AN OPPORTUNITY COST APPROACH FOR JOB ASSIGNMENT IN A SCALABLE COMPUTING CLUSTER[1]

Yair AMIR[2], Baruch AWERBUCH[2], Amnon BARAK[3], R. Sean BORGSTROM[2], and Arie KEREN[3]

Abstract.

A new method is presented for job assignment to and reassignment between machines in a computing cluster. Our method is based on a theoretical framework that has been experimentally tested and shown to be useful in practice. This "opportunity cost" method converts the usage of several heterogeneous resources in a machine to a single homogeneous "cost". Assignment and reassignment is then performed based on that cost. This is in contrast to previous methods for job assignment and reassignment, which treat each resource as an independent entity with its own constraints. These previous methods were intrinsically *ad hoc*, as there was no clean way to balance one resource against another.

1. INTRODUCTION

The more powerful a cluster of workstations is, the more important it is to use its resources wisely. A poor job assignment strategy can result in heavily unbalanced loads and thrashing machines, which cripples the cluster's computational power. Resources can be used more efficiently if the cluster can migrate jobs – moving them transparently from one machine to another. However, even systems that can reassign jobs can still benefit from a carefully-chosen assignment strategy.

Job migration is attractive because the arrival rate and resource demands of incoming jobs are unpredictable. In light of this unpredictability, jobs will sometimes be assigned to a non-optimal machine, and migration gives the system a second (or third, etc.) chance to fix such a mistake. It is intuitively clear that the ability to migrate jobs could lead to better performance – that is, faster completion times for the average job. Unless it is known where a job *should* be at any given time, however, the reassignment strategy could also make mistakes. The Mosix [1, 2] system, for example, allows this kind of transparent job migration in the Unix environment.

Determining the optimal location for a job is a complicated problem. The most important complication is that the resources available on a cluster of workstations are heterogeneous. In effect, the costs for memory, CPU, process communication and so forth are *incomparable*. They are not even measured in the same units: communication resources are measured in terms of bandwidth, memory in terms of space, and CPU in terms of cycles. The natural greedy strategy, balancing the resources across all of the machines, is not even well defined.

In this paper, we present a new job assignment strategy based on "economic" principles and competitive analysis. This strategy enables us to manage heterogeneous resources in a near-optimal fashion. The key idea of this strategy is to convert the total usage of several heterogeneous resources, such as memory and CPU, into a single homogeneous "cost." Jobs are then assigned to the machine where they have the lowest cost.

This economic strategy provides a unified algorithmic framework for allocation of computation, communication, memory and I/O resources. It allows the development of near-optimal online algorithms for allocating and sharing these resources.

Our strategy guarantees near-optimal end-to-end performance for the overall system on each single instance of job generation and resource availability. This is accomplished using online algorithms that know nothing about the future, assume no correlation between past and future, and are only aware of the state. In spite of this, one can rigorously prove that their performance will always be comparable to that of the optimal prescient strategy.

This work shows that the unified opportunity cost approach offers good performance in practice. First, we performed tests using a simulated cluster and a "typical" series of incoming jobs. Our method, with and without reassignments, was compared against the methods of PVM, a dominant static job assignment

[1] This research is supported, in part, by the Defense Advanced Research Projects Agency (DARPA) under grant F306029610293 to Johns Hopkins University.

[2] {yairamir, baruch, rsean} @cs.jhu.edu
Department of Computer Science, The Johns Hopkins University, Baltimore MD 21218.

[3] {amnon, ariek} @cs.huji.ac.il
Institute of Computer Science, The Hebrew University of Jerusalem 91904, Israel.

strategy, and Mosix, one of the more successful system that support transparent process migration. Each method was given an identical stream of jobs. Over 3,000 executions of this Java-based simulation were performed, each representing at least 10,000 simulated seconds. When no reassignments were allowed, our method was shown to be a dramatic improvement over PVM. When reassignments *were* allowed, our method was substantially better than that of the highly tuned, but ad hoc, Mosix strategy.

