On the Feasibility of Dynamic Rescheduling on the Intel Distributed Computing Platform

Zhuoyao Zhang[*] Linh T.X. Phan[*] Godfrey Tan[δ]
Saumya Jain[*] Harrison Duong[*] Boon Thau Loo[*] Insup Lee[*]
[*]University of Pennsylvania [δ]Intel Corporation

ABSTRACT

This paper examines the feasibility of dynamic rescheduling techniques for effectively utilizing compute resources within a data center. Our work is motivated by practical concerns of Intel's *NetBatch* system, an Internet-scale data center based distributed computing platform developed by Intel Corporation for massively parallel chip simulations within the company. NetBatch has been operational for many years, and currently is deployed live on tens of thousands of machines that are globally distributed at various data centers. We perform an analysis of job execution traces obtained over a one year period collected from tens of thousands of NetBatch machines from 20 different pools. Our analysis show that we observe that the NetBatch currently does not make full use of all the resources. Specifically, the job completion time can be severely impacted due to job suspension when higher priority jobs preempt lower priority jobs. We then develop dynamic job rescheduling strategies that adaptively restart jobs to available resources elsewhere, which better utilize system resources and improve completion times. Our trace-driven evaluation results show that dynamic rescheduling enables NetBatch to significantly reduce system waste and completion time of suspended jobs.

Keywords

Distributed computing, Dynamic rescheduling, Cloud resource management, Trace-driven analysis, Intel NetBatch

1. INTRODUCTION

We present a trace-driven analysis of Intel's *NetBatch* system, an Internet-scale computing infrastructure developed by Intel Corporation for running concurrently tens of thousands of chip simulations. NetBatch has been operational for many years but has evolved from a pure job execution engine into a distributed grid computing platform [11] and now a service-oriented computing *cloud* for Intel. The goal of our analysis is to examine the effectiveness of NetBatch's scheduling middleware and study the feasibility of *dynamic rescheduling* techniques for effectively utilizing compute resources within the shared platform.

The NetBatch platform resembles what is described as a *private cloud* [2]. These are internal data centers operated by a business or an enterprise for internal use. Private clouds like NetBatch share with many similar characteristics and challenges with public cloud computing efforts, such as Amazon's EC2 [1], including deployment at Internet-scale, supporting a large group of heterogeneous concurrent applications, and supporting different quality of service guarantees for different classes of users. To put the relevance in context, the NetBatch deployment today is estimated to involve hundreds of machine clusters called *pools*, distributed globally at dozens of data centers with varying wide-area network characteristics, utilizing tens of thousands of heterogeneous multi-core compute machines. At any moment, there are thousands of concurrent jobs with varying priorities and requirements being dispatched by engineers through Intel, and in aggregate, hundreds of millions of jobs per year.

One important challenge that NetBatch faces today is the need to accommodate jobs with varying priorities and goals while keeping both the system utilization high and latency low. In today's Intel environments, there are higher and lower priority jobs, with the former being able to suspend the latter when resources become constrained.

Based on our analysis of a year-long trace data, we find that the NetBatch currently does not make full use of all the resources. The behavior of high priority jobs can significantly impact the completion time of low priority jobs and high wait time of job exists even when the overall system utilization is relatively low. We study these issues in depth and present preliminary work to improve the average job completion time and throughput of the entire system.

Specifically, our contributions are as follows:

Trace-driven analysis: We present an analysis of job execution traces obtained over a year-long period collected from tens of thousands of machines deployed in 20 NetBatch pools from a large site. To our best knowledge, this is one of the first trace-driven efforts at empirically understanding the performance characteristics of scheduling policies within a distributed computing platform for use by a global enterprise. From the traces, we observe in many cases

- Significantly high completion time of jobs during peak times at certain pools, primarily caused by high priority jobs preempting low priority jobs, even when the computing resource utilization is only around 40% on average, and
- High wait time of jobs large due to ineffective scheduling of jobs and bursty workload conditions.

Dynamic rescheduling: Given the above observations, we explore dynamic rescheduling approaches whereby jobs are rescheduled to available resources elsewhere upon suspension or prolonged waiting within pool queues. We validate the potential benefits of the dynamic rescheduling strategies using a trace-driven simulator developed in-house at Intel. Our results are as follows:

- By rescheduling suspended jobs to other physical pools, we reduce the system waste time by more than 33% and the average completion time over suspended jobs by 50%.
- By rescheduling jobs stalled in waiting queues, we can further decrease the average completion time of all jobs, thereby improving the overall system performance.

