Tel-Aviv University
Raymond and Beverly Sackler
Faculty of Exact Science
Blavatnik School of Computer Science

Heuristics for Resource Matching in Intel's Compute Farm

This thesis was submitted in partial fulfillment of the requirements for the master degree (M.Sc.) at the Tel-Aviv University

School of Computer Science

Submitted by
Ohad Shai

This work was supervised by
Prof. Sivan Toledo and Prof Dror G. Feitelson

August 2014

Acknowledgments

I would like to express my gratitude to those who without them this research would not have been completed.

I would like to thank my advisor Prof. Dror Feitelson for his guidance of my first steps in the world of job scheduling. His vast experience in that field was of a great benefit for me, His patience and the atmosphere of the work are very much appreciated.

I would like to thank Dr. Edi Shmueli for his part in the research. Without his motivation, partnership and guidance I cannot imagine the research.

In addition, I would like to thank my managers at Intel, Nir Antebi and Moshe Sananes who gave me the appropriate ground and the required assistance for the study at Intel.

I would like to thank Prof. Sivan Toledo for the escort of the research in Tel Aviv University allowing that non-standard setup.

Last, I would like to thank my family. My parents Anat and Eran for their encouragement and support throughout my studies. My loved wife Yael and my children Roni and Tomer for giving me the opportunity along with their encouragement throughout my studies.

Abstract

In this work we investigate the issue of resource matching between jobs and machines in Intel's compute farm. We show that common heuristics such as Best-Fit and Worse-Fit may fail to properly utilize the available resources when applied to either cores or memory in isolation. In an attempt to overcome the problem we propose Mix-Fit, a heuristic which attempts to balance usage between resources. While this indeed usually improves upon the single-resource heuristics, it too fails to be optimal in all cases. As a solution we default to Max-Jobs, a meta-heuristic that employs all the other heuristics as sub-routines, and selects the one which matches the highest number of jobs. Extensive simulations that are based on real workload traces from four different Intel sites demonstrate that Max-Jobs is indeed the most robust heuristic for diverse workloads and system configurations, and provides up to 22% reduction in the average wait time of jobs.

Table of Contents

List of Figures

List of Tables

1 Introduction

Intel owns an Internet-scale distributed compute farm that is used for running its massive chip-simulation workloads [6, 9, p. 78]. The farm is composed of tens of thousands of servers that are located in multiple data centers that are geographically spread around the globe. It is capable of running hundreds of thousands of simulation jobs and tests simultaneously, and handles a rate of thousands of newly incoming jobs every second.

This huge compute capacity is managed by an in-house developed highly-scalable two-tier resource management and scheduling system called NetBatch. At the lower level NetBatch groups the servers into autonomous clusters that are referred to in NetBatch terminology as Physical Pools. Each such pool contains up to thousands of servers and is managed by a single NetBatch entity that is called the Physical Pool Manager or PPM. The role of the PPM is to accept jobs from the upper level, and to schedule them on underlying servers efficiently and with minimal waste.

At the upper level NetBatch deploys a second set of pools that are called Virtual Pools. Just like in the lower level, each virtual pool is managed by a single

NetBatch component that is called the Virtual Pool Manager or VPM. The role of the VPMs is to cooperatively accept jobs from the users and distribute them to the different PPMs in order to spread the load across the farm. Together, these two layers, VPMs at the top and PPMs at the bottom, strive to utilize every compute resource across the farm. This work focuses on the work done at the PPM level.

A basic requirement in NetBatch is the enforcement of fair-share scheduling among the various projects and business units within Intel that share the farm. Fair-share begins at the planning phase where different projects purchase different amounts of servers to be used for their jobs. These purchases eventually reflect their share of the combined resources. Once the shares are calculated, they are propagated to the PPMs where they are physically enforced. The calculation and propagation mechanisms are beyond our scope.

To enforce fair-share the PPM constantly monitors which jobs from which projects are currently running and the amount of resources they use. The PPM then selects from its wait queue the first job from the most eligible project (the project whose ratio of currently used resources to its share of the resources is the smallest) and tries to match a machine to that job. If the matching succeeds, the job is scheduled for execution on that machine. Otherwise, a reservation is made for the job, and the process is repeated while making sure not to violate previously made reservations. Such reservations enable jobs from projects that are lagging behind to obtain the required resources as soon as possible.

Matching machines to jobs is done using any of a set of heuristics. For example, one may sort the list of candidate machines according to some pre-defined criteria — e.g. increasing number of free cores or decreasing amount of free memory — and then traverse the sorted list and select the first machine on which the job fits. This leads to variants of Best-Fit and Worse-Fit schemes. Good sorting criteria reduce fragmentation thus allowing more jobs to be executed, and are critical for the overall utilization of the pool. Alternatively one may opt to reduce overhead and use a First-Fit heuristic.

The sorting criteria are programmable configuration parameters in NetBatch. This allows one to implement various matching heuristics and apply them on different resources to best suit the workload characteristics and needs. NetBatch also allows individual jobs to specify different heuristics, while the pool administrator can set a default policy to be used for all jobs.

In this work we argue that no heuristic applied to a single resource in isolation can yield optimal performance under all scenarios and cases. To demonstrate our point we use both simple test cases and workload traces that were collected at four large Intel sites. Using the traces, we simulate the PPM behavior when applying the different heuristics to schedule the jobs. We show that depending on the workload different heuristics may be capable of scheduling a higher number of jobs.

In an attempt to overcome the problem we develop "Mix-Fit" — a combined heuristic that tries to balance the use of cores and memory. Intuitively this should reduce fragmentation at the pool. However, while generally better than the previous heuristics, Mix-Fit too fails to yield optimal assignments in some cases.

As an alternative, we propose a meta-heuristic we call "Max-Jobs". Max-Jobs is not tailored towards specific workloads or configurations. Instead, it uses the aforementioned heuristics as sub-routines and chooses, in every scheduling cycle, the one that yields the highest number of matched jobs. This overcomes corner cases that hinder specific heuristics from being optimal in all cases, and conforms well to the NetBatch philosophy of maximizing resource utilization in every step. We demonstrate, through simulation, that Max-Jobs yields lower wait times by up to 22% for all jobs in average under high loads.

The rest of this work is organized as follows. Section 2 reviews the traces that were collected from Intel's pools. Section 3 provides more details on the problem of matching machines to jobs, and explores the performance of commonly used heuristics. Section 4 then describes the Mix-Fit heuristic, followed by the Max-Jobs meta-heuristic in Section 5, and simulation results in Section 6. Section 7 presents related work, and Section 8 concludes.

2 Workload Characterization

In order to perform the simulations that will be described later, we conducted a survey of the jobs that were executed in NetBatch during a one month period. The following section describes the traces of jobs that were collected.

NetBatch is a grid computing scheduling system. There exist few similar grid computing systems: LSF [3], OracleGrid [2] and HTCondor [1] are such examples. NetBatch is an in-house solution that was implemented at Intel, and is tailored for Intel's specific needs.

As a grid computing system, NetBatch accepts jobs, schedule to run it on one or more machines and manage all aspects of the job execution environment. When a job finishes it's execution in NetBatch , it is reported to a database with all the information that was gathered during execution. That data is then stored for few month for users introspection. Later, the data is aggregated by business analysis requirements, while the original data is dropped. We collected the data of job execution couple of month after these jobs were finished and before the data was deleted.

The data comes from traces that were collected at the PPM level during a

Property	Pool A	Pool B	Pool C	Pool D
Number of Jobs	13,368,191	13,085,800	13,313,793	9,054,066
Number of Users	1,104	1,615	1,146	862
Number of Machines	1,633	3,115	1,499	2,692
Number of Cores	16,588	33,300	18,816	40,372

Table 2.1: General summary of the traces collected during one month period.

one-month period, and which contain up to 13 million jobs each. The data was collected from four of Intel's largest pools at different locations. These pools were labelled A, B, C and D. The data is stored and available in the Parallel Workload Archive [4] in original format and The Standard Workload Format[7]. Table 2.1 describes general properties of the recorded data.

