is an "L2 cache" resource. The scheduling concern for the L2 cache measures the utilization of L2 caches on a system where there are multiple L2 caches shared by sets of cores, so the score would simply be the number of L2 caches in-use by vCPUs. The score remains constant with respect to the symmetry of the hardware resource the concern encompasses. So, two placements might use completely different NUMA nodes but if they use the same number of L2 caches then they will both have the same L2 cache score. A vector of numeric scores for all scheduling concerns uniquely identifies each placement that is distinct with respect to sharing of resources.

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 everything 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. In total, the scheduling concerns for a system need to cover all shared resources relevant to thread placement.

Our AMD test system (shown in Figure 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 aggregated bandwidth with a benchmark for each possible combination of nodes (we use *stream* for our measurements). Each concern is relatively simple, easy to implement, and can be developed independently. Since it does not require a performance expert, we envision the specification of scheduling concerns being provided as part of system BIOS.

It is easy to see how scheduling concerns could be developed for other hardware resources. For example, a system with asymmetric CPUs could have a CPU concern where the score is the frequency of the CPUs in use, or a concern that accounts for some nodes being closer to I/O links, where the score is 1 if the nodes in use are near I/O links and 0 otherwise.

Next, from the concerns and hardware topology we need to derive the *important placements*. An important placement must have three properties: **(1)** conform to our balanced assumption, **(2)** be feasible: i.e., not assign more than one

Component	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 components used on our AMD test system (shown in Figure 2).

virtual CPU to a single hardware thread, and **(3)** not be superseded by a strictly better placement.

Given a score s and the number of virtual CPUs 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: e.g. there are eight hardware threads per L3 cache on our AMD test system.

We now explain how we derive all important placements, providing an intuitive explanation of the algorithm in favour of a formal specification. As we explained earlier, we train the model and derive important placements for each expected number of virtual CPUs. In the following text, we use 16 vCPUs as an example.

Starting from the scheduling concern at the bottom of the hierarchy (L2/SMT in our example), we construct a combinatorial tree of all possible combinations of scores, so that at the bottom of the tree we will have our score vectors. To record only the scores that are useful, we enumerate scores only for placements meeting the feasibility and balance properties defined above. For example, given 16 virtual cores, the only possible scores for the L2/SMT concern are 8 and 16, because we can either put two virtual CPUs per L2 cache, giving us a score of 8 (eight total L2 caches used), or one vCPU per L2 cache giving us a score of 16. Feasibility constraints prevents us from putting more than two vCPUs per L2 cache. As we can see, by following these constraints we aggressively trim the combinatorial tree as we construct it, so there is no state explosion.

We continue with the combinatorial tree until we have traversed the entire machine hierarchy. In our example, we end up with fewer than 1000 leaves at the bottom of the tree: a computationally feasible outcome. Next, we trim the leafset to make it conform to the third property of important placements: a placement must not be superseded by a strictly better one. The rules for that step are as follows.

For concerns that affect cost or have potentially an inverse relationship with performance (L2/SMT and L3, in our example), we need to keep all unique score vectors. For concerns that don't affect cost and where performance can never decrease with the score (only the interconnect in our example) we need to keep placements that maximize the score for this concern, all other scores being equal.

Another rule for deciding which score vectors to keep looks at global properties for packing the machine. For example, from the previous step we know we need to keep the 4-node placement that 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 best combination (i.e. highest average interconnect score) of four-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.

All remaining score vectors that have not been kept are discarded. In our example this amounts to discarding placements that have poor interconnect scores and are not useful for packing the machine. Using these rules we eliminate most of the score vectors in the leafset.

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 (Figure 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.

4. Performance Predictions

Automatic model-building techniques use algorithms that learn how to map a set of features describing a data element to a predicted outcome. They rely on a sample representing the population, called a training set. 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: $[1.0, 0.8, 0.7]$. The data elements in our case are executions of workloads in different placements, and the features are some metrics describing the execution, typically hardware counters.