We also find that a simple random pool selection when applied appropriately can perform quite well alluding to the possibilities of scheduling decisions made by individual jobs rather than the system as it has been done traditionally.

2. ARCHITECTURE AND MOTIVATION

We provide an architectural overview of NetBatch and highlight its performance insights gathered from traces.

2.1 NetBatch Architecture

Figure 1 shows the overall architecture of NetBatch, which is designed designed as a hierarchical system consisting of several *physical pools*, each of which consists of hundreds or thousands of multi-core machines. Intel has many sites of operation across the globe. Each typical site has many physical pools that are often located in multiple data centers. To simplify computing operations and make the geography of physical pools transparent to the users, NetBatch deploys a middleware layer called *virtual pool managers* at each site. Each virtual pool manager is configured to connect to one or more physical pools located at the same site or different sites. A virtual pool manager accepts job submissions from users at that site, and then distributes jobs to the connected physical pools according to resource availability and NetBatch configurations. Thus, NetBatch inherently supports remote execution of jobs in a manner that is transparent to the users. In practice, chip simulation jobs are also I/O intensive, and they require access to a large amount of data. Data synchronization and large data transfers are typical handled through out-of-band mechanisms and are out of scope for this paper.

Jobs submitted to the virtual pool manager are immediately queued, and will be sent to a physical pool. The default scheduling follows a round-robin fashion. Once the job arrives at a physical pool, the *physical pool manager* dispatches the job to a particular machine based on the job requirements (e.g., OS and memory), the resource availability and the priority of the job. Specifically, the physical pool manager maintains a list of available resources. When a new job is assigned to the physical pool, the pool manager searches its list to find the first *eligible machine* (i.e., which satisfies the job requirements) that is available and schedules the job there. If all eligible machines are currently not available, two different approaches will be taken according to the job's priority: (a) if there is a job currently running on an eligible machine that has lower priority than the new job, this currently running job will be suspended by the new job; b) otherwise, the new job will be queued and waiting for resources to become available in the physical pool. On the other hand, if none of the machines in the list is eligible, the physical pool manager will return the new job to the virtual pool manager, who will then try to schedule the job at the next physical pool.

2.2 Priority-based Preemption

Different engineering and business groups within Intel purchase computing capacity to meet their needs. These groups get higher priorities assigned to jobs they submit since the resources are charged to their budget. This comes in the form of a property called ownership. Ownership is defined in this paper as a business group that pays for a particular host machine. When a group "owns" the machine, it means their jobs have high priority and can preempt other jobs on that machine. This sharing agreement enables machine owners to share their owned machines with others when they are idle while being entitled to use their owned resources whenever the need arises. In addition, the priority mechanism is also used to distinguish between jobs that are highly sensitive to turn-around time (and thus high priority) vs. jobs that are less sensitive to latency although in general, users clearly would like to have their jobs completed as soon as possible.

In NetBatch, priority-based job preemption is enforced at the host level of each physical pool. When a high priority job is scheduled by the virtual pool manager for execution in one of the physical pools, if none of the eligible machines in that physical pool is available right then, the physical pool manager will try to find a low priority job running on one of those machines. Once found, the lower priority job will be *suspended* to allow the new job (with higher priority) to execute. Such a priority-based job preemption is necessary for the NetBatch environment due to the business need to periodically run a large amount of jobs in a relatively short time. When a pool is highly utilized, low priority jobs may get suspended more than once.

To illustrate the impact of higher priority jobs on lower priority ones, Figure 2 shows the CDF of suspension time for suspended jobs (in minutes) collected from a large site with 20 physical pools, for a time period of 500,000 minutes (roughly a year) of NetBatch traces. We note that 20% of all jobs are suspended for more than 1100 minutes (18 hours), and the median suspension time is 437 minutes (7.3 hours), the average suspension time is 905 minutes (15 hour). In addition, we observe a long-tailed distribution of jobs that require more than 100k minutes to complete.