Figures 2.1 and 2.2 show the distribution of the jobs' cores and memory requirements [1]. As can be seen in the figures, the vast majority of the jobs are serial (single-thread jobs, requiring a single CPU core in order to execute). Memory requirements are mostly 8 GB and below, but there are jobs that require 16 GB, 32 GB, or even more memory (not shown) in order to execute. These observations are consistent across the pools.

[1] The requirements are specified as part of the job profile at submit time.

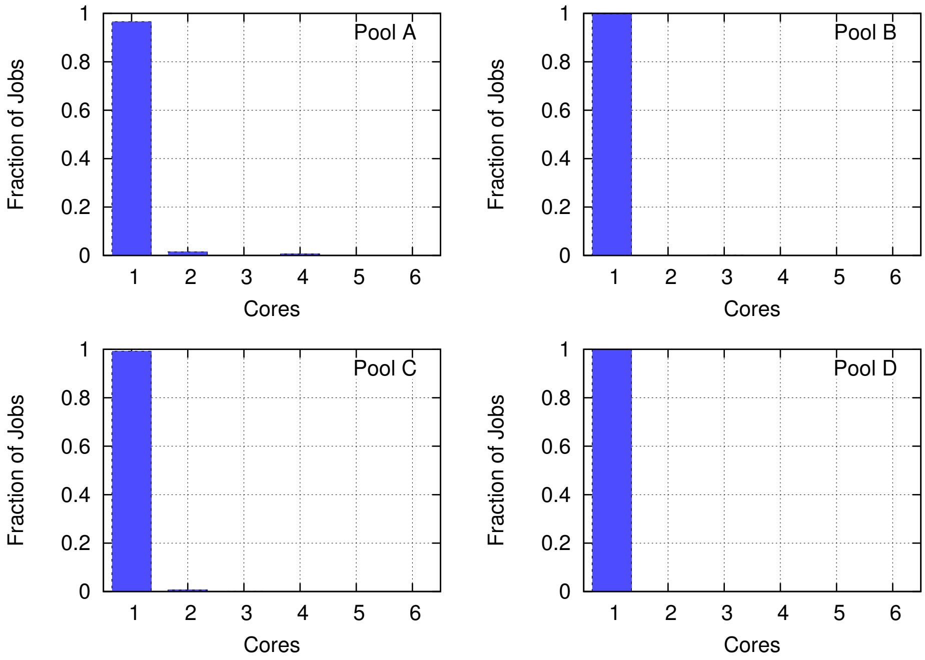


Figure 2.1: Jobs' cores requirements: the vast majority of the jobs are serial and require a single CPU core in order to execute.

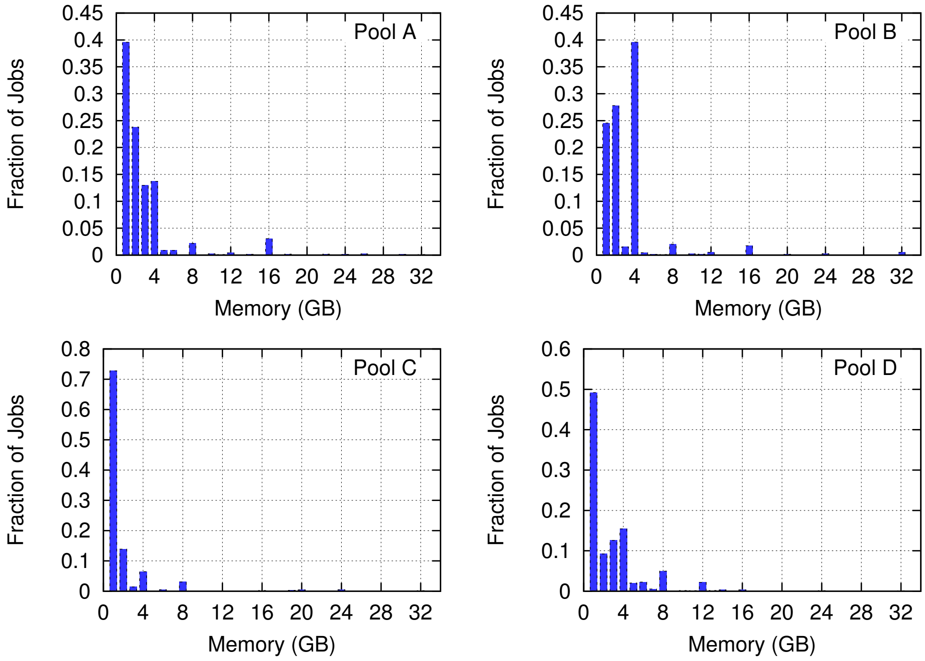


Figure 2.2: Jobs' memory requirements: demands are mostly 8 GB and below, but there are jobs that require 16 GB, 32 GB or even more memory in order to execute.

3 Matching Machines to Jobs

As described above, matching machines to jobs at the PPM is done by choosing the most eligible job from the wait queue, sorting the list of candidate machines according to some pre-defined criterion, traversing the sorted list, and selecting the first machine on which the job fits[1]. This is repeated again and again until either the wait queue or the list of machines are exhausted. At this point the PPM launches the chosen job(s) on the selected machine(s) and waits for the next scheduling cycle.

A job may be multi-threaded, but we assume that each job can fit on a single (multi-core) machine. In principle NetBatch also supports parallel jobs (called "MPI jobs") that span multiple machines, but in practice their numbers at the present time are small. The only added difficulty in supporting such jobs is the need to allocate multiple machines at once instead of one at a time.

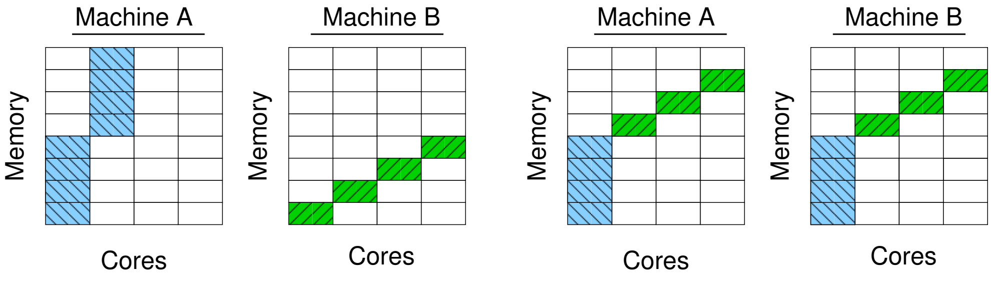
There are many criteria by which the machines can be sorted. In this work we focus on the number of free cores and amount of free memory, as this suits well the workload in Intel which is characterized by compute-intensive memory-

[1] This is done for practical reasons since trying all combinations is time consuming.

demanding jobs. Though I/O is definitely a factor, and some jobs do perform large file operations, there are some in-house solutions that are beyond the scope of this work that greatly reduce the I/O burden on the machines.

The two ways to sort the machines by available cores or memory are in increasing or decreasing order. Sorting them by *increasing* amount of free cores or memory and selecting the first machine on which the job fits effectively implements the Best-Fit heuristic. Best-Fit is known to result in a better packing of jobs, while maintaining unbalanced cores (or memory) usage across the machines in anticipation for future jobs with high resource requirements. Sorting the machines by *decreasing* amount of free cores or memory implements the Worse-Fit heuristic. Worse-Fit's advantage is in keeping resource usage balanced across machines, which is particularly useful for mostly-homogeneous workloads. For completeness we also mention First-Fit. First-Fit's advantage is in its simplicity, as it does not require the sorting of the machines. Our tests, however, revealed that it performs poorly in our environment, so we do not refer to it further.

We argue that no single heuristic, when applied to a single resource in isolation, can yield optimal performance under all workload scenarios. To demonstrate our point we begin by providing simple synthetic examples showing how different heuristics match different number of jobs under different workload conditions. We then put theory to the test by running simulations on the aforementioned traces, demonstrating the effectiveness of the different heuristics under different workloads.



