Improving and Stabilizing Parallel Computer Performance Using Adaptive Backfilling

David Talby Dror G. Feitelson
Hebrew University
Email: {davidt,feit}@cs.huji.ac.il

Abstract

The scheduler is a key component in determining the overall performance of a parallel computer, and as we show here, the schedulers in wide use today exhibit large unexplained gaps in performance during their operation. Also, different scheduling algorithms often vary in the gaps they show, suggesting that choosing the correct scheduler for each time frame can improve overall performance. We present two adaptive algorithms that achieve this: One chooses by recent past performance, and the other by the recent average degree of parallelism, which is shown to be correlated to algorithmic superiority. Simulation results for the algorithms on production workloads are analyzed, and illustrate unique features of the chaotic temporal structure of parallel workloads. We provide best parameter configurations for each algorithm, which both achieve average improvements of 10% in performance and 35% in stability for the tested workloads.

1. Introduction

Large parallel computers are often shared by many different users. A user wishing to run a job submits a request to a central scheduler, which has the responsibility to decide when and where the job will run. The scheduler is an on-line algorithm, which must optimize the execution of the incoming workload, balance throughput and fairness, and encapsulate the system's administrative policies and priorities. This makes the scheduler a key component, which determines the bottom-line level of service that users receive. Improving it can have a dramatic visible effect on a system's usability, for a given platform and policy.

This study began with the observation that widespread scheduling systems in use today, such as Maui [4] and Easy [6], sometimes exhibit inconsistent and unstable behavior. As the next section will show, in a typical year these schedulers have several "good" months, a few "bad" ones, and one or two "major blunders". In very bad months, the computer's average response time is several times slower than usual.

Another observation that we make is that another scheduling algorithm, Flex [7], offers comparable performance to Maui, with an important difference: Although it also has "good" and "bad" months, they are different than those of Maui in every case we examined. This raises an opportunity: If we could predict when each scheduler should be used, we can enjoy the best of both worlds.

This paper describes two algorithms that use different strategies to predict when each scheduler is most appropriate. There are two end goals: Improve the overall performance of the computer, and maintain a stable and predictable quality of service. In addition, the analysis of each algorithm will reveal new hints about the key question of *why* a given algorithm is better for a given workload.

2. The Inconsistent Performance Problem

2.1. Review of Backfilling Schedulers

The parallel computers considered in this paper are of the most widespread type today, which uses variable partitioning: A new job requires a certain number of processors upon its arrival, and these processors are dedicated to it once it starts running. In addition, each job provides an estimate for its runtime; this is an upper bound, since if a job exceeds it, it is killed. Users also have an incentive to provide low estimates, since this enables promoting jobs from the back of the queue to fill idle processors – an optimization known as backfilling.

This study compares four well-known backfilling schedulers. The first is the EASY Scheduler [6], used mainly in IBM SP2 machines since the mid '90s. When a new job is submitted, EASY checks its requirements against the currently running jobs and the first job in

the queue. The time in which the first job in the queue is going to run is called the shadow time; the idle nodes after the first queued job starts running are called extra nodes. The new job is backfilled (passes the first job in the queue and starts running immediately) if one of the following two conditions holds:

1. It requires no more than the currently free nodes, and will terminate by the shadow time.
2. It requires no more than the minimum of the currently free nodes and the extra nodes.

EASY uses an aggressive backfilling strategy, in the sense that the above two conditions are only checked for the *first* (oldest) job in the queue. This strategy is aimed at improving the response time and slowdown of small jobs. However, this comes at a price: subsequent queued jobs (not the first) can suffer an unbounded delay, and a job cannot be guaranteed when it will run.

The second backfilling scheduler, the Conservative Scheduler [5], uses a different rule: A job can be backfilled only if does not delay *any* job in the queue. This enables runtime guarantees and decreases starvation on one hand, but may hamper utilization and responsiveness for small jobs on the other.

The third scheduler is Maui [4]. Maui is a high-end scheduling system, successfully deployed in many computing sites over the past few years, from IBM SP2 machines to Linux clusters [1,4]. The scheduler supports backfilling, and is highly configurable: In particular, the number of jobs that cannot be delayed and the order in which jobs are backfilled can be controlled. The default and recommended configuration (which is used in all simulations in this paper) is a variation on EASY: Make reservations for the first job in the queue only, and backfill jobs by FCFS. That is, when a new job arrives, it is not the only one that the scheduler tries to backfill. Instead, all waiting jobs are sorted in order of ascending arrival time, and are then backfilled (this in fact rebuilds the queue) in that order.

The fourth scheduler that we consider here is the Flex Scheduler [7]. Flex takes a different approach from the above three algorithms, by trying to reach a global optimization of the entire queue, rather than just the head of the queue. This means that whenever a decision has to be made (job arrival, termination or cancellation events), all possible queues are compared, and the best one (according to a configurable criteria) is chosen. For example, if two jobs are queued and a third arrives, Flex will consider three alternative schedules: Running the new job first, in the middle, and last. Each possible schedule is graded, suffering a penalty for every job that must wait. To prevent starvation, Flex introduces the concept of slack: Each job is given a slack upon arrival, and it can never be delayed by *more than its slack*. This is safer than EASY and enables runtime guarantees, yet is more flexible than Conservative.