There are two ways to frame the problem. We could either train the model to predict a performance value for each specific placement. In that case, in addition to execution features, we need to provide the scores identifying the placement as the input to the model. The other alternative is to predict a *performance category*. That is, assuming that our data elements can be categorized according to their performance vectors (benchmarks in the same category have roughly similar performance vectors), we can train the model to predict the category and then use the category's average vector as the predicted outcome. Although we tried both methods, the latter has certain advantages, as we explain below.

In the rest of this section we describe how we categorize the benchmarks according to their performance behaviour (§4.1), present two modelling techniques: a classical machine learning (ML) approach that uses hardware counters as feature vectors (§4.2), and an *iterative* technique where we forego using hardware counters and determine a category of the benchmark at runtime by trying several different placements during the first seconds of the execution (§4.3). We also discuss an approach that is a combination of these two methods and evaluate the accuracy of all techniques (§4.4).

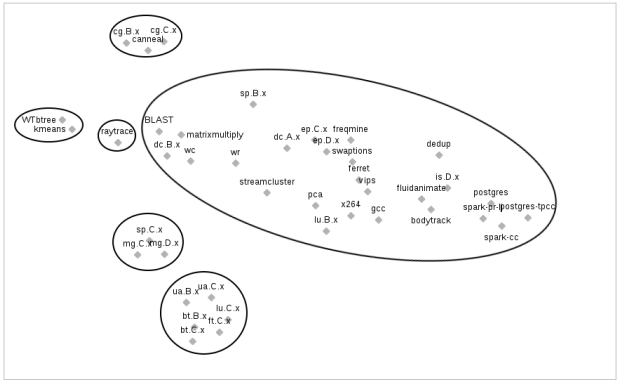


Figure 3. Performance clusters for workloads on the Intel system.

4.1 Categorizing performance behaviour

Although there is an infinite number of possible workloads and a complex interaction between scheduling concerns, we observe that workloads fall into a relatively small number of categories where each category is representative of a specific relationship to scheduling concerns. For example, workloads that are not memory intensive and are not adversely affected by sharing SMT contexts would belong to the same category (the category where thread placement does not matter on our test systems). Another category would be one where using fewer NUMA nodes and fewer physical cores greatly hurts performance, and so on.

To discover these categories, we use the clustering algorithm k-means, which uses performance vectors as the elements. It partitions the elements into k clusters so as to minimize the within-cluster sum of squares (i.e., Euclidean distance) between the vectors. We use the Silhouette coefficient [25] as the internal clustering evaluation metric to determine the optimal number of clusters and judge how well the data can be clustered.

The set of applications we experimented with are drawn from the NAS Parallel Benchmark suite [5], Parsec suite [7], the Metis map-reduce benchmarks [22], and BLAST [4]. 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, roughly one thread per core.

Figure 3 shows the clusters for our Intel test system, overlaid on a t-SNE [21] plot to aid visualization[6]. The workloads were clustered into seven groups. In order to be confident that the method actually results in useful categories, we manually inspected the results. Figure 4 shows the performance by placement for the workloads found in three example clusters. On average the performance of a placement among the workloads varies by about 15%. Considering that the range of performance from best to worst placement is around 50% and the fact that each workload in the group has the exact same trend for placement preferences, it is clear that clustering has worked well in this case. An inspection of the remaining clusters and the clusters produced on our AMD test system show similar results, with some clusters creating a bit tighter groupings and some a bit worse.

This method has two main benefits. First, it helps for determining the quality and completeness of the training workloads. If k-means cannot create good clusters, then the training set could be incomplete. In the case of our training workloads, the quality is good overall but the clusters do show a few weaknesses that could be improved. For example, the cluster containing *kmeans* and *WTbtree* contains only two elements and could benefit from adding other workloads.

The second benefit is that it provides a baseline for what can be considered "ideal" for predicting the performance of workload placements. If one had an oracle that returned the category that a workload belongs to, the resultant prediction

[6] T-SNE is a technique for visualizing multi-dimensional data on a 2D plane. It is not related to the actual clustering algorithm.