(a) Best-Fit

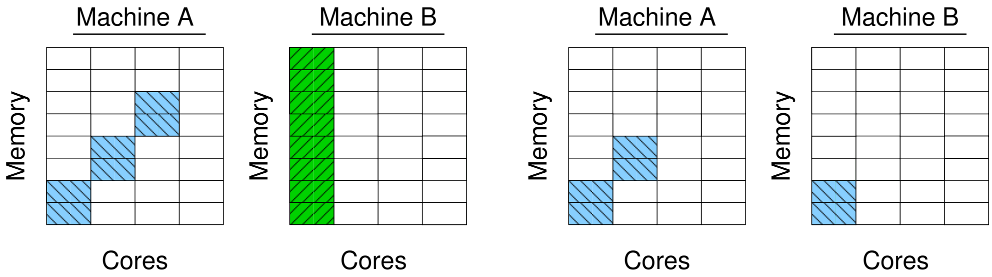
(b) Worse-Fit

Figure 3.1: Scenario for which Worse-Fit (right) is better than Best-Fit (left). Memory is depicted in 4 GB blocks. Shading indicates mapping of a job to a certain core and certain blocks of memory. Note that both cores and memory are mapped exclusively to distinct jobs.

3.1 Synthetic Examples of Heuristics Failures

In our examples we consider two machines, A and B, each having four cores and 32 GB of memory. Assume that 8 jobs are queued at the PPM in the following priority order: two jobs of one core and 16 GB of memory, and then 6 jobs of one core and 4 GB of memory. As can be seen in Figure 3.1(a), Best-Fit matches the first two jobs with machine A, totally exhausting its memory, and the next four jobs with machine B, thereby exhausting its cores. The end result is two non-utilized cores on machine A, half the memory non-utilized on machine B, and two jobs that remain pending at the PPM. Worse-Fit on the other hand matches the first two jobs on different machines, which leaves enough free space (cores and memory) for all the remaining 6 jobs to be matched. This is illustrated in Figure 3.1(b).

Another example is illustrated in Figure 3.2. The priority order here is 3 jobs of one core and 8 GB, followed by one job of one core and 32 GB of memory. As



(a) Best-Fit

(b) Worse-Fit

Figure 3.2: Scenario for which Best-Fit (left) is better than Worse-Fit (right).

can be seen, Worse-Fit spreads the first three jobs on different machines, which doesn't leaves enough memory for the 32 GB job to be matched. Best-Fit on the other hand matches the first three jobs on machines A, which allows the 32 GB to be matched with machine B.

3.2 Observations from the Workloads

Machines currently available on the market typically have multi-core CPUs and large amounts of memory. Therefore, we may expect to see situations similar to the ones described above. In addition, jobs comes with core and memory requirement, and in most cases jobs are allocated one per core. This may waste cycles due to wait states and I/O, but makes things much more predictable.

To characterize the use of cores and memory in each of the pools, we used the traces mentioned above, and partitioned them into buckets of 1000 jobs each. This resulted in 13K buckets for pools A, B, and C, and 10K buckets for pool D. Such small buckets allow us to observe bursts of activity that deviate from the average.

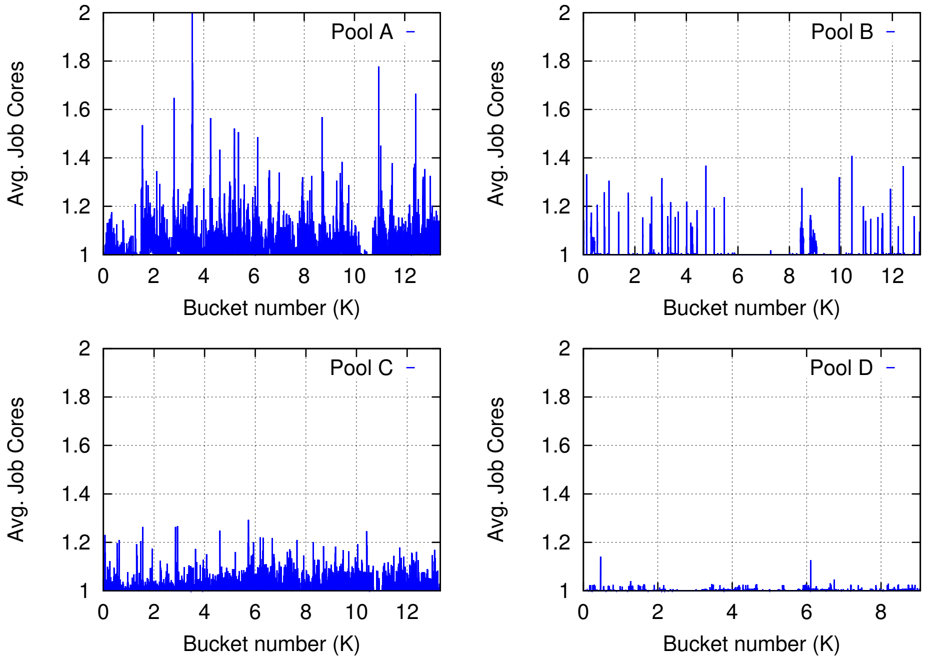


Figure 3.3: Bursts in jobs cores requirements: pool A is the burstiest. Pool B's bursts are sparse, while pool C's have only a small amplitude. In pool D there are virtually no bursts of jobs requiring more than one core.

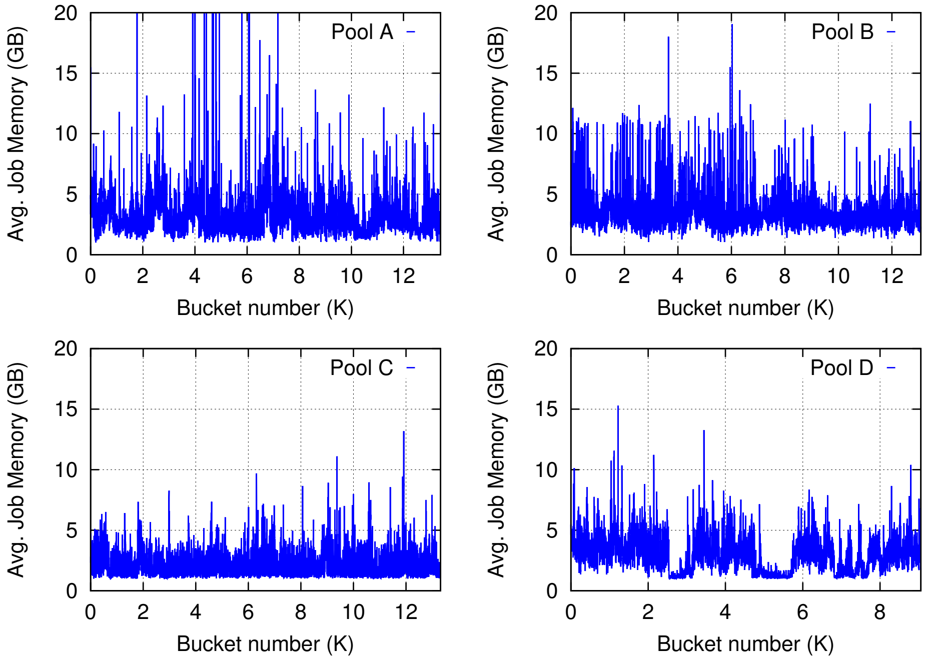


Figure 3.4: Bursts in jobs memory requirements: pools A and B are the most bursty; A in particular has bursts that exceed 20 GB on average. Pool C is somewhat steadier, while pool D exhibits periods of particularly low memory demands between the bursts.

Figures 3.3 and 3.4 show the jobs' average cores and memory requirements in each of the buckets, for each of the four pools, respectively. As can be seen, different pools exhibit different magnitudes of bursts of jobs with high core or memory demands. Pool A is the most bursty in both dimensions; it is the only pool that had a bucket in which the average job core requirement is higher than 2, and multiple buckets in which the average memory requirement is larger than 20 GB.

Pool B exhibits sparse bursts of jobs with high core demands, but intense bursts of high memory requirements. Pool C exhibits continuous moderate core demands, and also relatively steady memory bursts. Finally, pool D has virtually no bursts of jobs requiring more than one core, but it does exhibit bursts of high memory demands, along with periods of particularly low memory requirements.