2.2. Performance Comparison

The four algorithms have been studied and compared in depth [5,7]. The bottom-line results as reported so far are as follows:

- Easy is generally better than Conservative under the response time metric, and sometimes also under bounded slowdown, mainly in high-load workloads.
- Flex offers a level of performance 10%-20% better than that of Easy, depending on the workload. As we show here, Maui provides similar performance.

We have repeated these experiments by simulation on logs from three parallel IBM SP2 computers, which together constitute almost six years of real user activity.

Symbol	Machine	# Nodes	# Jobs	# Weeks
SDSC BLUE	ASCI Blue Horizon	1152	250440	140.1
SDSC SP2	IBM SP/2	128	73496	105.2
KTH SP2	IBM SP/2	100	28490	48.6

Table 1: Parallel Computers in Our Data Set

Table 2 uses both the average wait time and average bounded slowdown to compare the schedulers. The first metric is equal towards all jobs, while the second one penalizes causing short jobs to wait, encouraging better interactive behavior. This table confirms past results: Flex is the best scheduler in wait-time in two out of three cases, and in slowdown in one of the three; Maui is the best scheduler in the other cases, and Conservative and Easy typically have lower performance. However, this is not the whole picture.

Wait Time				
	Cons	Easy	Flex	Maui
BLUE	1878	2083	1661	1535
SDSP2	6379	6033	5500	5612
KTH	7302	6806	6250	6731
Bounded Slowdown				
	Cons	Easy	Flex	Maui
BLUE	9.1	11.0	8.0	10.2
SDSP2	28.4	29.2	25.7	17.6
KTH	89.2	88.9	70.5	63.3

Table 2: Schedulers' Performance for Entire Logs

2.3. Monthly Performance Gaps

In order to gain a better understanding of the algorithms than previously done, we re-ran the simulations and gathered per-month results. This is significant since as shown in [8] the usage pattern in production logs may change radically over time, and also because this gives us a total dataset of 67 unique workloads, instead of three. This monthly breakdown uncovered an unexpected result, shown in table 3.

Wait Time				
	Cons	Easy	Flex	Maui
BLUE	0	0	8	23
	0%	0%	26%	74%
SDSP2	0	2	6	15
	0%	9%	26%	65%
KTH	0	1	2	8
	0%	9%	18%	73%
Bounded Slowdown				
	Cons	Easy	Flex	Maui
BLUE	2	0	7	22
	6%	0%	23%	71%
SDSP2	3	0	4	16
	13%	0%	17%	70%
KTH	0	0	3	8
	0%	0%	27%	73%

Table 3: Schedulers' Superiority by Months

The table counts how many months, in absolute and relative terms, each scheduler "won" by providing the best performance among the four schedulers. As can be seen, Maui has a clear lead and "wins" about 70% of the months, under both metrics, consistently across all logs. When only Flex and Maui are compared, Maui "wins" about 75% of the months, and Flex wins the other 25%.

This is surprising because as we saw in Table 2, Maui's overall yearly statistics are not better than Flex's. This can only happen if Flex wins by very high margins when it does, and this is indeed what we found: about one month per year, Maui shows catastrophic performance, by presenting average wait times that are 2-4 *hours* longer than that of Flex. Such a wait time is about three times the yearly average on each of the machines we study, and is enough to make Flex the more responsive scheduler overall in two of three cases.

Flex shows considerable improvement over Maui in the months in which it leads, but this works the other way as well: Maui often outperforms Flex significantly. Table 4 shows both the average and maximum monthly gap in average wait time between these two schedulers; the gap is shown in percentage to make it possible to compare across logs. The numbers are consistently very high, which is an important unexpected fact, on which we base the second part of this work.

The last two rows of table 4 prove that these results are largely not the result of outliers in the logs. Following the work on flurries in [10], we re-ran the simulations on "cleaned" versions of the logs – such versions exist for the BLUE and SDSP2 logs.

	Months won by Flex		Months won by Maui	
	Average Gap	Max Gap	Average Gap	Max Gap
BLUE	46%	251%	43%	189%
SDSP2	44%	97%	29%	145%
KTH	56%	158%	29%	53%
BLUE cln	46%	144%	33%	147%
SDSP2 cln	26%	95%	25%	93%

Table 4: Average and Maximal Wait Time Gaps between Flex and Maui

The results show that although the removal of flurries (which exist in real-life workloads) reduces some of the figures, most of the performance gap remains unexplained.

Months in which the four schedulers greatly differ also do not stand out statistically as outliers. On the contrary – the statistics of workload attributes such as the inter-arrival time, runtime and parallelism is often similar across months with very different scheduling performance. Therefore, we suspect that the cause of the performance gaps is the temporal structure of the workload, or the correlations between adjacent jobs. More specifically, it is likely that self-similarity – shown in [8] to exist in parallel workloads – plays a role here, since it exhibits itself in long-term correlations between jobs, that cause long-term (i.e. over months) load patterns that change chaotically over time. However, the focus of this paper is not the theoretical explanation of the performance gap phenomenon, but rather a practical utilization of it, by using the most practical tool to handle chaos in dynamic systems: adaptability.

3. Performance Based Adaptive Scheduling

3.1. The Rationale

The practical opportunity that the performance gap presents is clear: we can theoretically improve performance by 30-40%, if we could predict in advance which scheduler will win each month. In practice, we can't predict the future but can afford mistakes, since a 10% gain will be a significant and very useful achievement as well.