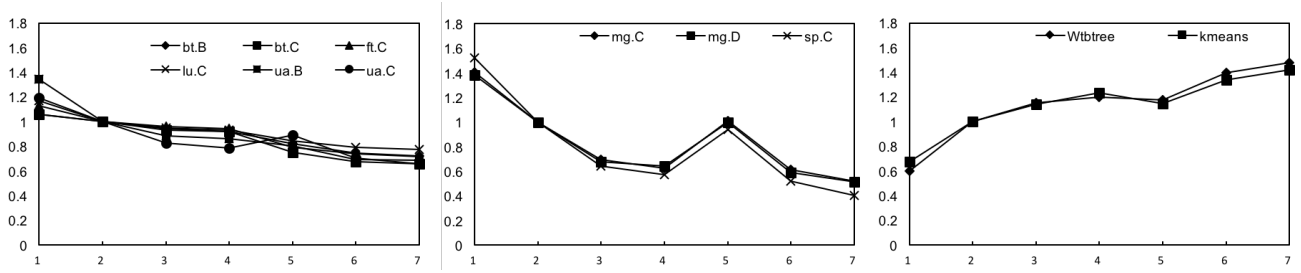


Figure 4. Performance by placement for workloads in three example clusters on Intel. The x-axis shows the placement ID. The y-axis shows performance relative to the baseline placement (#2).

is as good as can be expected without capturing and modeling the complex interactions between scheduling concerns on each type of hardware.

4.2 Predicting Categories with Machine Learning

To train a ML model, we need to decide on a set of features describing our data. A natural choice is hardware performance counters (HWCs); they capture metrics that appear to correlate with performance and are easy to measure online. One concern with HWCs is deciding which ones to use. Modern machines have hundreds of them. Sampling that many online cannot be done without large sampling errors. One approach is to include the list of important hardware counters in the specification of each scheduling concern. This assumes that the engineer providing the specification somehow knows which counters would work best: an assumption we found to be not in the spirit of maximally automating the prediction process. Another approach is to obtain measurements with all possible hardware counters during training and use feature selection methods to identify the best predictors. This approach is automatic, but increases the training time from hours to days.

We used a combination of the two approaches. We started with a large set of counters that seemed as plausible features and then used feature selection to pick the best ones. Feature selection involved taking all counter subsets of size 1 through 5 and automatically selecting the subset that most closely matches the original performance clusters according to the average Silhouette coefficient. For the AMD system, this approach yielded two counters: the L3 cache miss rate and the rate of stalls due to a full load or store queue; using more counters did not improve the accuracy of predictions. On the Intel system, we use the average interconnect traffic, load/store queue stalls, the cache miss rate, and the rate of cache misses serviced by a remote cache.

We train the *Random Forest* (RF) classifier to predict the performance cluster using the selected hardware counters as features. After the cluster is predicted, we use the average of the performance vectors in the cluster as the predicted outcome.

4.3 Iterative Method

The iterative method runs the workload in several placements by trying them during the first few seconds of the execution without interrupting the workload, measures the performance in each, and based on those results uses the process of elimination to determine the category to which it belongs. The downside of the iterative method is that we need to try multiple placements and incur the overhead of migrating threads and memory. The benefit is that we do not have to rely on hardware counters, so there is no burden on model users to select the right ones and no risk of poor predictions if the selected counters did not have predictive power.

The workload must first be run on the baseline placement. Every placement tried thereafter helps narrow down the category to which the workload belongs until only the workload's predicted category remains. If it is determined that the workload does not fall into any category, then predictions are unlikely to be accurate and we default to running it in all of the placements. As with the ML method, once we have determined the category, performance predictions are calculated by averaging performance within that category for each placement not already tried.

Algorithm 1 shows the method in detail. Note that the `while` loop where we try different placements will *not* run on every important placement, because after obtaining each measurement we eliminate clusters from the list of considered candidates (by placing them into the `excluded` list), and if we pick the right order in which to try the placements, we can eliminate many clusters in one go. With this in mind, at each step we choose the placement that will maximize the number of clusters eliminated in the worst-case. In other words, the placement that has the fewest number of clusters that all have overlapping intervals.