3.3 Comparing Heuristics

To demonstrate the effectiveness of the different heuristics under different workloads we performed the following experiment. We used the buckets described above, assigned all jobs in each bucket a submit time of 0, and gave each heuristic an opportunity to try and match, in simulation, as many jobs as possible from each bucket on a small synthetic pool of empty machines (total of 512 cores); jobs that could not be matched were simply skipped. For each bucket we then counted the number of jobs matched by each heuristic, and gave the winning heuristic(s) (the one(s) who matched the highest number of jobs) a point.

The results are shown in Figure 3.5. As can be seen, Worse-Fit-Cores signifi-

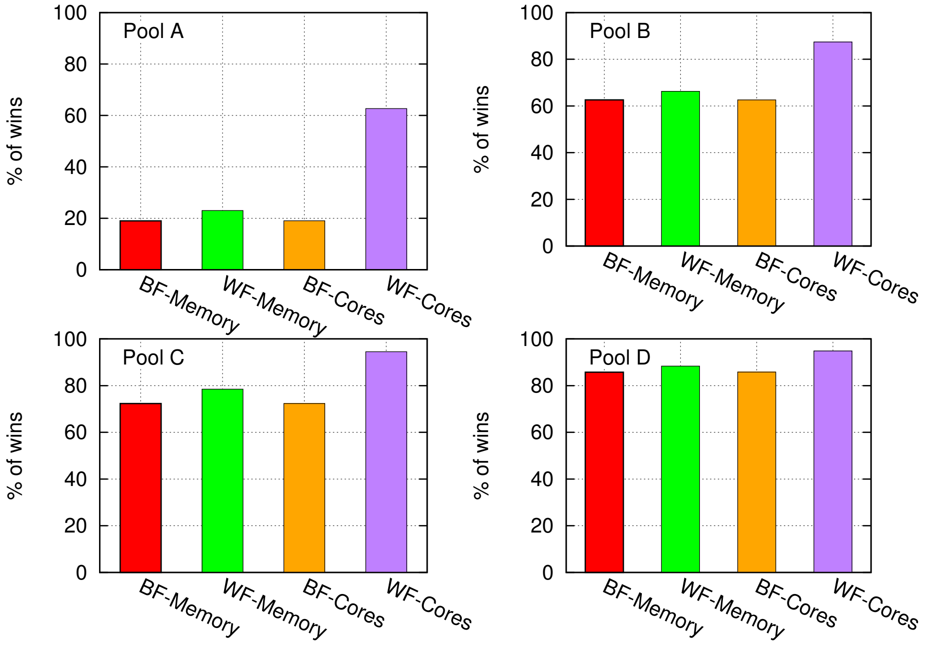


Figure 3.5: Percentage of wins by each heuristic: Worse-Fit-Cores significantly outperforms the other heuristics in pool A. The differences in pools B, C, and D are smaller.

cantly outperforms all other heuristics (collecting the highest percentage of wins) in pool A. It is also the best heuristic in pools B, C, and D, but the differences there are smaller. There is little difference among Best-Fit-Memory, Worse-Fit-Memory, and Best-Fit-Cores, although Worse-Fit-Memory is consistently slightly better than the other two. Notably, for pool D where there is virtually no core fragmentation as indicated in Figure 3.3 there seems to be little difference between the performance of the different heuristics.

An important observation is that though Worse-Fit-Cores appears to be the preferred heuristic, it did *not* win in all cases. This is shown by the gap between the Worse-Fit-Cores bars and the 100% mark, indicating that in 6–37% of the experiments other heuristics performed better. These gaps are the motivation for the Mix-Fit heuristic proposed next.

4 The Mix-Fit Heuristic

As demonstrated in the previous section, none of the one-dimensional heuristics is capable of maximizing the number of matched jobs under all workload scenarios. In this section we propose a new heuristic, Mix-Fit, that takes into account both cores and memory in an attempt to overcome the problem.

4.1 Balanced Resource Usage

The basic idea behind Mix-Fit is to try and reach balanced resource utilization across both cores and memory. This is achieved by considering the *configured* ratio of cores to memory on each machine, and matching the job with the machine on which the ratio of *used* cores to memory, together with this job, is closest to the configured ratio.

To see how this is done, envision a grid representing possible resource combinations (as was done in Figures 3.1 and 3.2). Each column represents a CPU core, and each row a block of memory (the sizes of such blocks are not really important as long as they are used consistently; they should correspond to the smallest unit being allocated). Assuming that cores and memory blocks are assigned exclusively to jobs, an allocation may be portrayed as a sequence of shaded squares on this grid, where each job is represented by a sequence of memory-squares in a specific core-column.

The configured ratio is represented by the diagonal of this grid, and the used ratio by the line connecting the top-right point of the grid with the top-right point of the last job. Mix-Fit defines a parameter, α, that denotes the angle between these two lines. Note that the used ratio is calculated after allocating the job being considered, so machines on which this job does not fit are excluded from the discussion. Mix-Fit then matches the job with the machine with the minimal α value. In case of a tie, the first machine with the minimal value is used.

Two important notes. First, The grid is drawn such that memory and cores are normalized to the same scale in each machine separately, thereby creating a square. This prevents the scale from affecting the angle. Second, the angle is based on lines emanating from the top right corner. It is also possible to have a similar definition based on the origin (i.e. the bottom-left corner). Choosing the top-right corner leads to higher sensitivity when the machine is loaded, which facilitates better precision in balancing the resources in such cases.

Let's see an example. Three machines are available, each with 4 cores and 32 GB of memory. Machine A already has one job with 24 GB, Machine B has 2 jobs with 8 GB each, and Machine C has one job with 2 cores and 4 GB memory and another job with 1 core and 4 GB memory. The next job that arrives requires one core and 8 GB of memory. The various α values of all three machines including the newly arrived job are demonstrated in Figure 4.1. The machine selected by

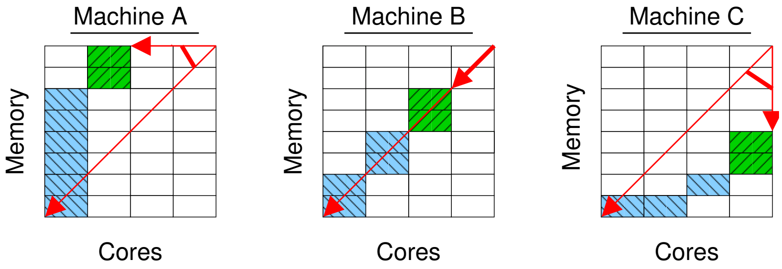
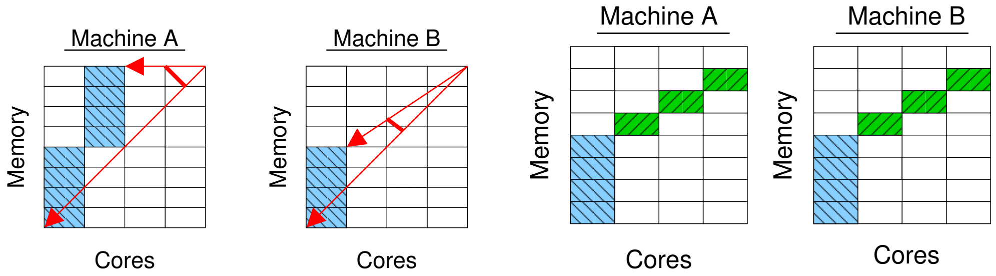


Figure 4.1: Example of various α angles calculated by Mix-Fit. The selected machine in this case is B where $\alpha = 0$.



(a) Options for placing the 2'nd job. The angle on machine B is smaller.

(b) Final allocation by Mix-Fit.

Figure 4.2: Mix-Fit behaviour for the example in Figure 3.1. The end result is identical to Worse-Fit.

Mix-Fit in this case is B where $\alpha = 0$.

To demonstrate why this may be expected to improve over the previous heuristics we will use the same examples we used above. Consider Figure 3.1, where Worse-Fit yielded the best match. After matching the first 16 GB job with machine A, Mix-Fit will match the second 16 GB job with machine B, as this will lead to a smaller α value as can be seen in Figure 4.2(a). It will then match the remaining 4 GB jobs with both machines until all cores get utilized. As can be seen in Figure 4.2(b) the end result is identical to Worse-Fit.