The impact in some cases goes beyond the higher completion times experienced by suspended jobs and can impact the engineering productivity. For example, some classes of chip simulation work has logical notions of tasks, each of which represents a set of jobs completing a specific function. Typically, 100% or a high percentage of jobs associated with a particular task needs to complete before the task result (combined from the results of those jobs) can be useful. Often when one or more of those low priority jobs cannot complete in a timely fashion, engineers lose productivity and/or system resources are wasted since the same task execution needs to be manually repeated at a different time.

2.3 Motivation for Rescheduling

Figure 4 shows the NetBatch utilization (black dotted line) as a percentage of the total number of available cores, and the number of suspended jobs (blue solid line). The data are gathered over a period of a whole year of NetBatch traces. We sampled the number of suspended jobs in the system and the system utilization every minute and aggregated them to get an average number based on a 100 minutes interval.

We make the following observations. First, the overall system utilization averages around 40%, and is typically in the range of 20%-60%. However, during the same period, higher priority jobs can preempt lower priority jobs, leading to a large number of suspended jobs and hence a poor utilization of resources. Second, higher priority jobs tend to be bursty in nature, as demonstrate in the trace, where we observe that job suspension can spike suddenly due to the arrival of a large number of higher priority jobs and last from several hours to a week.

Third, we observe that suspension may arise in cases even when the system is not overloaded (at 40-60% utilization). The typical reason is that latency sensitive jobs with high priority are usually configured to only run in specific sets of physical pools so that desired results can be achieved. During periods where there are bursts of high priority jobs being dispatched, those pools are quickly overwhelmed and lots of low priority jobs are suspended. However, during the same time period, other pools may be barely utilized.

The above observations leads to our approach of dynamic *rescheduling* of suspended jobs at different pools which may have more availability of resources required by the suspended jobs. Similarly, we also apply rescheduling techniques on jobs that have been waiting in queue for a long time. By rescheduling jobs that are stalled at a particular pool to an alternate pool, we can make better utilization of the overall resources so as to improve the efficiency of the system.

One potential shortcoming of our rescheduling-based approach for suspended jobs is the likelihood of wasted work due to the rescheduling of a suspended job in a different pool. One valid question to ask is why migration (e.g., as used in Condor [4]) or VM

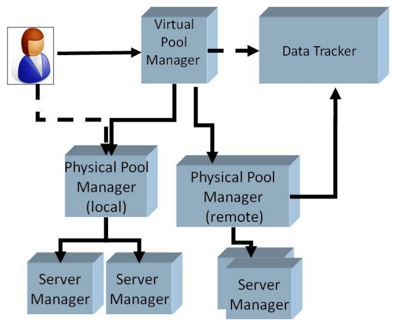

Figure 1: NetBatch Architectural Overview.

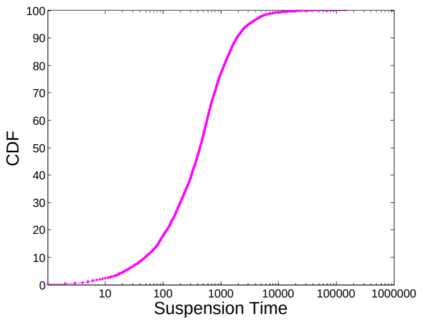

Figure 2: CDF of job suspension time.

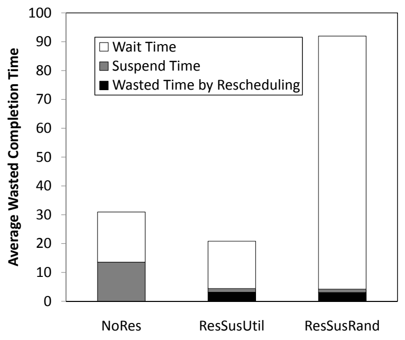

Figure 3: Average wasted completion time (in minutes) under normal load.