4.4 Results

Figures 5 and 6 show the per-workload and per-placement performance predictions for the ML and Iterative methods on the AMD and Intel test systems. The Ideal method generates predictions based on an oracle's knowledge of the workload's correct category (§4.1). The x-axis shows the IDs of the important placements, numbered 1-13 on the AMD sys-

```
Data: Clusters c, Placement ordering p
Result: Predicted cluster for current workload
foreach Cluster in c do
    foreach Placement do
        min = minimum performance within Cluster
          for Placement;
        max = maximum performance within Cluster
          for Placement;
        Interval of c for placement = [min, max];
    end
end
excluded = empty list;
Run baseline placement and measure performance;
while len(excluded) <len(c)-1 do
    Pop next placement from p;
    Run next placement and measure performance;
    foreach Interval for placement do
        if Performance is outside interval then
            Append cluster belonging to interval to
              excluded;
        end
    end
    if Performance is outside all intervals then
        Find closest interval and exclude all other
          clusters
    end
    if All clusters excluded then
        Return no cluster;
    end
end
Return cluster not in excluded;
```

Algorithm 1: Iterative method

tem and 1-7 on the Intel system. The y-axis shows the performance in the placements relative to the baseline. Placement #7 was chosen as the baseline for the AMD system, while the Intel system uses placement #2. For the Iterative method, the number shown in parentheses indicated how many placements had to be tried before the category could be confidently predicted.

The results for both methods 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) or *spark-pr-lj* (a PageRank algorithm run on the LiveJournal database) is included in the training. Similarly, the performance clusters are computed excluding the application that is being predicted. We cross-validated every single workload, but for space constraints we omit the results for the NAS benchmarks and a few Parsec benchmarks. They are qualitatively similar to others and we are happy to provide them upon request.

For both systems the iterative method produces accurate predictions within a couple percent for most workloads, and the number of placements that had to be tried before narrowing down the right category was two or three in most cases. For five of the 41 workloads on the Intel system, a cluster could not be determined, so all seven placements were tried. The AMD system had no workloads where a cluster could not be identified. The accuracy of the iterative method is sub-par only in cases where the training set did not include any workloads that behaved similarly to the predicted benchmark. This was the case for *kmeans* on the AMD system, which was the only benchmark in our training set that preferred SMT. As explained in §4, we can identify weaknesses in the training set using the performance clusters, so it is straightforward to figure out where extra time and effort could be spent on adding new training workloads. Overall, the iterative method always has better prediction accuracy than ML, at the cost of needing extra measurement placements, and its accuracy can be improved in a reliable way.

Prediction accuracy of ML was a lot less reliable. On the AMD system it produced good results overall, but the accuracy was still noticeably worse than that of the iterative method. The real shocker is the results on Intel, where the ML method produced many poor predictions. It completely missed the performance trend for *canneal*, *postgress-tpch*, both Spark workloads, *streamcluster*, *WTbtree*, producing errors of over 40% for several placements.

It is surprising that the ML method produced such high errors despite the careful HWC selection effort. Our results deviate from those in many other studies that used HWCs in similar contexts (§2). One of the reasons could be because our method pursues a much more ambitious goal. Past algorithms generally only targeted one or a couple of scheduling concerns and did not attempt to predict performance; they only tried to give the best placement. HWCs provide enough predictive power for these simpler decisions, but it turns out they are not always sufficient for predicting fine-grained variations in performance when many scheduling concerns are at play. Pandia is the only system known to us that accurately predicted performance on a similar Intel system [17], but it required six offline runs, whereas the ML method relies on HWCs in a single run.

An example of why HWCs could have poor predictive power, and one of the reasons why the Intel system produced worse predictions, is the struggle to predict performance in the single-node placement (Placement #1 in Fig. 6). There is a huge latency difference for inter-thread 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 HWCs. 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 HWCs on a single placement.

4.5 Discussion

We found the iterative method to be more robust in terms of accuracy. It relies only on the assumption that workloads can be assigned to groups according to their behaviour on different placements. The ML method, in contrast, must rely upon the predictive power of the chosen HWCs. The first assumption is easily satisfied in practice and any deviations can be corrected upon training: e.g., by improving the training set if clustering turns out to be weak. In contrast, if the ML method yields poor accuracy because the hardware counters do not have predictive power, it is very difficult to know how to improve the model. Should we spend more effort on selecting the right counters? Are the counters just not designed to measure predictive events? Are they simply buggy?