Next, lets re-examine Figure 3.2 where Best-Fit yielded the best results. In this scenario Mix-Fit will match the first three 8 GB jobs with machine A, and

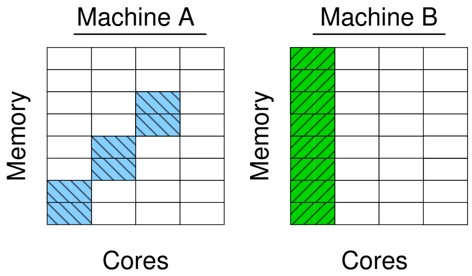


Figure 4.3: Jobs allocated by Mix-Fit for the Example in Figure 3.2. The result is identical to Best-Fit.

then the 32 GB job with machine B, replicating the behaviour of Best-Fit. Note that α would have been the same for the second 8 GB, whether it would have been matched on machine A or B. But as noted above, in such cases the First-Fit heuristics is used as a tie breaker and hence the job is matched with machine A. As can be seen in Figure 4.3 the end result is identical to Best-Fit.

4.2 Mix-Fit's Results

To check the performance of Mix-Fit we repeated the buckets experiment from Section 3.3, but this time including Mix-Fit in the set of competing heuristics. The results are shown in Figure 4.4. As can be seen, Mix-Fit wins by only a small margin in pool A, performs similarly to Worse-Fit-Cores in pools B and D, and is slightly outperformed by Worse-Fit-Cores in pool C.

These results are counter-intuitive, since in a two-dimensional environment of cores and memory, where both resources are subject to sudden deflation by bursts of jobs with high demands, a reasonable strategy would be to try and balance the usage between the resources, in order to secure a safety margin against bursts of

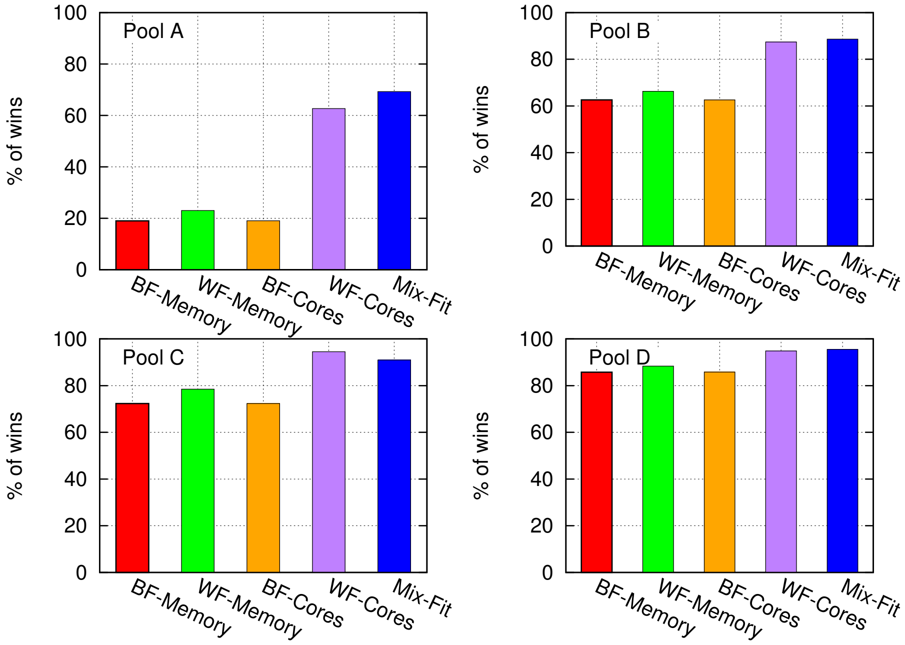


Figure 4.4: Percentage of wins by each heuristic: Mix-Fit wins by only a small margin in pool A, performs similarly to Worse-Fit-Cores in pools B and D, and is slightly outperformed by Worse-Fit-Cores in pool C.

any kind. This strategy, however, which Mix-Fit employs, seems to yield some improvement only under the most bursty situations (pool A). This leads us to default to a meta-heuristic, Max-Jobs, which is described next.

5 The Max-Jobs Meta-Heuristic

The experiment described above indicates that counter-intuitively, Mix-Fit does not yield the best performance in all pools. As an alternative, we therefore suggest the use of the Max-Jobs meta-heuristic.

A meta-heuristic is an algorithm that employs other heuristics as subroutines. In our case, Max-Jobs uses all of the heuristics described before: Best-Fit-Cores, Best-Fit-Memory, Worse-Fit-Cores, Worse-Fit-Memory, and Mix-Fit. At each scheduling cycle, Max-Jobs picks the best schedule produced by any of these heuristics for this cycle. In other words, the meta-algorithm runs all the available heuristics as black-boxes and selects the one with the best result for the currently queued jobs. The target function defining "best" is maximizing the number of jobs assigned to machines in this cycle. Importantly, additional heuristics can be added later and the system will take advantage of them in those cases that they perform the best.

Pseudo-code for the Max-Jobs meta-heuristic is given in Figure 5.1.

```
L – list of heuristics
S – list of proposed schedules (mapping jobs to hosts)

foreach heuristic H in L
        S[H] = H.Schedule(waitingQueue)
maxJobsSchedule = MaxJobsSchedule(S)
Dispatch(maxJobsSchedule)
```

Figure 5.1: The Max-Jobs meta-heuristic.

6 Simulation Results

To experiment with Max-Jobs, Mix-Fit and the rest of the heuristics, we developed a Java-based event-driven simulator [14] that mimics the matching behaviour at the PPM. The simulator accepts as input a jobs trace file, a machines configuration file, and a parameter defining which matching heuristic to apply. It first loads the two files into memory, building an event queue of job arrival events sorted according to the timestamps from the trace (hence preserving the original arrival order and inter-arrival times of the jobs), and a list of machine objects according to the configuration.

The scheduling function is invoked by the scheduler at regular intervals, as is commonly done in many large-scale systems. In our simulations we used an interval of 30 seconds. This allows the scheduling overhead to be amortized over multiple jobs that are handled at once, and may also facilitate better assignments of jobs to machines, because the scheduler can optimize across a large number of jobs rather than treating them individually.

In each scheduling cycle, the scheduler begins by picking the first arrival event from the queue and trying to match a machine to the arriving job using the selected heuristic[1]. If the matching succeeds the job is marked as "running" on the selected machine, and a completion event is scheduled in the event queue at a time-stamp corresponding to the current time plus the job's duration from the trace. Otherwise a reservation is made for the job. Specifically the machine with the highest available memory is reserved for the job for its future execution, thus preventing other jobs from being scheduled to that machine during the rest of the scheduling cycle.

For the workload we used the traces that were described in Section 3, and which contains 9–13 million jobs each. The parameters we used from the traces are the jobs' arrival times, runtime duration, and the number of cores and amount of memory each job requires in order to execute (see Figures 2.1 and 2.2 for the distributions). For the machines we used a special NetBatch command to query the present machine configurations from each of the pools on which the traces were collected.

Our initial simulations revealed that the original load in the traces is too low for the wait queue in the simulated PPM to accumulate a meaningful number of jobs. This may stem from fact that the load in the month in which the traces were collected was particularly low, or that the configuration has changed by the time we ran the machines query (a few months later). In any case the results were that all heuristics performed the same.

To overcome this problem we increased the load by multiplying the jobs arrival time by a factor, β, that is less than or equal to one. The smaller the value of β, the

[1] For simplicity we skipped the fair-share calculation.