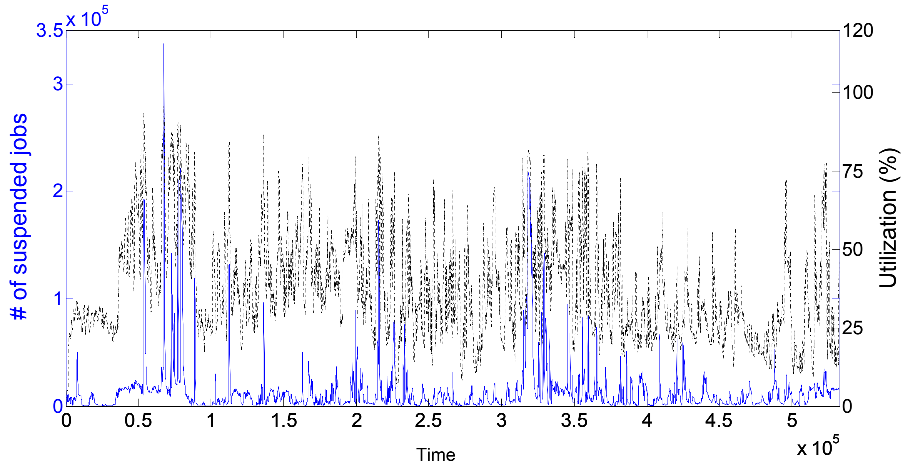

Figure 4: Suspension (# number of jobs) and Utilization (%) over a one year period.

migration approaches (e.g., as used in VMWare [8]) are not used by NetBatch. We think that the VM migration approach works best for some environments and not as well in other environments like Intel's. This is because running chip simulation workloads (like the ones being executed inside Intel's data centers) on visualized hosts often lead to performance overhead between 10% to 20% as indicated in recent experiments [9].

More importantly, our rescheduling approach for waiting jobs complements to either a restart strategy or VM migration method used to restart or migrate suspended jobs in alternate pools. In fact, as we show in Section 3, a simple rescheduling scheme applicable for both suspended and waiting jobs can be very effective and its implementation easily achieved in existing environment.

3. RESCHEDULING STRATEGIES

In this section, we perform a trace-driven simulation analysis to study the potential benefits of various dynamic rescheduling strategies in NetBatch.

3.1 Experimental Setup

Our evaluation is carried out using a hybrid event-based and agent-based simulator called *ASCA* (for Agent-based Simulator for Compute Allocation), developed at Intel [12]. ASCA models the operational capability and semantics of various fine-grained components of NetBatch such as sites, pools, queues, job requirements and priorities, virtual and physical pool managers, round-robin physical pool scheduling. It samples at each minute the current states of all NetBatch components (e.g., idle, busy) and jobs (e.g., running, suspended, waiting), as well as the jobs' resource usages, and outputs the results as logs for post-analysis. Reference [12] describes the implementation of this simulator in greater detail and provides detailed analysis results to demonstrate that the simulator achieves the performance characteristics of the actual NetBatch deployment in terms of utilization and job completion times [12]. Here, we incorporate with the existing ASCA simulator various dynamic rescheduling strategies for our performance evaluation.

The simulator takes as inputs NetBatch trace as described in Section 2.2. The NetBatch trace describes the complete information of the jobs submitted to the site (from several sites across the globe) during the period, including computing resource and memory requirements, submission time and priority. The simulator is configured to emulate 20 physical pools, each of which contains hundreds to tens of thousands of machines with varying CPU speed and memory.

To better understand the effect of our dynamic rescheduling approaches, we focus on the (total of 248000) jobs that are submitted During a one week busy period in the trace (i.e., with submission time between 76000 and 86080 minutes as shown in Figure 4). This particular period of the trace captures a typical burst of high-priority jobs and as a result, a burst of job suspension. We execute these jobs on the ASCA simulator until all 248000 jobs are completed, and gather experimental results based on the following job and system-wide metrics. All time-based metrics are calculated in minutes.

- The *Suspend Rate*, which is the fraction of all jobs submitted to NetBatch that have been suspended at least once during the job lifetime.
- The average completion time *AvgCT*. This is further broken into two subcategories, where we consider all jobs and only jobs that have been suspended at least once.
- The average suspend time *AvgST* of jobs that have been suspended at least once.
- The average wasted completion time *AvgWCT* of jobs, where wasted time for a job is defined as the average duration in which a job exists in NetBatch, but do not make progress towards job completion. This waste metric include the following three components: (c1) *Wait Time*, i.e., time duration in which jobs are waiting either at the virtual or physical pool level; (c2) *Suspend Time*, i.e., the amount of time the job is suspended in a suspended queue; and (c3) *Wasted Time by Rescheduling*, i.e., any completion time wasted as a result of restarting a job.