A second series of tests was performed on a real system, to validate this simulation. This system consisted of a collection of Pentium 133, Pentium Pro 200 and Pentium II machines with different memory capacity, connected by Fast Ethernet, running BSD/OS [3]. The physical cluster and the simulated cluster were slightly different, but the proportional performance of the various strategies was very close to that of the Java simulation. This indicates that the simulation appropriately reflects events on a real system.

In Section 2, we will discuss the model we used and our assumptions. In Sections 3 and 4, we will describe our algorithm and the theoretical guarantees that come with it. In Section 5, we will show our experimental evidence that this strategy is useful in practice. Section 6 concludes the paper. For additional information about this research, consult the following web site: http://www.cnds.jhu.edu/projects/metacomputing.

2. THE MODEL

The goal of this work is to improve performance in a cluster of n machines, where machine i has a CPU resource of speed $r_c(i)$ and a memory resource of size $r_m(i)$. We will abstract out all other resources associated with a machine, although our framework can be extended to handle additional resources.

There is a sequence of arriving jobs that must be assigned to these machines. Each job is defined by three parameters:

- Its arrival time, $a(j)$,
- The number of CPU seconds it requires, $t(j)$, and
- The amount of memory it requires, $m(j)$.

We assume that $m(j)$ is known when a job arrives, but $t(j)$ is not. A job must be assigned to a machine immediately upon its arrival, and may or may not be able to move to another machine later.

Let $J(t,i)$ be the set of jobs in machine i at time t. Then the CPU load and the memory load of machine i at time t are defined by:

$l_c(t,i) = |J(t,i)|$,

and

$l_m(t,i) = \sum_{j \in J(t,i)} m(j)$ respectively.

We will assume that when a machine runs out of main memory, it is slowed down by a multiplicative factor of τ, due to disk paging. The *effective CPU load* of machine i at time t, $L(t,i)$, is therefore:

$$L(t,i) = \begin{cases} l_c(t,i) & \text{if } l_m(t,i) \le r_m(i), \\ \text{and } l_c(t,i) * \tau & \text{otherwise.} \end{cases}$$

For simplicity, we will also assume that all machines schedule jobs *fairly*. That is, at time t, each job on machine i will receive $1/L(t,i)$ of the CPU resource. A job's completion time, $c(j)$, therefore satisfies the following equation:

$\int_{a(j)}^{c(j)} \frac{r_c(i)}{L(t,i)} = t(j)$, where i is the machine the job is on at any given time.

The *slowdown* of a job is equal to $\frac{c(j)-a(j)}{t(j)}$.

Our goal in this paper is to develop a method for job assignment and/or reassignment that will minimize the average slowdown over all jobs.

3. THEORETICAL BACKGROUND

We will evaluate the effectiveness of our (online) algorithms by their *competitive ratio*, measured against the performance of an optimal offline algorithm. An online algorithm ALG is c-competitive if for any input sequence I, $\mathrm{ALG}(I) \le c\ \mathrm{OPT}(I) + \alpha$, where OPT is the optimal offline algorithm and α is a constant.

3.1 INTRODUCTION AND DEFINITIONS

The theoretical part of this paper will focus on how to minimize the maximum usage of the various resources on a system – in other words, the best way to balance a system's load. One such algorithm, described in [4], proves useful in practice even when our goal is to minimize the average slowdown instead, which corresponds to minimizing the sum of the squares of the loads.

In preparation for a discussion of this algorithm, Assign-U, we will examine this minimization problem with three different machine models and two different kinds of jobs. The three machine models are:

1. **Identical Machines.** All of the machines are identical, and the speed of a job on a given machine is determined only by the machine's load.
2. **Related Machines.** The machines are identical except that some of them have different speeds – in the model above, they have different r_c values,

and the memory associated with these machines is ignored.

3. **Unrelated Machines.** Many different factors can influence the effective load of the machine and the completion times of jobs running there. These factors are *known*.

The two possible kinds of jobs are:

1. **Permanent Jobs.** Once a job arrives, it executes forever without leaving the system.
2. **Temporary Jobs.** Each job leaves the system when it has received a certain amount of CPU time.

We will also examine a related problem, called the *online routing* problem.

3.2 IDENTICAL AND RELATED MACHINES

For now, we will assume that no reassignments are possible, and that the only resource is CPU time. Our goal, therefore, is to minimize the maximum CPU load.