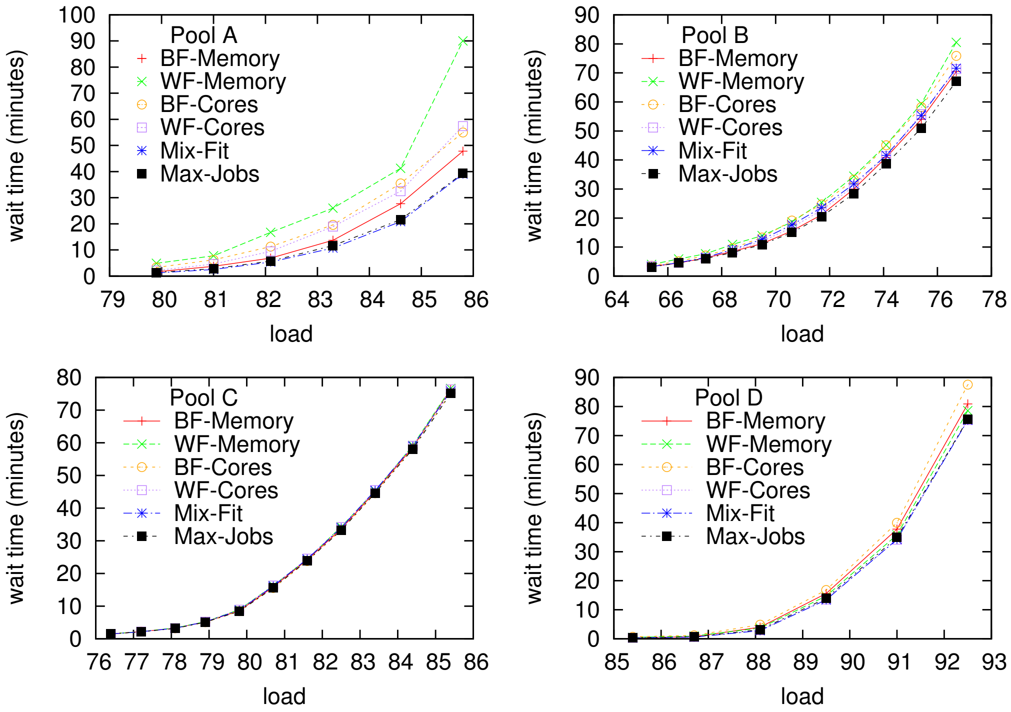


Figure 6.1: Average wait time of jobs. System load is expressed as percent of capacity.

smaller the inter-arrival times become between the jobs, which increases the rate of incoming jobs and the load on the simulated pool. We ran high-load simulations with β values ranging between 0.58–0.95. In the figures below, we translate the β values into an actual load percentage for each pool.

Metrics that were measured are the average wait time of jobs, the average slowdown, and the average length of the waiting queue during the simulation. The results are shown in Figures 6.1 to 6.3, for each metric, respectively. Since the metrics are dependent and the results are similar between the metrics, we will only discuss the differences between the heuristics and pools.

In correlation with the buckets experiment in Figure 4.4, Mix-Fit showed

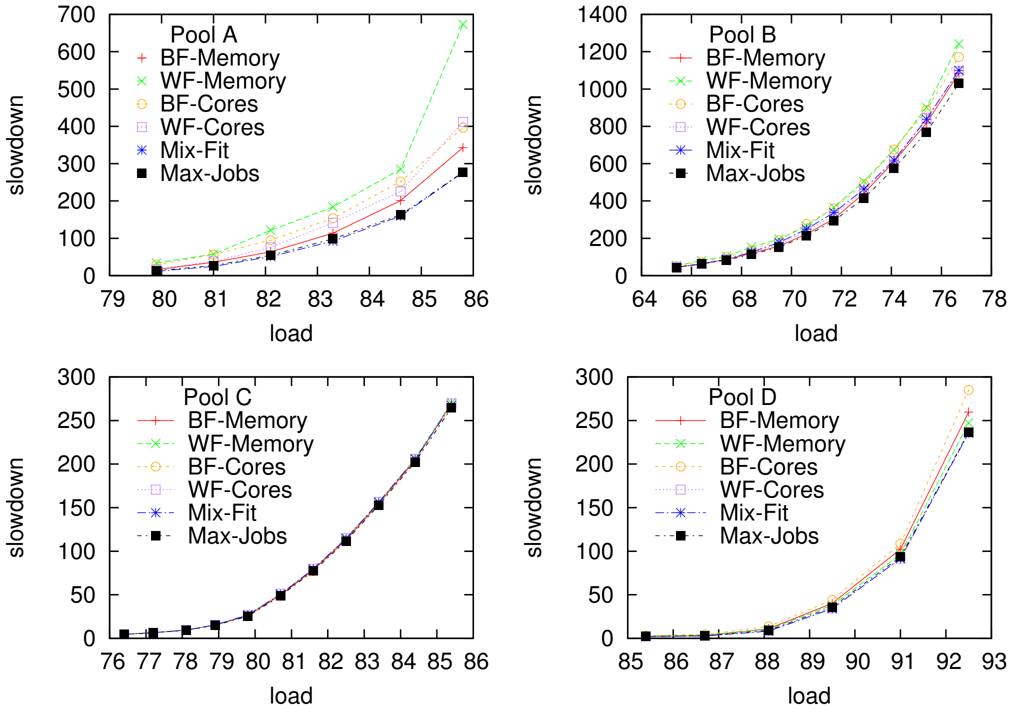


Figure 6.2: Average slowdown of jobs.

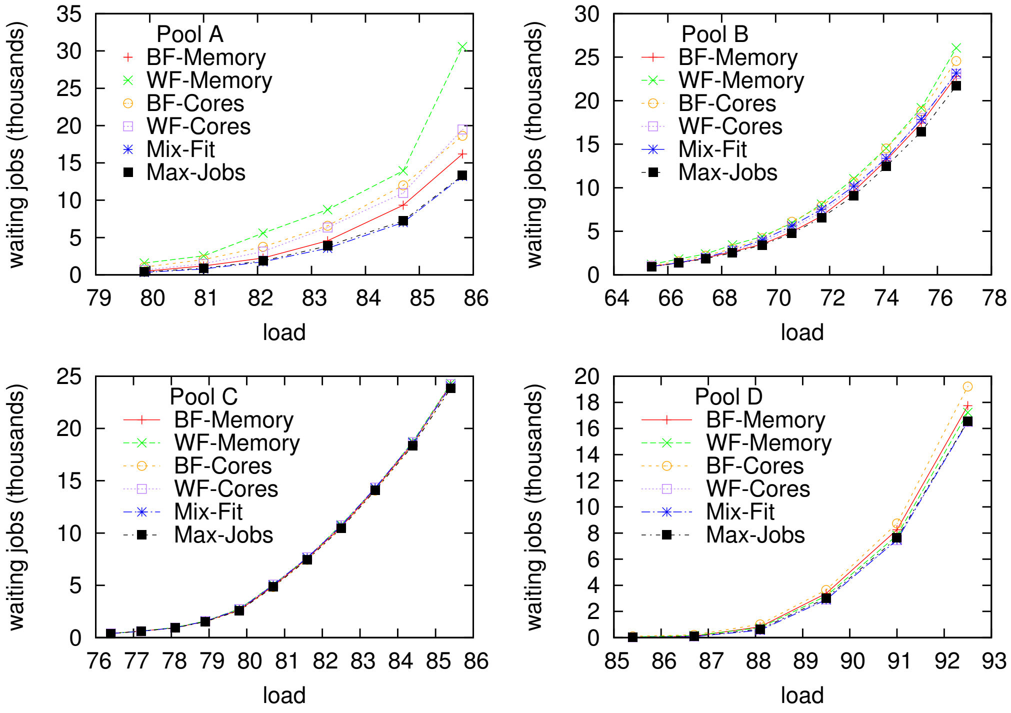


Figure 6.3: Average wait-queue length.

marked improvement over the other heuristics in pool A, and was able to reduce the waiting time by 22%, slowdown by 23%, and queue length by 22% under the highest loads simulated.

The second-best heuristic on pool A, Best-Fit-Memory, appears to slightly outperform Mix-Fit in pool B, especially in the mid-range load, as opposed to the buckets experiment. This may be caused by the fact that pool B had the most intense bursts of high memory demands and the largest fraction of 4 GB jobs, making the conservation of memory resources of prime importance. At the same time, Best-Fit-Memory performs relatively poorly on pool D.