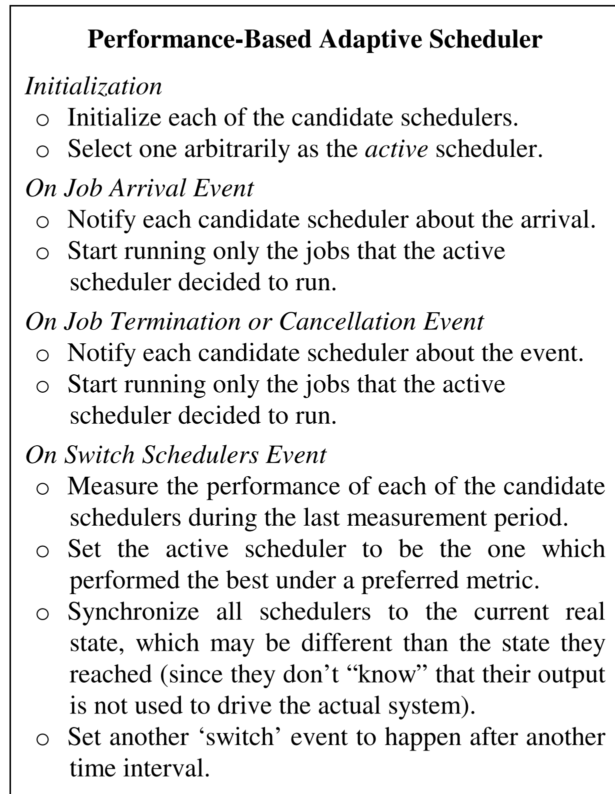
An adaptive algorithm makes sense only if it can make one very basic assumption – that past behavior predicts future behavior. This is ensured by locality. Locality is often caused by users working in sessions, which has been noted to cause very similar or even identical jobs to be repeatedly submitted over short periods of time. In addition, the daily and weekly cycles of human users cause similar load conditions in successive hours and days. Finally, long-range dependence implied by self-similarity is also very evident [8]. All of these overlapping features ensure us that *some* time scales will enable us to make predictions about the workload, but also complicates the situation, since the different types of correlations are not continuous and may sometimes cancel out one another. Finding the "right" time scales will be a major challenge in making the adaptive schedulers work.

3.2. The Algorithm

The performance-based adaptive scheduler (PBAS), summarized in figure 1, assumes that the scheduler that performed best in the recent past is likely to perform best in the near future. The idea is simple: Run a set of *candidate schedulers* in the background, while using one of them as the *active* scheduler; at regular intervals, review the performance of all schedulers, and choose the one with the best performance to be the active scheduler for the next time frame.

3.3. Empirical Results

The PBAS algorithm can be configured by four parameters: The candidate schedulers set, the time frame between switching events, the metric used to compare schedulers' performance, and the history time frame to consider when comparing performance. In our runs, the history time frame was always identical to the switching time frame, because using a longer history period has an unclear semantics when switching happens during that period, and we did not want to add to the algorithm's complexity.


Performance-Based Adaptive Scheduler

Initialization
- Initialize each of the candidate schedulers.
- Select one arbitrarily as the *active* scheduler.

On Job Arrival Event
- Notify each candidate scheduler about the arrival.
- Start running only the jobs that the active scheduler decided to run.

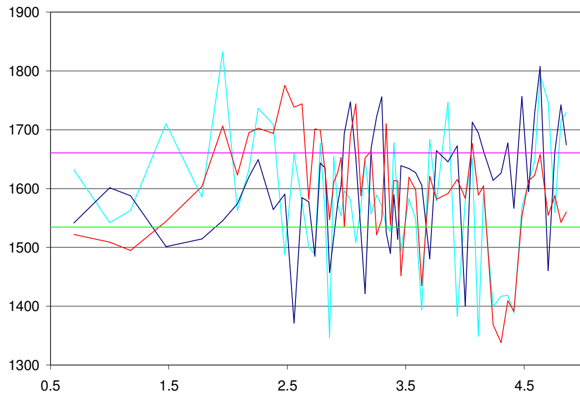
On Job Termination or Cancellation Event
- Notify each candidate scheduler about the event.
- Start running only the jobs that the active scheduler decided to run.

On Switch Schedulers Event
- Measure the performance of each of the candidate schedulers during the last measurement period.
- Set the active scheduler to be the one which performed the best under a preferred metric.
- Synchronize all schedulers to the current real state, which may be different than the state they reached (since they don't "know" that their output is not used to drive the actual system).
- Set another 'switch' event to happen after another time interval.

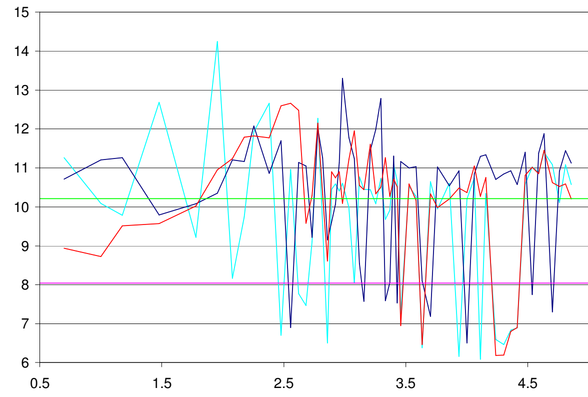
Figure 1: PBAS Algorithm

The six graphs of figure 2 summarize the PBAS results for different metrics (wait time and bounded slowdown) and logs. The candidate schedulers set includes only Flex and Maui, for a reason that will be explained later. The time scale is logarithmic, since we tested over five orders of magnitude of time – from one minute to fifty days. The metrics used to judge performance and decide to switch schedulers are response time (PBAS-Resp), bounded slowdown (PBAS-BSld), and utilization (PBAS-Util). The wait time and slowdowns metrics were tested as well, but result in similar or worse performance so were left out.