One option is to combine the iterative method with an ML method. In cases where the ML method is confident in its predictions, the predictions can be used, and no overhead is incurred from trying additional placements. When it is not confident, the iterative method can take over. We implemented a combined method by first renumbering clusters so that first cluster and the last cluster are the most distinct and therefore are the easiest for ML to distinguish. Then we used RF regression to predict the class number. If the first or the last cluster is predicted then we can be confident in the prediction, otherwise the iterative method is used. For the Intel system this allows us to correctly predict six of the workloads using ML and only running the baseline placement.

It is worth emphasizing that any method relying on a training set can fail if it encounters, online, a workload that behaves nothing like any element in training (as we have seen with *kmeans* on the AMD machine). A practical solution must detect low accuracy before we settle on a potentially poor placement. Simply comparing the actual and the predicted performance in the placement recommended by the model is not sufficient. For example, on the Intel system we saw situations where the ML-trained model would recommend a two-node placement as the best-performing one, and the performance prediction in that placement was actually rather accurate, but the model completely failed to foresee that a one-node placement would yield even *better* performance, which would be detrimental for a workload like *WTBtree* on Intel. On the other hand, performance clusters (§4.1) serve as a good base for such a metric. With the iterative method, if we cannot find the precise cluster for the workload, we can infer low confidence and either default to a conservative placement (see §5) or run the workload in all important placements. Similarly, for the ML method, a large variance among the predictions of individual trees in the RF could be used as a low confidence signal.

5. Case Study: Maximizing Machine Usage

To provide an example of how our model could be used in practice, we developed a scheduler that maximizes the number of instances packed per machine, while ensuring that it complies with a target performance.

The scheduler uses the iterative method for performance predictions, so it requires trying several placements in the beginning of the run. The workload is not interrupted as we try different placements, but threads (and possibly memory) must be migrated, so the application may not perform optimally during this time. The amount of overhead incurred depends on how the performance in the attempted placements compares to the one on which we will ultimately settle, and whether switching from one placement to another requires memory migration[7]. We expect the worst-case overhead for an application using 8GB per node to be on the order of a few seconds. With that understanding, the results reported below do *not* include this initial overhead.

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 scheduler is agnostic to the metric used and only requires that the application makes this metric available at runtime. For clarify of presentation, in our experiments we simply set the performance goals to correspond to 90%, 100% and 110% of the performance observed in the baseline placement. All workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 on the Intel system.

We compare our scheduler to the following three scheduling policies: **Conservative**: a naive scheduling policy that allocates the entire machine to each instance, allowing only one instance per machine. **Aggressive**: a naive scheduling 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 virtual CPUs to cores, allowing Linux to perform the mapping in the way it wishes, and possibly creating unneeded contention. As an alternative, we also evaluate a less naive 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. However, this policy requires a careful analysis of the interconnect topology in order to find the correct set of nodes.

We evaluate the schedulers 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). Figures 7 and 8 show the results for three applica-

[7] To minimize memory migration overhead, we rely on the method proposed by Lepers [20], which is roughly 17× faster than the default method on Linux.