Similarly, Worse-Fit-Cores that was the best heuristic in the buckets experiment (except for Mix-Fit) appears to perform poorly in the load simulation in both pools A and B. This may stem from the fact that the buckets experiments were conducted in a highly artificial setting where all jobs were presented in advance, and were matched to empty clusters of machines. In such a scenario Worse-Fit-Cores — which is similar to round-robin allocation — performed well, but when confronted with a continuous on-line scenario, where machines typically already have some of their resources taken, it did not. This is another indication that challenges faced by on-line schedulers are different from those faced by batch (or off-line) schedulers, and that it is important to match the simulation type to the system type. In our case this means that the dynamic simulations described here are more relevant than the bucket experiments used above.

In pool C all the heuristics achieved essentially the same performance. This reflects an unchallenging workload that can be handled by any heuristic.

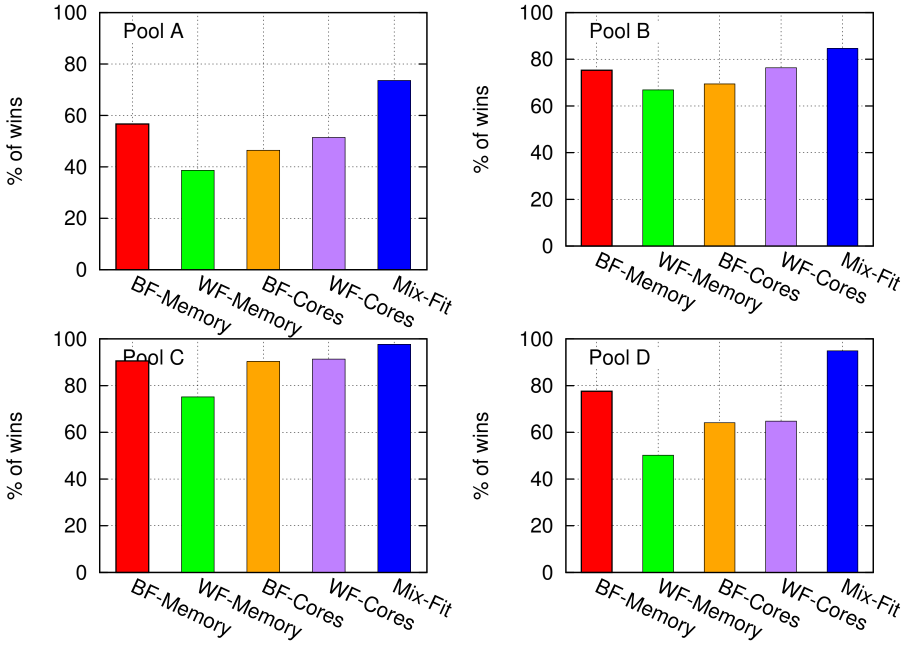


Figure 6.4: Selected heuristics by Max-Jobs. Sum is more than 100% because in many cases several heuristics produced the same result.

Finally, in pool D Mix-Fit had similar results to the second best heuristic, Worse-Fit-Cores. It looks like the non-bursty nature of that pool gives an advantage to balancing heuristics such as Worse-Fit-Cores.

Figure 6.4 shows the fraction of times each heuristic was selected by Max-Jobs. As can be seen, Mix-Fit is dominant, even more than in the above buckets experiment, but still getting as low as 73% in pool A. Best-Fit-Memory is markedly better than Worse-Fit-Cores especially in pools A and D.

As expected, the Max-Jobs meta-heuristic is the best scheme all around, and seems to be largely robust against workload and configuration variations. This is due to the fact that it uses the best heuristic at each scheduling cycle. However, its final result (in terms of average performance across all the jobs in the trace) is not necessarily identical to that of the best heuristic that it employs. On one hand, Max-Jobs can be better than each individual heuristic, as happens for example in pool B. This is probably because it can mix them as needed, and use a different heuristic for different situations as they occur. On the other hand, Max-Jobs is sometimes slightly inferior to the best individual heuristic, as seen for example in pool A. This is probably due to situations in which packing jobs very densely leads to reduced performance in a successive scheduling round.

7 Related Work

There are very few externally available publications that relate to NetBatch. Zhang et al. investigated the use of dynamic rescheduling of NetBatch jobs between pools which improves utilization at the farm level [24]. Our work in effect complements theirs by focusing on utilization improvements within the individual pools.

The question of assigning machines to jobs has received some attention in the literature. Xiao et al. studied a problem similar to ours and also concluded that one-dimensional strategies yields sub-optimal performance [22]. In their work, however, cores are considered shared resources, and thus the investigation focused on the interference between the jobs. Amir et al. proposed a load balancing scheme where the targets for process migration are selected so as to avoid saturation of any single resource [5]. This is similar to avoiding high α values in our terminology.

The idea of symbiotic scheduling is also related to our work. Symbiotic scheduling attempts to find sets of jobs that complement each other and together use the system resources effectively. This was initiated in the context of hyper-threading (or simultaneous multi-threading) processors [17, 10], and extended also to the domain of clusters [21].

Meta-schedulers like the Max-Jobs approach have also been used before. For example, Talby used such a meta-scheduler to select among different versions of backfilling in scheduling large-scale parallel machines [19]. However, this was done by simulating recent work in the background and then switching to the version that looks best. Such an approach depends on an assumption of locality, meaning that the future workload will also benefit from this version. In our work we actually run all the contending variants on the jobs in the queue, and select the one that indeed achieves more assignments.

Another meta-scheduler example is the portfolio scheduler [8] that was developed in parallel to our work. The portfolio scheduler is a general-purpose mechanism that applies to scientific computing with various target functions for scheduling. Max-Jobs on the contrary, applies to batch systems and its target function is specified as maximizing the total number of running jobs.

Reservations with on-line backfilling, such like was suggested by Srinivasan et al.[18], may improve utilization and throughput. However at Intel, reservations are preferred over backfilling so such an approach was not evaluated.

It should be noted that due to the assumption that cores and memory are allocated exclusively to jobs, our problem is not directly related to the well-known 2D bin-packing problem. In particular, it is not allowed to pack multiple jobs with limited memory requirements onto the same core [12]. It does, however, correspond the problem of allocating virtual machines to physical servers which has gained much attention in recent years. This has been called the *vector bin-*

packing problem, since the allocation can be depicted as the vector-sum of vectors representing the resource requirements of individual virtual machines [13]. This directly corresponds to our depiction of rectangles that connect at their corners in Figures 3.1, 3.2, etc.

The ideas suggested for vector bin-packing are all very similar to our Mix-Fit algorithm. For example, they are also based on normalizing the resources and creating a square (or multi-dimensional cube, if there are more resources than 2). The available resources are then represented by a diagonal vector, the consumption by other vectors, and the basic idea is to try to make these vectors close to each other. However, the details may differ.

Mishra and Sahoo [12] describe the SandPiper algorithms used in Xen, and the VectorDot algorithm [16]. They show that both suffer from failures similar to the ones we demonstrated in Section 3. For example, the VectorDot algorithm uses the dot product of the consumed resources vector and the request vector to identify requests that are orthogonal to the current usage, and thus may be expected to complement it. However, this is subject to artifacts because the lengths of the vectors also affect the result. They then suggest a rather complex approach for identifying complementary vectors based on a discretization of the space called the “planar resource hexagon”. They did not, however, evaluate its performance compared to existing heuristics.

Panigrahy et al. study a wide range of First-Fit-Decreasing-based algorithms [13]. The idea is to combine the requirements for different resources in some way into a single number, and then pack in a decreasing order. However, this approach loses the important geometrical structure of the problem. They therefore also consider heuristics based on the dot product or the maximal required resource. The evaluations are based on presumed synthetic distributions.

Compared with these previous works, our approach of using just the angle between two vectors is among the simplest. Basing the comparison at the top-right corner for improved discrimination seems to be novel. It would be interesting to evaluate the effect of these details, but our results so far indicate that they may not have much impact for real workloads.

Another approach for utilization improvement that was previously introduced is altering resources requirements of job submission. Yom-Tov And Aridor suggested reduction of memory requirements [23]. This approach is orthogonal to ours, and can be combined in the same system altogether. Tang et al. adjusted job's runtime estimates [20]. This approach is not relevant to NetBatch . NetBatch scheduler uses transient reservations, hence there is no promise about the actual dispatch time of jobs. In addition, our jobs did not have any runtime estimates during submission.