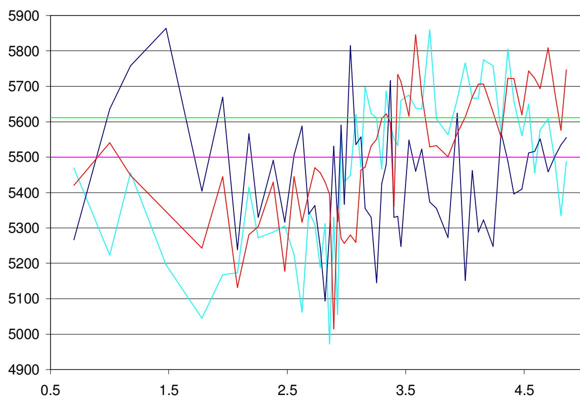
Perhaps the most surprising visual impact of figure 2 is the fact that results are highly non-continuous: performance seems to be very sensitive to the time frame, and does not lend itself to any elegant explanation. There are several unique points which are consistent across logs, but most of the time, behavior can only be predicted by experimentation. The same happens across metrics: using different metrics for same time frames results in inconsistent and non-continuous performance. Also, with a few rare exceptions, using a certain metric to decide on switching does not guarantee good results in that metric.



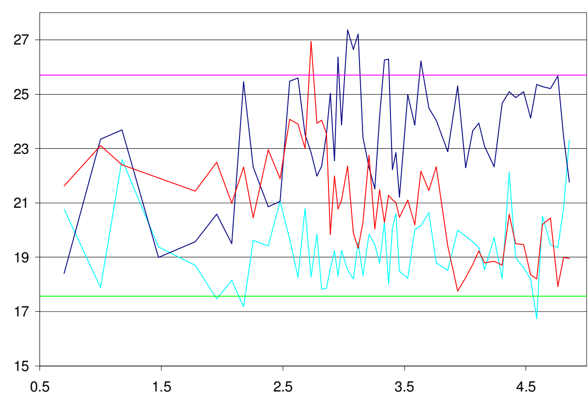
BLUE



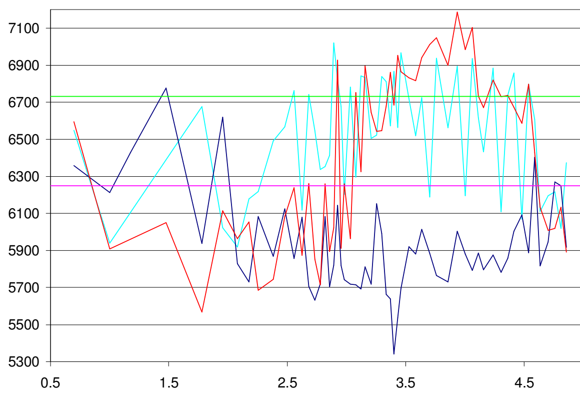
BLUE



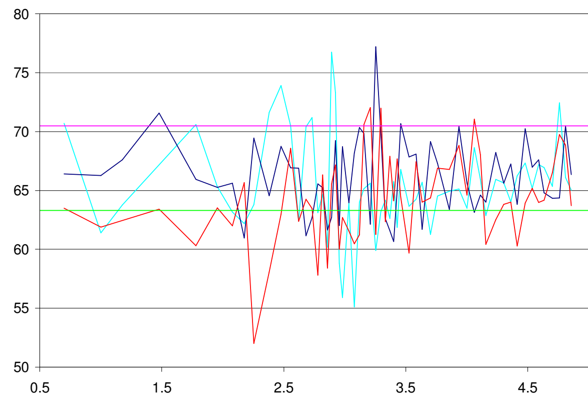
SDSP2



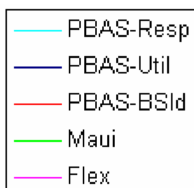
SDSP2



KTH



KTH



X Axis is log-minutes:
Hour≈1.7, Day≈3.2, Week≈4

Y Axis is **Wait Time** on left &
Bounded Slowdown on right

Figure 2: PBAS Results by Time Frame and Metric

Maui and Flex are shown as straight lines in the plots, since they are not adaptive. The visuals show that most PBAS runs are between the Maui and Flex results for that log, which is expected – after all, the adaptive scheduler can only run one of them at any given time.

It is also visible that the wrong selection of parameters can lead to performance that is worse than both Maui and Flex, but on the other hand – the right selection can result in outperforming both. It is

expected and doesn't matter than most parameter combinations don't work, as long as one or more combination consistently improves performance. In our case, there are two consistently winning combinations, presented in table 5. "Util-7 Days" means switch every 7 days, to the scheduler that achieved the highest utilization in the last period. The numbers show the percent of improvement over Flex and Maui, both in average wait time (performance), and in standard deviation between wait times of months (stability).