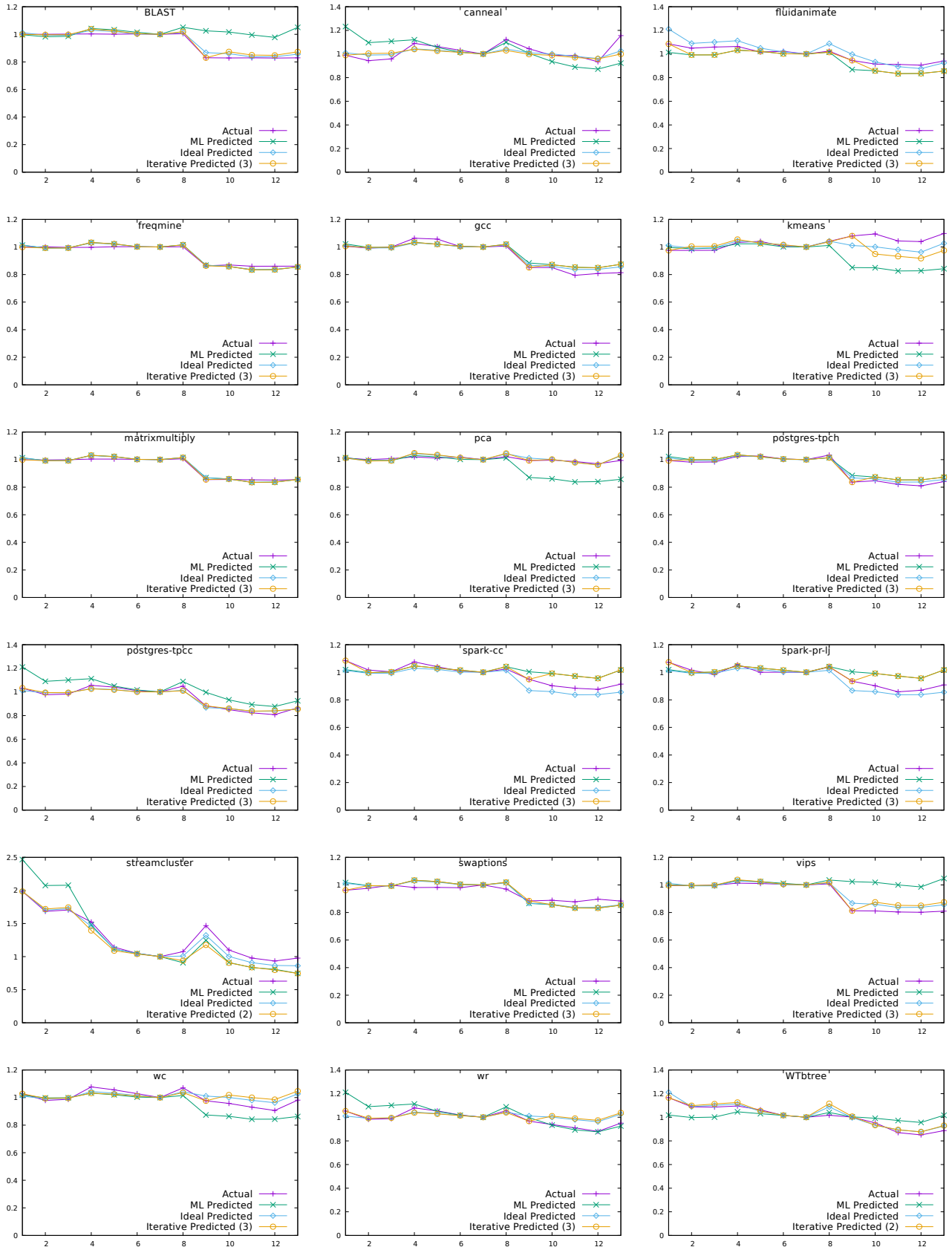


Figure 5. Accuracy of predictions on the AMD system.