  To compute this average, we first determine the total wasted completion time for all jobs that are executed during the duration of the trace, and then divide by the number of jobs. By minimizing the wasted completion time of jobs, we effectively increase the overall goodput of the system.

We note that we also conducted similar experiments by fixing a specific time window instead of number of jobs during each simulation run. The results under a fixed time window are similar to those under a fixed number of jobs (the identical set) and thus we only include the results under the latter.

3.2 Rescheduling of Suspended Jobs

We first examine the benefits of dynamic rescheduling of suspended jobs, using the current default round-robin scheduler used by NetBatch. In this set of experiments, whenever a currently running job on a machine is suspended by a newly arrived job with higher priority, it could be restarted (from the beginning) at a different pool. As soon as a rescheduling decision is made, the job will be sent to the alternate pool directly for re-execution. If the alternate pool happens to be heavily loaded and cannot execute the job immediately, the job will be stored in the alternate pool's waiting queue until compute resources become available.

We evaluate the following rescheduling schemes which differ in their approaches to select the alternate pool for the suspended job to restart:

- *ResSusUtil*: The pool with the lowest utilization among all candidate pools is designated as the alternate pool.
- *ResSusRand*: A randomly selected pool among all candidate pools is designated as the alternate pool.

3.2.1 Round-robin Initial Scheduler

One important consideration in evaluating rescheduling schemes is how they interact with initial scheduling schemes. As stated in Section 2.1, NetBatch's virtual pool managers use a round-robin scheduler to distribute jobs across candidate pools. To disambiguate from rescheduling schemes, we call the scheduler at each virtual pool manager *initial scheduler*. We consider two initial scheduling schemes: i) *round-robin scheduler*, which distributes jobs across candidate pools in a sequential order and ii) *utilization-based scheduler*, which distributes jobs to the pool in the order of decreasing utilization.

	Suspend rate	**AvgCT**		**AvgST**	**AvgWCT**
		Suspend	**All**		
NoRes	1.14%	2498.7	569.8	1189.1	31.0
ResSusUtil	1.56%	1265.4	560.0	82.2	20.8
ResSusRand	1.52%	7580.7	638.7	80.7	91.9

Table 1: Performance under normal load scenario

Normal Load Scenario

Table 1 summarizes the results over the aforementioned one week period. We make the following observations from the table. First, *ResSusUtil* outperforms *NoRes* for all metrics. For instance, there is a 50% reduction (from 2498.7 to 1265.4 minutes) of *AvgCT* over suspended jobs when rescheduling is performed. Moreover, the average wasted completion time reduced by 33% (from 31.0 minutes to 20.8 minutes) when rescheduling is performed. Second, we observe that dynamic rescheduling may backfire if the alternate pool is randomly selected, as shown by the performance degradation of *ResSusRand*. The reason is a poor selection such as choosing a pool that is already has a lot of waiting jobs leading to an even longer wait time for the restarted job before it can run in the alternate pool.

To further drill down on the throughput improvements due to rescheduling, Figure 3 details the components constituting the average wasted completion time of the system for all three strategies. *NoRes* does not have wasted time by rescheduling (since it never reschedules job) but incurs a larger amount of suspend time. In contrast, by sending the job to other physical pools, *ResSusUtil* greatly reduce the suspend time. Moreover, the benefits of rescheduling (as reflected by the reduction in suspend time) clearly outweigh its costs (wasted time by rescheduling), resulting in overall better resource utilization. As expected, *ResSusRand* has a large number of wait time which results in the worst performance overall, indicated that dynamic rescheduling is most beneficial when the alternate pool is chosen carefully based on system load.

High Load Scenario

	Suspend rate	**AvgCT**		**AvgST**	**AvgWCT**
		Suspend	**All**		
NoRes	1.26%	5846.1	988.7	4402.4	450.1
ResSusUtil	1.83%	1475.1	962.2	86.2	423.9
ResSusRand	1.60%	6485	1180	73.2	636.3

Table 2: Performance under high load scenario.

To explore the performance of dynamic rescheduling under high load scenarios, Table 2 shows a similar experiment carried out under a high degree of resource contention and job preemption. To emulate a high load situation, we reduce the number of compute cores available to each pool by half while keeping the submitted job trace unchanged. As expected, under high load, *AvgCT* for all jobs (988.7 minutes) is almost doubled compared to that of the previously described normal scenario (569.8 minutes) due to the increased system utilization.