Configuration	Versus	KTH	SDSP2	BLUE
		Performance Gain		
Util 7-Days	Maui	13%	8%	9%
	Flex	6%	6%	16%
Resp 12-Hour	Maui	5%	11%	12%
	Flex	-3%	10%	19%
		Stability Gain		
Util 7-Days	Maui	63%	29%	17%
	Flex	-2%	4%	14%
Resp 12-Hour	Maui	24%	27%	24%
	Flex	-11%	3%	22%

Table 5: PBAS Best Parameter Configurations and Overall Improvements over Flex and Maui

The average performance gain of the Utilization-7 Days configuration over Maui is 10%, and its average stability gain is 36%. The numbers for the Response Time-12 Hours configuration are 9% and 25%. The average performance gains for Flex are similar, but the stability gains are smaller, since Flex is more stable than Maui to begin with – for example, it is 64% more stable than Maui in the KTH log. Another good configuration was Bounded Slowdown-1 Hour, which worked very well for KTH and SDSP2, but caused a small decline in performance of the BLUE log.

3.4. Candidate Schedulers

All the PBAS experiments analyzed so far use only Maui and Flex, for a simple reason – using any other candidate set, and in particular the set of all four schedulers, provides significantly worse results. Figure 3 demonstrates this visually, for specific logs and PBAS configuration – but the results repeat for other configurations and logs as well.

The explanation is as follows. Since the adaptive scheduler makes local decisions, there are occasions when Easy or Cons will be chosen. Since these algorithms are generally weaker than Flex and Maui, the probability that their past success indicates future success is smaller than for Flex and Maui. This means that erroneous choices are made more often.

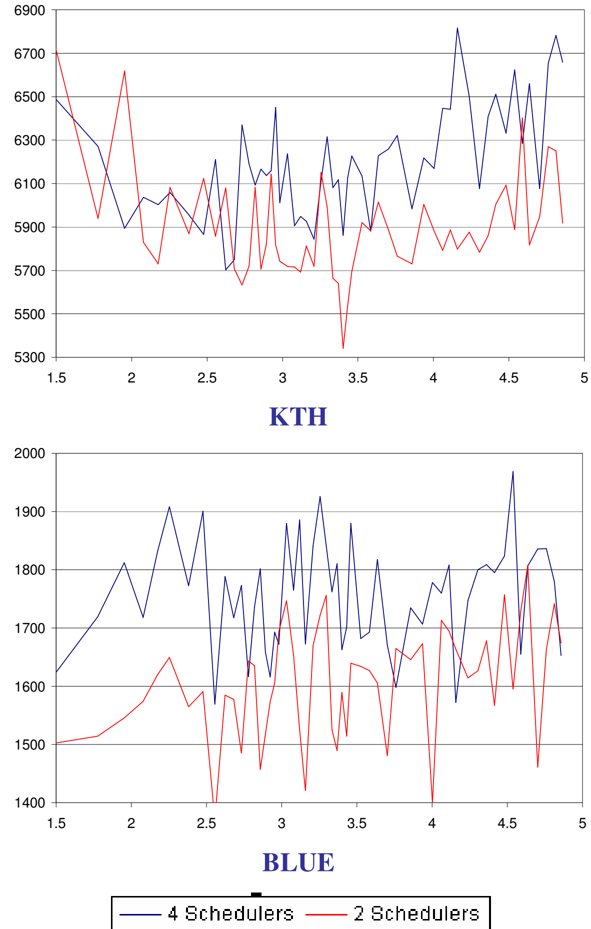


Figure 3: 4-Scheduler PBAS vs. 2-Schedulers PBAS Y Axis is wait time, switching metric is utilization

Table 6 illustrates this from a different point of view, by listing the portion of time that PBAS uses each scheduler. The five rows are results for the BLUE log, using utilization as the switching metric, and all four schedulers as the candidate set. As the table shows, Cons and Easy are used together between 21% and 47% of the time, depending on the time frame. This means that during this portion of the time, PBAS uses an inferior scheduler – and as figure 3 shows, this is apparent in its performance. To borrow an analogy from sports: If an adaptive algorithm is a team's coach, then it should prefer to use a strong player on a bad day, than a mediocre player on a winning streak.

	% Flex	% Maui	% Easy	% Cons
30 min	27%	52%	17%	4%
4 hours	22%	47%	23%	8%
24 hours	21%	52%	19%	9%
7 days	11%	43%	40%	7%
50 days	17%	44%	33%	6%

Table 6: PBAS Switching Patterns

4. Workload Based Adaptive Scheduling

4.1. The Rationale

Performance-Based Adaptive Scheduling takes a greedy approach to profit from the performance gap. The ability to bypass the need to understand the root cause of a phenomenon and still benefit from it is the key advantage of adaptive algorithms – but yet, if such an understanding can be reached, then the rewards in terms of performance are usually high [3]. On the other hand, some dynamic environments are too complex to fully "understand", and then an adaptive heuristic behavior is the best practical choice.