When the machines are identical, and no other resources are relevant, the *greedy algorithm* performs well. This algorithm assigns the next job to the machine with the minimum current CPU load. The greedy algorithm has a competitive ratio of 2 – 1/n (see [5]). When the machines are related, the greedy algorithm has a competitive ratio of log n.

3.3 UNRELATED MACHINES

ASSIGN-U is an algorithm for unrelated machines and permanent job assignments, based on an exponential function for the 'cost' of a machine with a given load [6]. This algorithm then assigns each job to a machine to minimize the total cost of all of the machines in the cluster. More precisely, let:

- a be a constant, $1 < a < 2$,
- $l_i(j)$ be the load of machine i before assigning job j, and
- $p_i(j)$ be the load job j will add to machine i.

The online algorithm will assign j to the machine i that minimizes the marginal cost

$$H_i(j) = a^{l_i(j)+p_i(j)} - a^{l_i(j)}.$$

This algorithm is O(log n) competitive for unrelated machines and permanent jobs. The work presented in [7] extends this algorithm and competitive ratio to temporary jobs, using up to O(log n) *reassignments* per job. A reassignment moves a job from its previously assigned machine to a new machine. In the presence of reassignments, let

- $h_i(j)$ be the load of machine i just before j was last assigned to i.

When any job is terminated, the algorithm of [7] checks a 'stability condition' for each job j and each machine M. This stability condition, with i denoting the machine on which j currently resides, is:

$$a^{h_i(j)+p_i(j)} - a^{h_i(j)} \le 2*(a^{l_M(j)+p_M(j)} - a^{l_M(j)})$$

If this stability condition is not satisfied by some job j, the algorithm reassigns j to machine M that minimizes $H_M(j)$.

For unrelated machines and temporary jobs, without job reassignment, there is no known algorithm with a competitive ratio better than n.

3.4 ONLINE ROUTING OF VIRTUAL CIRCUITS

The *ASSIGN-U* algorithm above minimizes the maximum usage of a single resource. In order to extend this algorithm to several resources, we examine the related problem of online routing of virtual circuits. The reason this problem is applicable will be discussed shortly. In this problem, we are given:

- A graph G(V,E), with a capacity $u(e)$ on each edge e,
- A maximum load mx, and
- A sequence of independent requests (s_j , t_j , $p{:}E \rightarrow [0,mx]$) arriving at arbitrary times. s_j and t_j are the source and destination nodes, and $p(j)$ is the required bandwidth. A request that is assigned to some path P from a source to a destination increases the load l_e on each edge $e \in P$ by the amount $p_e(j) = p(j)/u(e)$.

Our goal is to minimize the maximum link congestion, which is the ratio between the bandwidth allocated on a link and its capacity.

Minimizing the maximum usage of CPU and memory, where memory usage is measured in the fraction of memory consumed, can be reduced to the online routine problem. This reduction works as follows: create two nodes, {s, t} and n non-overlapping two-edge paths from s to t. Machine I is represented by on e of these paths, with a *memory* edge with capacity $r_m(i)$ and a *CPU* edge with capacity $r_c(i)$. Each job j is a request with s as the source, t as the sink, and p a function that maps memory edges to the memory requirements of the job and CPU edges to 1. The maximum link congestion is the larger of the maximum CPU load and the maximum memory (over)usage.

ASSIGN-U is extended further in [6] to address the online routing problem. The algorithm computes the marginal cost for each possible path P from s_j to t_j as follows:

$$H_P(j) = \sum a^{l_e + p_e(j)} - a^{l_e} ,$$

and assigns request j to a path P that yields a minimum marginal cost.

This algorithm is O(log n) competitive [6]. By reduction, it produces an algorithm for managing heterogeneous resources that is O(log n) competitive in its maximum usage of each resource.