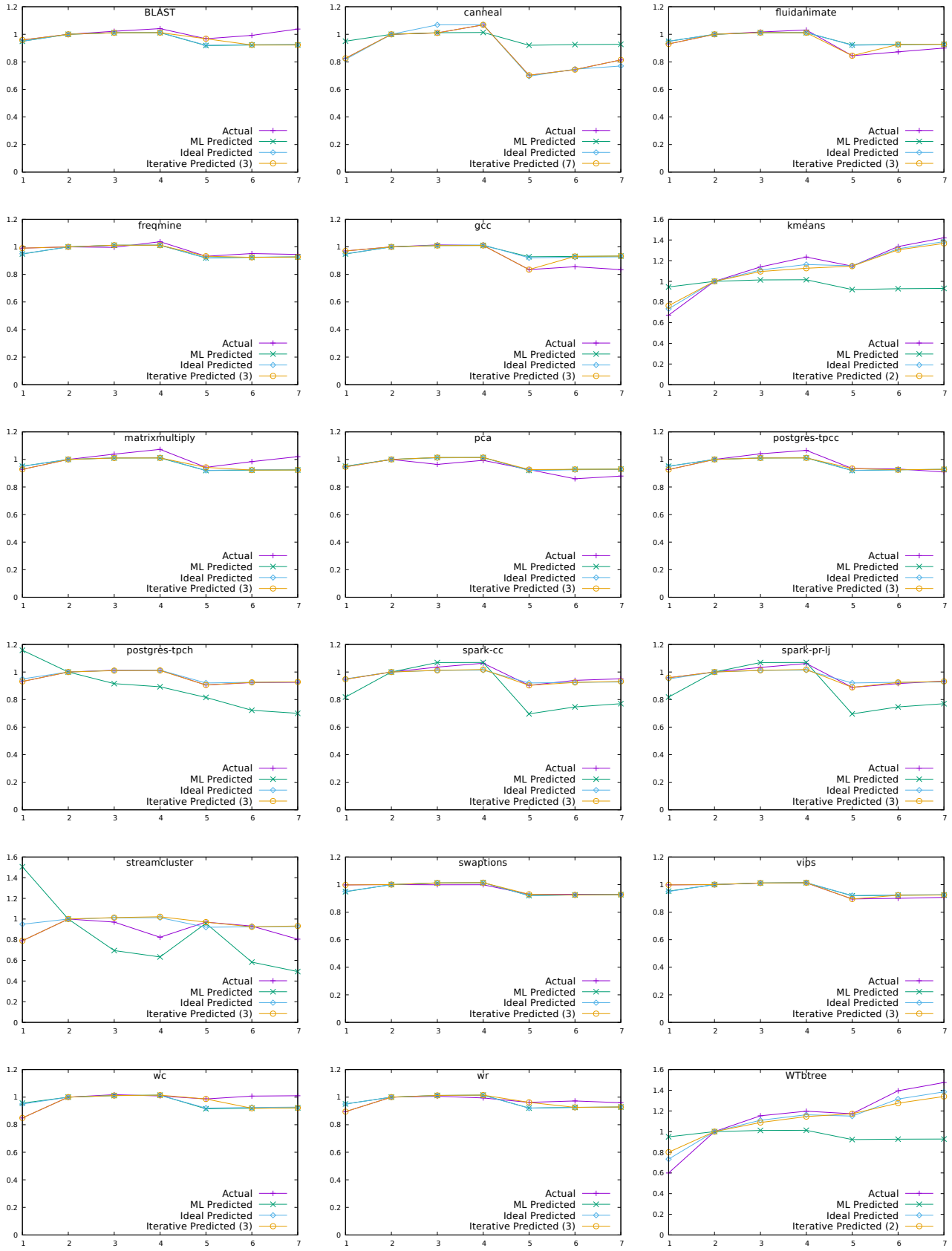
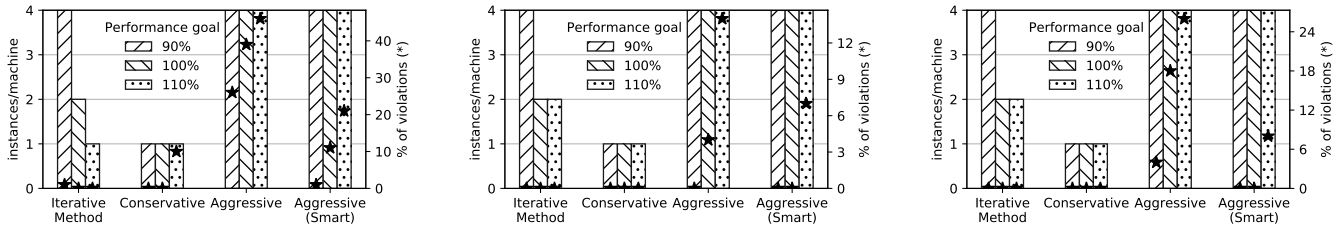
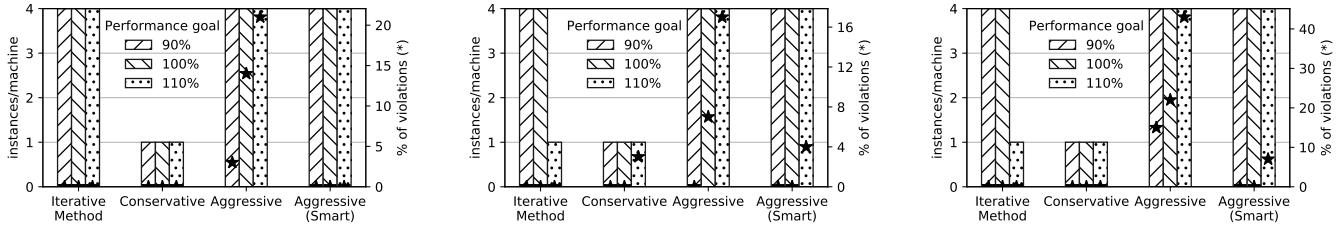