Workload-Based Adaptive Scheduling is an intermediate path between these extremes. Instead of blindly following the best-performing candidate scheduler, we will examine which local features of the workload are correlated with preferring a given scheduler. Afterwards, our on-line algorithm will work by monitoring that feature, and switching schedulers accordingly. This should enable us to benefit by identifying and reacting to a new trend in the workload as its first jobs enter the queue – instead of when the first jobs terminate, when the first performance statistics become available for PBAS.

The workload variables that were tested are the number of jobs and users, the load, and the medians and intervals of the runtime, parallelism, total CPU work and inter-arrival time. The average and standard deviation metrics were not used, since they are known to be unreliable in parallel workloads due to the heavy tails of the involved distributions [2]. These variables were compared against two measures of relative algorithmic superiority: the absolute difference between the wait times of Flex and Maui, and the relative difference (in percent) between the wait times of the two algorithms. The first measure tests correlation to "major blunders" since large absolute values dominate the computation, while the second (relative) regards small and large wait times equally.

performance of Maui over Flex. Negative values indicate that large values correlate with better Flex performance. The correlations were computed between the statistics of each month of a log, to the difference between Flex and Maui's performance for that month. As the table shows, most variables do not consistently indicate a preference towards a certain algorithm, and this dataset is too small to assume that a majority means anything. There are two exceptions: the number of processors (both median and interval) for both the absolute and relative performance measures, and the median of inter-arrival times or related job count for the absolute or relative measure, respectively.

4.2. The Algorithm

The Workload-Based Adaptive Scheduler (WBAS) uses the number of processors as an indicator of scheduler preference. It was chosen since it works for both the relative and absolute measures, and since its correlations were the most consistent across logs.

The algorithm is adaptive and works in the following manner. When a switching event occurs, it checks if the number of processors requested in the last time frame is greater than the long-term average parallelism of the workload – which indicates that Flex should be activated – or not, which means that Maui should be chosen. The long-term average parallelism of the workload is updated each time a job arrives, using an exponential moving average, to enable the global average to slowly adjust even after the scheduler has been running for a long time.

Log	Jobs Count	Runtime Load	Users /TJobs	Runtime Median	Runtime Interval	Procs Med.	Procs Int.	CPU Med.	CPU Int.	IA-Med.	IA-Int.
Correlations to relative Flex/Maui difference:											
KTH	0.44	-0.35	0.13	0.83	-0.38	-0.05	-0.60	N/A	N/A	-0.46	0.64
SDSP2	0.01	0.11	-0.05	-0.04	0.28	-0.30	-0.19	-0.02	0.29	0.02	-0.11
BLUE	0.20	-0.05	-0.09	0.14	-0.02	-0.33	-0.24	0.00	0.09	-0.21	-0.13
Correlations to absolute Flex-Maui difference:											
KTH	0.39	-0.03	0.11	0.32	-0.28	-0.14	-0.72	N/A	N/A	-0.52	-0.14
SDSP2	0.10	0.08	-0.16	-0.24	0.16	-0.19	-0.21	-0.16	0.19	-0.11	-0.28
BLUE	-0.01	-0.12	0.07	0.02	0.13	-0.09	-0.02	0.01	0.17	-0.11	0.08

Table 7: Correlations between workload attributes and relative Flex-Maui performance

Initialization

- Select one of the candidate schedulers arbitrarily as the *active* scheduler

On Job Arrival Event

- Update the moving average of parallelism *AP* using the number of processors requested by the new job P_{new} and this formula:

$$AP = (1 - MAF^{-1}) \times AP + MAF^{-1} \times P_{new}$$

- Start running only the jobs that the active scheduler decided to run.

On Job Termination or Cancellation Event

- Only start running jobs the active scheduler decided to run.

On Switch Schedulers Event

- Measure the average level of parallelism of jobs that arrived during the last time frame: AP_{last}.
- If $AP < AP_{last}$, then activate Flex, else use Maui.
- If the active scheduler changed, synchronize it to the current situation.
- Set another 'switch' event to happen after another time interval.

Figure 4: WBAS Algorithm

4.3. Empirical Results

The WBAS can be configured by two parameters: The switching time frame, and the moving average factor (MAF) which determines the length of the "long-term memory" of the algorithm. The candidate set and history time frame, in the sense used in PBAS, are irrelevant here. Since both parameters are continuous, we could not test all possible combinations in order to find the best one, and therefore took the following approach to find the best combination. In figure 5, all the performance-per-switch-time graphs use MAF of 5000, and all the performance-per-MAF graphs use one day as the switch time. These values were chosen after preliminary test runs, which indicated that these values both perform well and are relatively stable to small changes. The performance measure in figure 5 is wait time in all graphs; the scales for both MAF and switch times are logarithmic, since as usual we tested values from several orders of magnitude.

The first question to answer, before analyzing patterns in the graphs, is whether WBAS improves overall performance. The answer is a clear yes, and table 8 shows the best configurations that do so. As was done in table 5, the results are the percentage of improvement compared to Maui and Flex.