4. FROM THEORY TO PRACTICE

For each machine in a cluster of n machines, with resources $r_1 \ldots r_k$, we define that machine's *cost* to be:

$$\sum_{i=1}^{k} f(n, utilization\, of\ r_i) ,$$

where f is some function. In practice, using ASSIGN-U, we will choose f so that this sum is equal to:

$$\sum_{i=1}^{k} n^{\frac{utilized\ r_i}{\max usage\ of\ r_i}} .$$

The *marginal cost* of assigning a job to a given machine is the amount by which this sum increases when the job is assigned there. An "opportunity cost" approach to resource allocation assigns jobs to machines in a way that minimizes this marginal cost. ASSIGN-U uses an opportunity cost approach.

In this paper, we are interested in only two resources, CPU and memory, and we will ignore other considerations. Hence, the above theory implies that given logarithmically more memory than an optimal offline algorithm, ASSIGN-U will achieve a maximum slowdown within O(log n) of the optimal algorithm's maximum slowdown.

This does not guarantee that an algorithm based on ASSIGN-U will be competitive in its average slowdown over all processes. It also does not guarantee that such an algorithm will improve over existing techniques. Our next step was to verify that such an algorithm does, in fact, improve over existing techniques in practice.

The memory resource easily translates into ASSIGN-U's resource model. The cost for a certain amount of memory usage on a machine is n^u, where u is the proportional memory utilization (used memory / total memory.) For the CPU resource, we must know the maximum possible load. Drawing on the theory, we will assume that L, the smallest integer power of two greater than the largest load we have seen at any given time, is the maximum possible load. This assumption, while inaccurate, does not change the competitive ratio of ASSIGN-U.

The cost for a given machine's CPU and memory load, using our method, is:

$$n^{\frac{used\ memory}{total\ memory}} + n^{\frac{CPU\ load}{L}} .$$

In general, we will assign or reassign jobs so as to minimize the sum of the costs of all the machines in the cluster.

To examine the behavior of this "opportunity cost" approach, we evaluated four different methods for job assignment:

1. **PVM** (for "Parallel Virtual Machine") is a popular metacomputing environment for systems without preemptive process migration. Unless the user of the system specifically intervenes, PVM assigns jobs to machines using a strict Round-Robin strategy. It does not reassign jobs once they begin execution.

2. **Enhanced PVM** is an opportunity cost-based strategy that assigns each job to the machine where the job has the smallest marginal cost. As with PVM, initial assignments are permanent.

3. **Mosix** is a set of kernel enhancements to BSD/OS that allows the system to migrate processes from one machine to another without interrupting their work. Mosix uses an improved load-balancing strategy that also endeavors to keep some memory free on all machines. Mosix is not omniscient; when the system is exchanging process information in preparation for possible process reassignment, each machine is only in contact with a limited selection of other machines.

4. **Enhanced Mosix** is an opportunity cost-based strategy intended for use on systems (such as Mosix clusters) that can preemptively migrate processes. It assigns or reassigns jobs to minimize the sum of the costs of all of the machines. Enhanced Mosix has the same limits on its knowledge as unenhanced Mosix.

5. EXPERIMENTAL RESULTS

Our first test of the ASSIGN-U algorithm was a Java simulation of the four job (re)assignment methods above. We based our simulated cluster on the local cluster of six Pentium machines. Each incoming job required $2/r$ seconds of CPU time on the fastest machine and $(1/m)$% of the largest machine memory, where r and m were independently-generated random numbers between 0 and 1. (This distribution is based on the observations of real-life processes in [8].) Since these algorithms are meant for metacomputing clusters, 5% of all jobs instead required 20/*r* seconds, and are divided into 1 to 20 parallel components. Jobs arrived at about one per ten seconds for one thousand simulated seconds, distributed randomly to provide a variety of load conditions to each of our methods.

In each execution of the simulation, all four methods were provided with an identical scenario, where the same jobs arrived at the same rate.

5.1 SIMULATION RESULTS

The results of the simulations were evaluated in two different ways:

- An important concern is the overall slowdown experienced using each of the four methods. The *average slowdown by execution* is an unweighted average of all of the simulation results, regardless of the number of jobs in each execution. The *average slowdown by job* is the average slowdown over all of the jobs in all of the executions of the simulation. These results, incorporating 3000 executions, are given in Table 1.
- The behavior of Enhanced PVM and Enhanced Mosix is different in lightly-loaded and heavily-loaded scenarios. This behavior is illustrated in Figures 1 to 4, detailing the first 1000 executions of the simulation.