Interestingly, we note that the benefits of dynamic rescheduling on suspended jobs are further enhanced under the high load situation. As shown in the table, *AvgCT* of suspended jobs is reduced by 75% (i.e., from 5846.1 minutes to 1475.1 minutes). This is due to the fact that under high-load, a low priority job usually experiences a much longer suspend time in the presence of high priority jobs, and hence suspended jobs have a greater chance of completing earlier by being rescheduled into another pool with available resources. In the worst case, if all alternate pools are even more utilized than the current pool, *ResSusUtil* will simply retain the suspended job in its current pool, ensuring that rescheduling will not negatively impact system performance.

High Suspension Scenario

Since the rescheduling of suspended jobs mainly improve the completion time of suspended jobs, a higher fraction of suspended jobs naturally leads to a larger impact on the average completion time of all jobs. In this particular trace, around 1.14% of all jobs have been suspended. While the suspension rate increases to around 1.6% when jobs are dynamic rescheduled (due to a more aggressive use of system resources), it is still relatively low and hence, does not significantly affect the *AvgCT* of all jobs. To investigate the performance of rescheduling under high suspend rate, we created a job trace that result in a suspend rate of 14%. Here, there is a more significant reduction of 7% in *AvgCT* for all jobs, and an equally high reduction of 44% in *AvgCT* of suspended jobs.

3.2.2 Utilization-based Initial Scheduler

	Suspend rate	**AvgCT**		**AvgST**	**AvgWCT**
		Suspend	**All**		
NoRes	1.50%	5936.0	994.2	4916	456.6
ResSusUtil	1.72%	1466.9	946.2	84.5	407.6
ResSusRand	1.62%	7979.9	1229.9	72.3	686.8

Table 3: Performance with utilization-based initial scheduling

The advantage of the round robin initial scheduler adopted by the NetBatch system is its simplicity in design and implementation. The virtual pool managers also need not maintain any statistics of their physical pools. However, by ignoring the varying resource availability across various physical pools, round robin may result in a long job waiting time even though there are resources available. In this section, we run simulations using the utilization-based initial scheduler, under which each job entering a virtual pool manager is scheduled to the physical pool that currently has the lowest utilization. As before, if a job becomes suspended during execution, a dynamic rescheduling approach will be applied as before.

Table 3 shows our evaluation results for the three strategies *NoRes*, *ResSusUtil*, and *ResSusRand* with the additional use of utilization-based initial scheduling. We present the results under high load scenario, because this load reflects more closer to the current Intel environments which have increased utilization since the experimental trace was collected. The same experiments under normal load scenario are also carried out and the observations are similar.

We can find that dynamic rescheduling *ResSusUtil* still works with the utilization-based initial scheduler, achieving 75% reduction in the average completion time for suspended jobs (from 5936.0 to 1466.9 minutes) and a 11% reduction in the average wasted completion time (from 456.6 to 407.6 minutes). Another observation is that when no rescheduling method is used, utilization-based initial scheduling leads to a higher suspend rate compared to that under RR Initial Scheduler (from 1.26% to 1.50%). This is possibly due to the fact that the utilization-based initial scheduler tends to send more jobs to larger pools which leas to more suspension when high priority jobs burst in those pools.

It is worth noting, however, that exact implementation of the utilization-based scheduling requires the virtual pool manager to know the current situation in every physical pool at any time, which can be impractical in reality given the unavoidable propagation latency between different pools in a geographically distributed system.

3.3 Rescheduling of Waiting Jobs

As evident in earlier experiments, wait time is often the largest component of job completion time. We observe situations whereby jobs assigned to heavily utilized pools are stuck for long periods of time in the wait queue. This is particularly exacerbated by NetBatch's use of the round robin scheduler. In these scenarios, rescheduling a suspended job does not improve completion time, since the waiting jobs stuck in queues are not considered suspended. In this section, we apply the rescheduling approaches to reschedule not only suspended jobs but also jobs waiting in a queue for longer than a specific threshold.

- *ResSusWaitUtil*: Reschedule each waiting job that have passed the threshold at the pool with lowest utilization.
- *ResSusWaitRandom*: Reschedule each waiting job that have passed the threshold at a random pool.