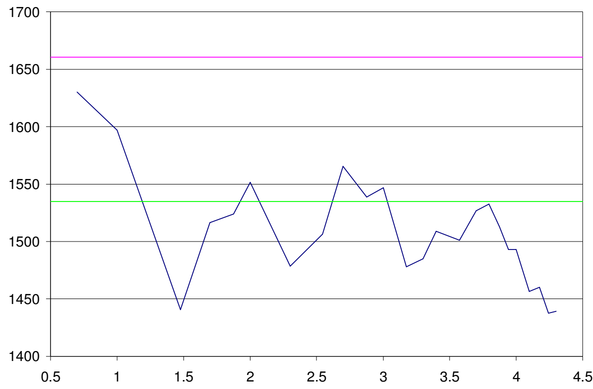
Configuration	Versus	KTH	SDSP2	BLUE
Performance Gain				
7500 MAF 1 Day	Maui	20%	10%	1%
	Flex	14%	8%	9%
7500 MAF 2 Hours	Maui	15%	7%	8%
	Flex	9%	6%	15%
750 MAF 1 Day	Maui	17%	11%	0%
	Flex	11%	10%	7%
Stability Gain				
7500 MAF 1 Day	Maui	60%	32%	14%
	Flex	-10%	8%	11%
7500 MAF 2 Hours	Maui	28%	22%	17%
	Flex	-10%	-5%	14%
750 MAF 1 Day	Maui	62%	31%	5%
	Flex	-5%	7%	2%

Table 8: WBAS Best Parameter Configurations and Overall Improvements over Flex and Maui

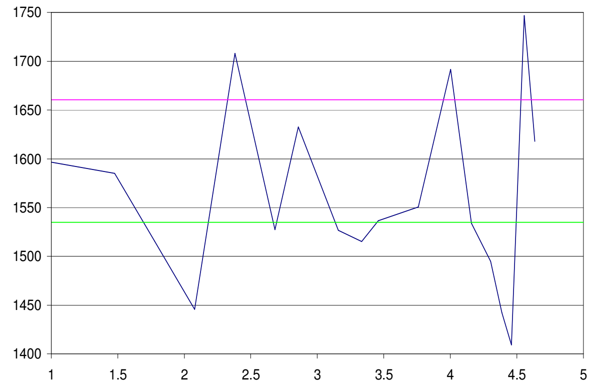
The performance gain is measured by the average wait time, and the stability gain is measured by the standard deviation of monthly wait times.

The average performance gain of the first two combinations over Maui is 10%, and their average stability gains are 35% and 22%. The third combination's average improvement over Maui is 9%, and its average stability gain is 33%. These averages are similar to the improvements achieved using PBAS, with the distinction that PBAS achieved much better gains for the BLUE log, while WBAS achieved better gains for the KTH log. The difference between the KTH and BLUE logs that is responsible for this result is unknown at this time. The average improvements over Flex are similar, except for the much smaller stability gains, which also appeared in PBAS and are caused by the fact that Flex is more stable than Maui.

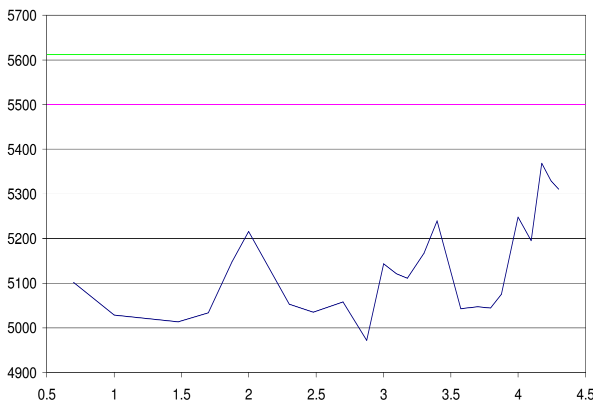
Another difference between PBAS and WBAS, made obvious by visually comparing figure 2 and figure 5, is that the performance of WBAS is less sensitive to parameter changes. This is not to say that the WBAS results are "well-behaved": we still cannot see continuous and predictable lines, and the best way to know the results for a given parameter combination is to run it in a simulation. But there is still an improvement compared to PBAS: For example, changing a MAF of 5000 by several hundreds has a moderate effect, and performance changes are usually visible only when crossing the borders to a higher or lower order of magnitude. The switching time frame is more sensitive than that, but we can still observe some common features in the three logs' graphs.