Slowdown by …	PVM	Enhanced PVM	Mosix	Enhanced Mosix
(Job)	15.4044	10.7007	9.4208	8.20262
(Executions)	14.3338	9.79463	8.5567	7.47886

Table 1: Average slowdown in the Java simulation for the different methods.

Each point in the figures below represents a single execution of the simulation for the two named methods. In Figure 1, the X axis is the average slowdown for PVM, and the Y axis is the average slowdown for enhanced PVM. Similarly, in Figure 2, the X axis is the average slowdown for Mosix, and the Y axis is the average slowdown for enhanced Mosix. The light line is defined by ‘x = y’. Above this line, the un-enhanced algorithm does better than the enhanced algorithm. Below this line, the enhanced algorithm does better than the un-enhanced algorithm.

Enhanced PVM, as Table 1 has already shown, does significantly better than straight PVM in almost every circumstance. More interesting, however, is the behavior of enhanced Mosix when compared to Mosix. The larger Mosix’s average slowdown was on a given execution, the more improvement our enhancement gave. Intuitively, when an execution was hard for all four models, Enhanced Mosix did much better than unenhanced Mosix. If a given execution was relatively easy, and the system was not heavily loaded, the enhancement had less of a positive effect.

This can be explained as follows. When a machine becomes heavily loaded or starts thrashing, it does not just affect the completion time for jobs already submitted to the system. If the machine does not become unloaded before the next set of large jobs is submitted to the system, it is effectively unavailable to them, increasing the load on all other machines. If many machines start thrashing or become heavily loaded, this effect will build on itself. Every incoming job will take up system resources for a much longer span of time, increasing the slowdown experienced by jobs that arrive while it computes. Because of this pyramid effect, a ‘wise’ initial assignment of jobs and careful re-balancing can result (in the extreme cases) in a significant improvement over standard Mosix, as shown in some of the executions in Figure 2.

It is particularly interesting to note that, as seen in Table 1 and Figure 3, the enhanced PVM method, which makes no reassignments at all, manages to achieve respectable (though inferior) performance compared to Mosix. This emphasizes the power of the opportunity cost approach: its performance on a normal system is not overwhelmed by the performance of a much superior system that can correct initial assignment mistakes.

The importance of migration is demonstrated by Figure 4. Even when using the opportunity cost algorithm, it is still very useful to have the migration ability in the system. In fact, Enhanced Mosix outperform Enhanced PVM in all of the cases, sometimes considerably.

5.2 REAL SYSTEM EXECUTIONS

Our algorithms were also tested on a real cluster. The same model for incoming jobs was used, and jobs were assigned using the PVM, Enhanced PVM, and Mosix strategies. Enhanced Mosix has not yet been implemented on a real system.

Tables 2 and 3 show the slowdowns for 50 executions on this real cluster. Figures 5 and 6 show the results point-by-point. The results of the real system executions are as follows:

Slowdown for …	PVM	Enhanced PVM	Mosix
(average by job)	19.81842	12.2721	8.474874
(by execution)	18.83488	11.69833	8.683132

Table 2: Average slowdown in the real cluster for 3 (re)assignment methods.

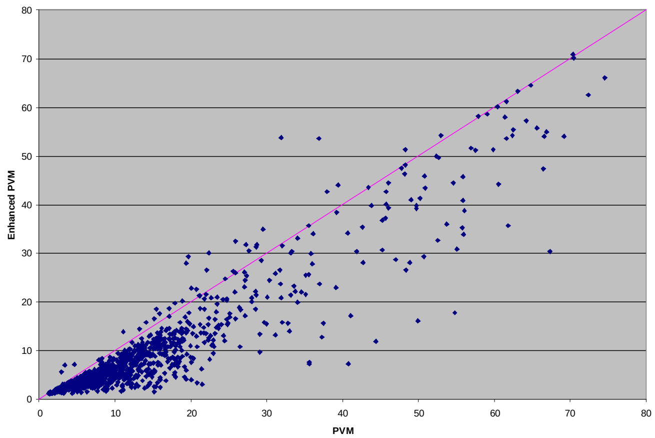

Figure 1: PVM vs. Enhanced PVM
(Simulation)