Figure 6. Accuracy of predictions on the Intel system.



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

Figure 7. Instances packed per machine (vertical scale on the left) and percentage of performance goal violations (vertical scale on the right) on the AMD system.



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

Figure 8. Instances packed per machine (vertical scale on the left) and percentage of performance goal violations (vertical scale on the right) on the Intel system.

tions: WiredTiger running a B-tree search benchmark, Postgres running TPC-H, and Spark running PageRank with a Livejournal database on AMD and Intel machines.

The scheduler based on model predictions always meets the performance goal while packing more instances per machine than the conservative scheduler, in most cases. The conservative policy not only wastes resources, but also, surprisingly, may cause performance target violations (Figs. 7(a) and 8(b)), because the Linux scheduler may map vCPUs unevenly to shared resources, causing contention where it could be avoided.

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 number 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.

6. 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. Manual and static mapping solutions are impractical, so operators waste resources and money by using conservative and sub-optimal 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. We presented two methods for predicting performance. One is robust in accuracy at the cost of several workload migrations; the other has less overhead, but can be surprisingly inaccurate, because hardware counters, conventionally used as features for predictive models, turned out to be not as predictive as previously thought. Our best method can lead to very significant advantages in machine utilization while keeping performance guarantees; often $4\times$ higher utilization than static methods that often fail to meet the performance goal.

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 [11] 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 [24], 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. Our methodology is future-proof, general, and provides significant benefits to system operators that want to make reliable performance-utilization tradeoffs.

References

[1] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types.

[2] Dell EMC Enterprise Infrastructure Planning Tool. http://dell-ui-eipt.azurewebsites.net.

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

[4] 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.

[5] 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.

[6] BARIELLE, S. Calculating TCO for energy. *IBM Systems Magazine: Power* (2011), 38–40.

[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] BLAGODUROV, S., AND FEDOROVA, A. In search for contention-descriptive metrics in hpc cluster environment. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2011), ICPE '11, ACM, pp. 457–462.

[9] BLAGODUROV, S., FEDOROVA, A., VINNIK, E., DWYER, T., AND HERMENIER, F. Multi-objective job placement in clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 66:1–66:12.

[10] BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 1–1.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] GAUD, F., LEPERS, B., DECOUCHANT, J., FUNSTON, J., FEDOROVA, A., AND QUÉMA, V. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 231–242.

[17] GOODMAN, D., VARISTEAS, G., AND HARRIS, T. Pandia: comprehensive contention-sensitive thread placement.

[18] 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.

[19] LACHAIZE, R., LEPERS, B., AND QUEMA, V. Memprof: A memory profiler for numa multicore systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 53–64.

[20] 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.

[21] MAATEN, L. V. D., AND HINTON, G. Visualizing data using t-SNE. *Journal of Machine Learning Research 9*, Nov (2008), 2579–2605.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[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] 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.

[31] 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.

[32] 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.

[33] ZHURAVLEV, S., SAEZ, J. C., BLAGODUROV, S., FEDOROVA, A., AND PRIETO, M. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv. 45*, 1 (Dec. 2012), 4:1–4:28.