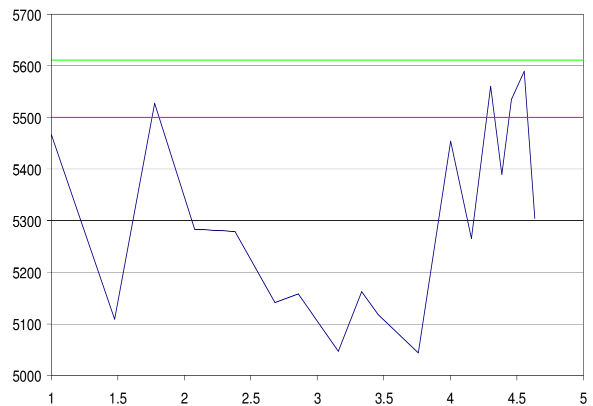
BLUE



BLUE

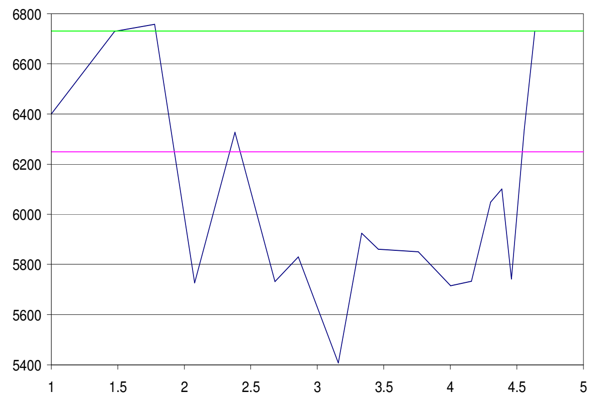


SDSP2



SDSP2

KTH



KTH

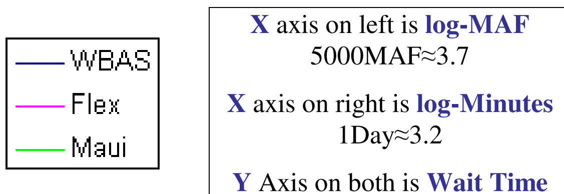


Figure 5: WBAS Results by Time Frame and MAF

Since both PBAS and WBAS show the same average overall performance, the selection between them can be based on secondary considerations. Most of these are in favor of WBAS: Its parameters are more robust to minor errors, it is simpler to implement since it doesn't require simulating both schedulers in memory at runtime, and it is slightly more efficient since only one scheduler needs to run at any given time.

A final word is due about the algorithms' performance. The runtime of adaptive algorithms is the sum of runtimes of the candidate algorithms, plus the time required to synchronize between them. This means that the performance of an adaptive scheduler is the same, in the Big-O sense, as that of its slowest candidate. In practice, both algorithms run an entire log simulation in several dozen seconds on a strong PC, which equals millisecond-range time per scheduling event. Moreover, switching events are done asynchronous to user requests, so users experience the same service times that Flex and Maui provide alone.

5. Conclusions

This paper makes three novel contributions. First, it identifies a practical problem – the per-month performance gap. The problem is shared by all the logs we examined, has a dire affect on a system's usability, and cannot be gracefully handled once it starts. Since our study focuses on the most widely used schedulers today, deployed on several platforms, we suspect that this problem is widespread and often ignored.

Second, we propose a practical solution, in the form of adaptability. The combination of the different approaches of Maui and Flex, with the recommended parameter sets that we found, significantly improves the stability and predictability of the tested systems. The average improvement for our dataset was 35% using the recommended configurations.

Moreover, the new algorithms enable a 10% gain in overall performance over existing algorithms, which makes them useful for any large parallel computer. Compared to other alternatives for boosting performance, improving the scheduler is easy: It is a software-only, relatively independent module of the computer; implementing the algorithms described here requires writing several thousand lines of code.

What adaptive algorithms don't explain is why they do or don't work. Our results provide many hints about the temporal structure of parallel workloads, and the factors that dominate them. The resulting picture is a complex and highly chaotic one, yet it is obvious that future research to further understand it is required, and can yield practical benefits.

The third contribution of this paper is the methodology, for deriving an adaptive algorithm to solve a given problem. Much of the process done in this study – defining the candidate algorithms set, tuning in to find the best parameter combinations, trying both performance- and input-based switching criteria – is applicable to many other problems of similar nature.

7. Acknowledgements

This research was supported in part by the Israel Science Foundation (grant no. 167/03).

The logs used in this paper are freely available in the Parallel Workloads Archive [9]. We would like to thank the following people for graciously providing the workloads, as well as helping with background information and interpretation: Travis Earheart and Nancy Wilkins-Diehr for the SDSC Blue Horizon, Lars Malinowsky for the KTH log, and Victor Hazlewood for the SDSC SP2 log.

8. References

[1] B. Bode, D. Halstead, R. Kendall, Z. Lei and D. Jackson, *The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters*. Proceedings of the 4th Annual Linux Showcase and Conference, October 2000, Atlanta, USA.

[2] A. B. Downey and D. G. Feitelson, *The Elusive Goal of Workload Characterization*. In *Perf. Eval. Rev.* **26**(4), pp. 14-29, Mar 1999.

[3] A. E. Eiben, J. E. Smith, J. D. Smith, *Introduction to Evolutionary Computing*. Natural Computing Series, Springer-Verlag, 2003.

[4] D. Jackson, Q. Snell, M. J. Clement, *Core algorithms of the Maui scheduler*. In JSSPP, pages 87–102, 2001.

[5] A. W. Mu'alem and D. G. Feitelson, *Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*. *IEEE Trans. Parallel & Dist. Syst.* **12(6)**, pp. 529-543, 2001.

[6] J. Skovira, W. Chan, H. Zhoi, and D. Lifka, *The EASY – LoadLeveler API project*. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 41-47, Springer Verlag '96. LNCS vol. 1162.

[7] D. Talby and D. G. Feitelson, *Supporting priorities and improving utilization of the IBM SP2 scheduler using slack-based backfilling*. In 13th *Intl. Parallel Processing Symp.*, pp. 513-517, Apr 1999.

[8] D. Talby, D. G. Feitelson, and A. Raveh, *Comparing logs and models of parallel workloads using the Co-Plot method*. In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (Eds.), pp. 43-66, Springer-Verlag, 1999. Lecture Notes in Computer Science Vol. 1659.

[9] The Parallel Workloads Archive, *http://www.cs.huji.ac.il/labs/parallel/workload*

[10] D. Tsafrir and D. G. Feitelson, *Workload flurries*. TR 2003-85, School of Computer Science and Engineering, Hebrew University of Jerusalem, Nov 2003.