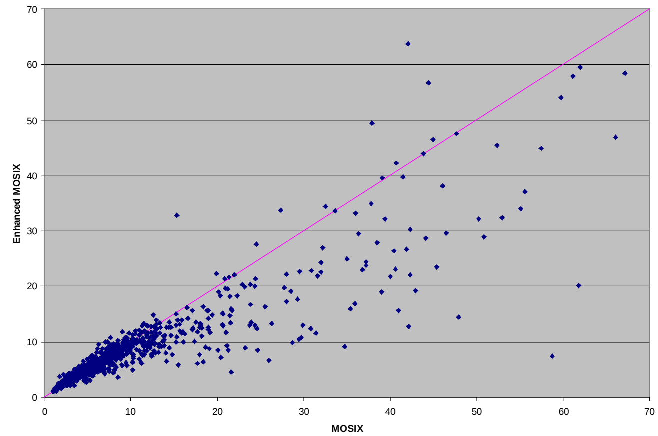

Figure 2: Mosix vs. Enhanced Mosix
(Simulation)

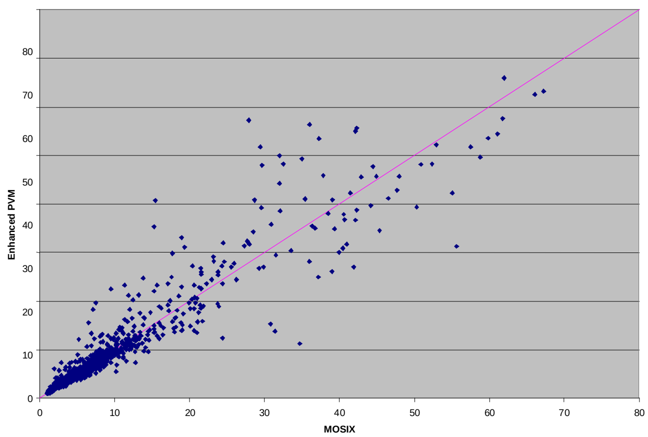

Figure 3: Mosix vs. Enhanced PVM
(Simulation)

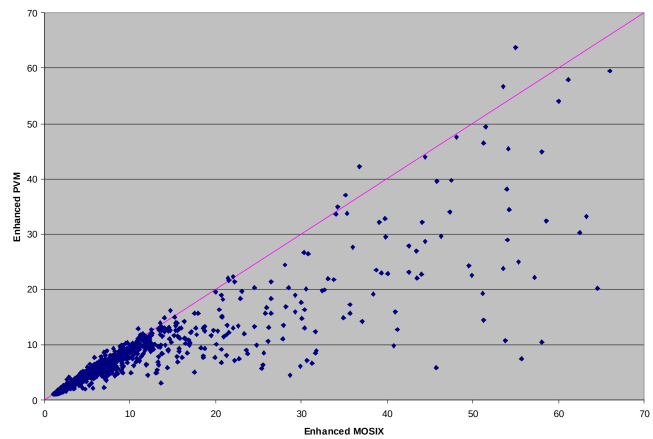

Figure 4: Enhanced Mosix vs. Enhanced PVM
(Simulation)

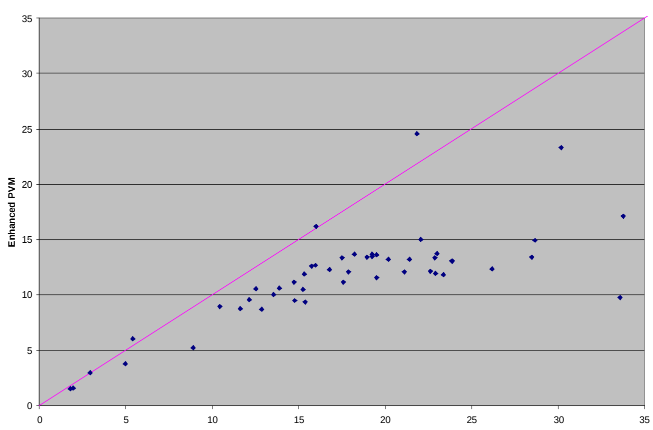

Figure 5: PVM vs. Enhanced PVM
(Real Executions)