We set the waiting time threshold to be 30 minutes, which is about twice the expected average waiting time in the original system. Note that both schemes mentioned above also reschedule suspended jobs as described in Section 3.2. We use the same high load scenario and job trace used in Section 3.2.1.

3.3.1 Round-robin Initial Scheduler

	Suspend rate	**AvgCT**		**AvgST**	**AvgWCT**
		Suspend	**All**		
NoRes	1.26%	5846.1	988.7	4402.4	450.1
ResSusWaitUtil	1.46%	1224.3	951.4	72.7	414.2
ResSusWaitRand	1.50%	1417	954.7	62.3	417.6

Table 4: Performance with round robin initial scheduling

As shown in Table 4, the combined rescheduling approach (*ResSusWaitUtil*) additionally improves upon the previous rescheduling approach (*ResSusUtil* in Table 2). Specifically, by increasing job mobility and avoiding long waiting time of jobs, the average completion time of suspended jobs is reduced by 79% (from 5846.1 to 1224.3 minutes). Moreover, the average wasted completion time of the entire system is reduced by by 8% (from 450.1 to 414.2 minutes).

Interestingly, we observe that unlike our earlier observed performance of *ResSusRand*, the random strategy here *ResSusWaitRand* performs *almost as well* as a utilization-based approach. The reason is that although the random strategy may make poor choices of the alternate pool, the rescheduled job can gain multiple second chances to select another pool and is restarted again once it gets stuck in a queue.

3.3.2 Utilization-based Initial Scheduler

	Suspend rate	**AvgCT**		**AvgST**	**AvgWCT**
		Suspend	**All**		
NoRes	1.50%	5936	994.2	4916	456.6
ResSusWaitUtil	1.74%	1467.2	937.9	84.5	402.0
ResSusWaitRand	1.71%	1603.1	935.7	100.6s	399.7

Table 5: Performance with utilization-based initial scheduling

Our final experiment uses the combined rescheduling approach together with utilization based initial scheduling. As shown in Table 5, the results show again that the random strategy *ResSusWaitRand* performs well by allowing the rescheduled job select another pool restarted again once it gets stuck.

Such observation inspires the possibility of a simple implementation of the system without any effort of tracking the physical pools' statistics. More importantly, *ResSusWaitRand* can be implemented without any coordination or changes to the system's scheduler. Each job can simply keeps a timer to keep track of how long it has been in a queue and when a threshold is reached, dequeues itself from the queue and resubmits to a randomly selected candidate pool. The fact that no system metrics are needed to make a rescheduling decision enables the rescheduling decision to be made solely by the waiting job. However, the advantage of design simplicity does come at a cost of much more frequent restart operations. In certain environments, frequent restarts may not be desirable since each restart operation may include time consuming operations like transferring large amount of data and job binaries to the alternate pool.

The above evaluations demonstrate the effectiveness of rescheduling in unpredictable workload environments. While it is often difficult to find an optimal initial scheduling policy for such a dynamic and unpredictable environment, a simple rescheduling – which can be easily implemented – complements the initial scheduling by providing a second chance for jobs to adjust to the most recent system state in case the initial scheduling decision is not optimal.

3.4 Summary of Results

We summarize our experimental results as follows. First, our results demonstrate that rescheduling of suspended jobs can greatly improve the average completion time of suspended jobs without hurting the completion time of other jobs, and also increase system throughput by reducing average wasted completion time. These improvements are also apparent under high load scenarios, particularly in cases where suspension rates are high. The effectiveness of the rescheduling approach are also compatible with different initial schedulers.

Second, by additionally rescheduling waiting jobs, we can further improves average completion times overall jobs. Specifically, when adopting a simple random scheme in our combined rescheduling approach, the results are surprisingly better than rescheduling only suspended jobs and are close to that of the utilization-based strategy. This opens up the possibility of simpler scheduling framework without help from central scheduler, i.e., a job can make decision to restart by itself in certain cases.

4. RELATED WORK

NetBatch can be viewed as a real-world deployment of an Internet-scale *grid computing* [5] infrastructure. Condor [4, 7] batch processing system provides a transparent middle-ware for executing jobs on shared computational resources. To our best knowledge, NetBatch's scale of deployment is larger than any existing Condor pools, both in terms of the number of jobs and machines. NetBatch's scale is closer to that of p2p computational platforms such as SETI@Home, with much more stringent requirements on job completion times.