Lee et al. investigated the problem of virtual machines allocation taking into consideration the consolidation of virtual machines onto the same physical platform, and the possible resulting resource contention [11]. In principle such considerations are also applicable to our work. However, we note that the configuration of NetBatch pools is such that I/O and bandwidth are seldom a bottleneck.

Finally, it is important to remember that since the PPM considers the jobs one at a time there is a limit on the optimizations that can be applied. Looking further into the queue and considering more than one job may yield significant improvements [15].

8 Conclusions

Matching jobs with resources is an NP-hard problem. The common practice is therefore to rely on heuristics to do the matching. In this work we investigated the problem of resource matching in Intel's compute farm, and showed that none of the well-known heuristics such as Best-Fit or Worse-Fit yield optimal performance in all workload scenarios and cases. This stems from two reasons. First, these heuristics focus on a single resource, either cores or memory, whereas in reality the contention may apply to the other resource. To address this problem we implemented a specialized heuristic, Mix-Fit, that takes both resources into account and tries to create an assignment that leads to a balanced use of the resources. In principle this can be extended to more than two resources. While this too failed to be optimal in all cases, it did show some improvement under certain conditions.

Second, the nature of dynamically changing demands prevent a specific use-case-tailored algorithm to be optimal for all cases. For that, we proposed a meta-heuristic called Max-Jobs, that is not tailored to a specific workload or scenario. Rather, it uses the other heuristics as black-boxes, and chooses, in every scheduling cycle, the one that yields the maximal number of matched jobs. We have demonstrated through simulations that max-jobs is highly competitive with all the individual heuristics, and as such is robust against changes in workload or configuration.

References

[1] Htcondor. http://research.cs.wisc.edu/htcondor/.

[2] Oracle® Grid Engine® white paper. http://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-overview-167117.pdf.

[3] Platform LSF® - IBM® grid computing products and services. http://www.redbooks.ibm.com/redbooks/pdfs/sg246650.pdf.

[4] The Parallel Workloads Archive – NetBatch traces. http://www.cs.huji.ac.il/labs/parallel/workload/l_intel_netbatch/index.html, 2013.

[5] Yair Amir, Baruch Awerbuch, Amnon Barak, R. Sean Borgstrom, and Arie Keren. An opportunity cost approach for job assignment in a scalable computing cluster. *IEEE Trans. Parallel and Distributed Systems*, 11(7):760–768, Jul 2000.

[6] Bob Bentley. Validating the Intel® Pentium® 4 microprocessor. In *38th Design Automation Conf.*, pages 244–248, Jun 2001. doi: 10.1145/378239.378473.

[7] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Proceedings of the Job Scheduling Strategies*

for Parallel Processing, IPPS/SPDP '99/JSSPP '99, pages 67–90, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66676-1. URL `http://dl.acm.org/citation.cfm?id=646380.689537`.

[8] Kefeng Deng, Ruben Verboon, Kaijun Ren, and Alexandru Iosup. A periodic portfolio scheduler for scientic computing in the data center. In *17th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2013)*, Boston, USA, May 2013.

[9] Nicholas D. Evans. *Business Inovation and Disruptive Technology: Harnessing the Power of Breakthrough Technoloy ...for Competitive Advantage*. Financial Times Prentice Hall, 2003.

[10] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *15th Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pages 91–102, Mar 2010. doi: 10.1145/1736020.1736033.

[11] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. Technical Report MSR-TR-2011-9, Microsoft Research, Jan 2011.

[12] M. Mishra and A. Sahoo. On theory of VM placement: Anomalies in existing methodologies and their mitigation using a novel vector based approach. In *IEEE Intl. Conf. Cloud Computing*, pages 275–282, 2011. doi: 10.1109/CLOUD.2011.38.

[13] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2011.

[14] Ohad Shai. Batch simulator (simba). Open source project hosted at: http://code.google.com/p/batch-simulator, 2012.

[15] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of parallel and distributed computing*, 65:1090–1107, 2005.

[16] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *SC 2008: High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2008. doi: 10.1109/SC.2008.5222625.

[17] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *9th Intl. Conf. Architect. Support for Prog. Lang. & Operating Syst.*, pages 234–244, Nov 2000.

[18] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. Selective reservation strategies for backfill job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 55–71. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-00172-0. doi: 10.1007/3-540-36180-4_4. URL `http://dx.doi.org/10.1007/3-540-36180-4_4`.

[19] David Talby and Dror G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *19th Intl. Parallel & Distributed Processing Symp.*, Apr 2005. doi: 10.1109/IPDPS.2005.252.

[20] Wei Tang, N. Desai, D. Buettner, and Zhiling Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11, April 2010. doi: 10.1109/IPDPS.2010.5470474.

[21] Jonathan Weinberg and Allan Snavely. Symbiotic space-sharing on SDSC's DataStar system. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 192–209. Springer Verlag, 2006. doi: 10.1007/978-3-540-71035-6_10. Lect. Notes Comput. Sci. vol. 4376.

[22] Li Xiao, Songqing Chen, and Xiaodong Zhang. Dynamic cluster resource allocations for jobs with known and unknown memory demands. *IEEE Trans. Parallel and Distributed Systems*, 13(3):223–240, Mar 2002.

[23] Elad Yom-Tov and Yariv Aridor. A self-optimized job scheduler for heterogeneous server clusters. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 4942 of *Lecture Notes in Computer Science*, pages 169–187. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78698-6. doi: 10.1007/978-3-540-78699-3_10. URL `http://dx.doi.org/10.1007/978-3-540-78699-3_10`.

[24] Zhuoyao Zhang, Linh T. X. Phan, Godfrey Tan, Saumya Jain, Harrison Duong, Boon Thau Loo, and Insup Lee. On the feasibility of dynamic rescheduling on the intel distributed computing platform. In *Proc. 11th Intl. Middleware Conference Industrial track*, pages 4–10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0456-6. doi: 10.1145/1891719.1891720. URL `http://doi.acm.org/10.1145/1891719.1891720`.

תקציר

במסגרת מחקר זה, אנו בוחנים שיטות להתאמת עבודות למכונות בחוות המחשוב של אינטל. אנו מראים כי יוריסטיקות נפוצות כגון Best-Fit או Worst-Fit, עלולות לפגוע בנצילות השרתים כאשר הם מופעלות באופן חד מימדי על כמות הזיכרון הפנוי או ניצולת המעבד בשרתים.

בניסיון להתגבר על בעיה זו הצענו יוריסטיקה חדשה Mix-Fit, אשר מנסה לאזן בין דרישות דו ממדיות. במחקר רואים שיפור כלשהו בשימוש ביוריסטיקה זו, אך ניצול המשאבים עדיין אינו אופטימאלי. כפתרון אנו מציעים את Max-Jobs, מטה-יוריסטיקה שמשתמשת בתוצאות היוריסטיקות האחרות כדי לשפר את הפתרון הכללי על ידי בחירה אד-הוק של היוריסטיקה הטובה ביותר בכל התאמה מחדש.

במסגרת המחקר הרצנו סימולציות לבחינת התוצאות. הסימולציות הורצו בעזרת רשומות של עבודות שנאספו ב 4 חוות שרתים מהגדולות של אינטל במשך חודש. מתוצאות המחקר ניתן ללמוד כי Max-Jobs היא אכן היוריסטיקה החסינה ביותר מפני שינויים ויכולה להביא לשיפור של עד 22% בזמן ההמתנה הממוצע של עבודות בחווה.

אוניברסיטת תל-אביב
הפקולטה למדעים מדוייקים
ע"ש ריימונד ובברלי סאקלר
בית הספר למדעי המחשב ע"ש בלבטניק

שיטות התאמת עבודות למכונות
בחוות השרתים של אינטל

חיבור זה הוגש כחלק מהדרישות לקבלת התואר
"מוסמך אוניברסיטה" (M.Sc.) באוניברסיטת תל-אביב

בית הספר למדעי המחשב

על ידי אוהד שי

העבודה הוכנה בהדרכתם של פרופ' סיוון טולדו ופרופ' דרור פייטלסון

אב תשע"ד