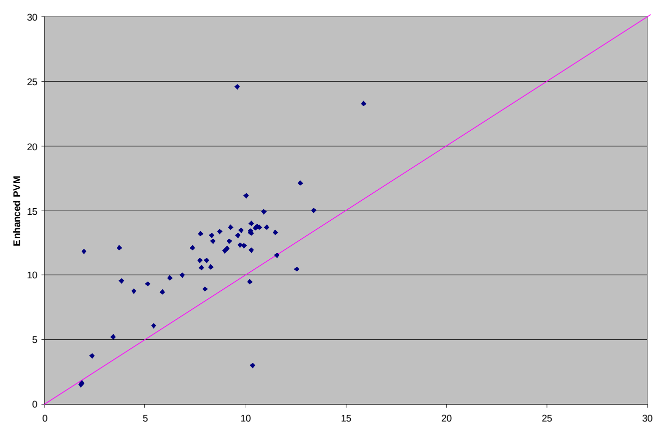

Figure 6: Mosix vs. Enhanced PVM
(Real Executions)

The test results in Table 2 imply that the real-life thrashing constant and various miscellaneous factors increased the average slowdown. This indicates that we were too conservative in picking simulation parameters. Nevertheless, these results do not substantially change the relative values. Mosix performed substantially better on a real system, as expected, but Enhanced PVM also performed better, compared to regular PVM. We consider this to be a strong validation of our initial Java simulations and of the merits of this opportunity cost approach.

Slowdown on Real System for …	**PVM vs. Mosix**	**E.PVM vs. Mosix**	**PVM vs. E.PVM**
(average by job)	2.282405	1.413327	1.614917
(by execution)	2.222438	1.380355	1.610048
Slowdown in Simulation for …	**PVM vs. Mosix**	**E.PVM vs. Mosix**	**PVM vs. E.PVM**
(average by job)	1.635148	1.13859	1.439569
(by execution)	1.675155	1.144674	1.463435

Table 3: Average relative slowdowns for 3 job (re)assignment methods.

This table shows the ratio of the slowdowns for the various methods. Thus, the PVM vs. Mosix column shows the ratio of PVM's slowdown to Mosix's slowdown using the relevant measure. The first method named provides the numerator.

6. CONCLUSIONS

The opportunity cost approach is a universal framework for efficient allocation of heterogeneous resources. The theoretical guarantees are weak: one can only prove a logarithmic bound on the performance difference between the algorithm and the optimum offline schedule. However, the optimum offline schedule is not really an option; in reality, our algorithm competes with naive online heuristics.

In practice, this approach yields simple algorithms that significantly outperform widely used and carefully optimized methods. We conclude that the theoretical guarantees of logarithmic optimality is a good indication that the algorithm will work well in practice.

REFERENCES

[1] The Mosix Multicomputer Operating System for Unix http://www.mosix.cs.huji.ac.il

[2] A. Barak and O. La'adan, The MOSIX Multicomputer Operating System for High Performance Cluster Computing, *Journal of Future Generation Computer Systems*, Vol. 13, No. 4-5, pages 361-372, March 1998.

[3] Berkeley Software Design, Inc. Web site, http://www.bsdi.com.

[4] B. Awerbuch, Y. Azar and A. Fiat, Packet Routing via Min-Cost Circuit Routing, *Proceedings of the Israeli Symposium on Theory of Computing and Systems*, 1996.

[5] Y. Bartal, A. Fiat, H. Karloff and R. Vohra, New algorithms for an ancient scheduling problem, *Proceedings of the ACM Symposium on Theory of Algorithms*, 1992.

[6] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin and O. Waarts, On-Line Machine Scheduling with Applications to Load Balancing and Virtual Circuit Routing, *Proceedings of the ACM Symposium on Theory Of Computing (STOC)*, 1993.

[7] B. Awerbuch, Y. Azar, S. Plotkin and O. Waarts, Competitive Routing of Virtual Circuits with Unknown Duration, *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1994.

[8] M. Harchol-Balter and A. Downey, Exploiting Process Lifetime Distributions for Dynamic Load Balancing, *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.