In addition to our use of job rescheduling, popular techniques for resource management that are potentially applicable to NetBatch in future include the use of virtual machine migration [4] and redundant executions [6, 13] to trade-off utilization and completion time. Our approach is orthogonal to these techniques, and our choice of rescheduling is largely driven by current practical concerns outlined in Section 2.1. In addition, our recent internal evaluation of NetBatch typical workloads have shown that although VM performance have improved significantly over the years [8], it has yet to go down to a low single digit before it becomes a viable option from both performance and cost perspective for the NetBatch environment, where jobs tend to be memory and I/O intensive. Nevertheless, it is interesting future work to explore ways to integrate dynamic rescheduling approaches with VMs within the context of the NetBatch platform.

We also note recent work around adaptive job execution techniques after the job is already executed [10]. Unlike our work, Shan *et al.* focus on opportunistic job migration when a "better" resource is discovered and do not discuss about suspended jobs. Our NetBatch workloads and scale will enable us to study these issues in depth in a realistic yet challenging setting.

5. CONCLUSION AND FUTURE WORK

In this paper, we perform a trace-driven analysis of Intel's NetBatch system, an Internet-scale data center based distributed computing platform developed by Intel Corporation for massive chip simulation. We propose a dynamic rescheduling approach which makes better utilization of the resources by restarting jobs to other available pool.

Our initial results are promising: by performing rescheduling on suspended jobs, the average completion time for previously suspended jobs can be reduced by around 50% and we are able to make better use of the system resource by reducing wasted throughput by around 33%. Under high load, the improvements to the average completion time of suspended jobs are even more significant, achieving a reduction of 75%. Such reduction is further increased to 79% with the additional rescheduling of jobs in wait queues.

We are currently exploring the use of more sophisticated rescheduling strategies that combine job duplication techniques and inter-site rescheduling, with the use of multiple metrics (e.g., utilization, queue lengths, prediction of job completion times within a pool) in combination for making rescheduling decisions. We are also exploring improvements to the simulator to incorporate network delays and other rescheduling associated overheads. Finally, we plan to validate our trace-driven simulation results on the NetBatch platform itself.

While our study is motivated by the practical needs of Intel's NetBatch, our dynamic rescheduling strategies have broader applicability, particularly in the context of scheduling cloud computing programs in a heterogeneous environment. Beyond the NetBatch system, our longer-term goal is to develop a holistic understanding of the resource management challenges of similar computational clouds at the global scale. Similar to NetBatch's need to accommodate multiple classes of jobs, public clouds will soon need to design more effective scheduling solutions that can accommodate jobs of different Service Level Agreements (SLAs) and thus pricing. To this end, we plan to generalize the dynamic rescheduling techniques presented in this paper to other cloud computing middleware [3], in order to perform priority-based allocation of cloud resources in heterogeneous environments.

6. REFERENCES

[1] Amazon Elastic Compute Cloud, Virtual Grid Computing. `http://aws.amazon.com/ec2`.

[2] M. Armbrust, A. Fox, R. Griffith, and et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, UC Berkeley, Feb. 2009.

[3] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *osdi*, 2004.

[4] M. L. et. al. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, UW-Madison, 1997.

[5] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[6] G. Koole and R. Righter. Resource allocation in grid computing. *J. of Scheduling*, 11(3):163–173, 2008.

[7] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *ICDCS*, 1988.

[8] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX*, 2005.

[9] S. Sammanna, T. Tang, and V. Lal. Server Virtualization using Xen based VMM. In *Intel IT Technical Leadership Conference*, 2008.

[10] H. Shan, L. Oliker, and R. Biswas. Job superscheduler architecture and performance in computational grid environments. *SC Conference*, 2003.

[11] Srinivas Nimmagadda et al. High-End Workstation Compute Farms Using Windows NT. In *3rd USENIX Windows NT Symposium*, 1999.

[12] G. Tan, D. Duzevik, E. Bunch, T. Ashburn, E. Wynn, and T. Witham. Agent-based Simulator for Compute Resource Allocation. In *Intel IT Technical Leadership Conference*, 2008.

[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.