Kappa: A system for Linux P2P Load Balancing and Transparent Process Migration

Gaurav Mogre
NITK
Surathkal, India
gaurav.mogre@gmail.com

Avinash Hanumanthappa
NITK
Surathkal, India
avinash947@gmail.com

Alwyn Roshan Pais
NITK
Surathkal, India
alwyn@nitk.ac.in

***Abstract*—Process migration is the act of transferring a process between two computers. If used effectively, process migration could be used to improve the throughput of a cluster of computers. To perform this, a process must be migrated from a "slow" system to a "fast" system. To measure the "fastness" of a system, various parameters must be considered, such as the current load of the system, the time slices expected to be given to a new process, and the speed of the hardware.**

In our system, named Kappa, the various parameters are aggregated together to form the system metric of a system. This metric is then used to determine which computer the process should be migrated to. In order to perform this, load balancing needs to be performed. Load Balancing in Kappa is performed in a peer-2-peer network in two steps: An initial approximation stage, and a probing stage. In the approximation stage, a rough measure of the distribution of the system metric is calculated within the cluster. In the probing stage, the initial information collected is used to direct a probe across the network to find the optimal target for load balancing. The actual process migration takes places transparently once the target machine is found.

Kappa was implemented in Linux, and allows for transparent process migration between paired Linux computers. It is implemented as a user-space program, with minimal hooks in the system calls. Kappa was tested for a cluster of Linux computers with varying system metric, and an improvement of 50% to 200% in execution time was recorded for CPU intensive processes.

Keywords- linux, process migration, two stage load balancing, system performance, peer to peer

I. INTRODUCTION

The future of computing lies in the field of High Performance Computing. Cluster Computing techniques have proven to improve the throughput of a cluster of computers. But even though we see advances in the high performance computing field, we also see that these advances have not trickled down to home users. Systems such as MOSIX have not gained popularity amongst home users for a variety of reasons. One such reason is the change in usability of the system. SSI (Single System Image) based techniques ask for a radical shift in the operation of computers.

We aim to bridge the gap between high-performance computing and home users by creating a system which utilizes the power of distributing tasks across a cluster, while maintaining the usability of single-user machines. To achieve this, it is not possible to keep processor scheduling and resource allocation as shared decisions. Thus, a collaborative means to allow processes to execute within the cluster must be made.

Kappa tries to bridge this gap between high-performance computing and home users.

II. TERMINOLOGY AND DEFINITIONS

- System Metric: A relative approximate measure of the execution time of a process executing within a system.
- Distribution Metric: A relative measure of the time to migrate a process and complete its execution on a remote system
- Probing: Sending a probe message along a directed path until a system with suitable distribution metric is found
- Transparent process migration: Process migration that requires no change of existing programs to utilize the facility.

III. SURVEY OF EXISTING SYSTEMS

A study of existing systems for process migration was done. The systems studied were Mosix, Sprite, Mach, and LSF/Condor. A comparison of the features of these systems are listed in Table I

TABLE I. COMPARISON OF FEATURES OF EXISTING SYSTEMS

Characteristics	**Existing Systems**			
	Mosix	***Sprite***	***Mach***	***LSF/Condor***
Initial Migration time	Moderate	moderate	Low	Low
Residual	None	None	Yes	None

dependency				
Residual time and costs	None	Moderate	High	None
Freeze cost	Moderate	Moderate	Small	Moderate
Freeze time	Moderate	Moderate	Low	Moderate
Transparency	Full	Full	Full	Limited
decentralization	Distributed	Centralized	Distributed	centralized
Fault resilience	Yes	Limited	No	yes
Knowledge relevance	Aging	Periodic	Negotiation	none

Observations: We see that the systems that were discussed have their own characteristics and goals. We see that the following goals are not the focus of these systems:

- Implementation of a popular platform: Systems such as MOSIX and Sprite are implemented on the Unix operating system. We see that these systems are not geared towards more popular commercial platforms. Mach is a microkernel in itself, while LSF and Condor aren't full-fledged process migration solutions
- Maintaining "single-user" interface: The MOSIX and Sprite systems are both based on a SSI architecture, in which a distributed system is identified as a single entity. The Mach system partially gears towards SSI. LSF and Condor are libraries, which don't offer an integrated solution. Using these systems would mean users must get accustomed to a new environment of execution.
- Non-preemptive process migration: The systems that were discussed use a check pointing mechanism to store the state of a process before it is migrated. This would mean an additional overhead to understand the process state, and to send it across a network, and to resume it on the target system. We could reduce this overhead by allowing non-preemptive process migration. While this may reduce the amount of task migration, it would also reduce the overhead with the migration.

IV. Design

Kappa performs four main tasks to carry out peer-2-peer load balancing and process migration:

- Analysis of System Metric
- Preliminary information gathering (for load balancing)
- Probing
- Process Migration

A. System Metric Determination

The system metric is a relative measure of how fast a system is expected to finish execution of a process since the time it has been created on that system. The system metric is composed of two parts: (a) Performance of hardware (b) Current System Load

1) Performance of the CPU

To measure the CPU performance, two methods can be applied: (a) Measurement of the various parameters (b) Benchmarking. For Kappa, we used standard tools for benchmarking the CPU performance. This is because: (a) It is not always possible to determine programmatically the various parameters of the processor, (b) It is not always possible to quantify the measurement of a certain parameter. For eg. The memory architecture, branch prediction schemes, etc. cannot be directly quantified without a context, (c) There exist benchmarking standards which accurately measure the performance of the processor.

To test the CPU performance, the SPEC CPU2006 was chosen. However, since we need a relative and approximate measure of the CPU performance, the SPEC2006 was subsetted[9] using PCA and k-clustering to give four benchmark programs: sjeng(), gcc(), libquantum() and xalan(). Since the CPU2006 suite is not freely available, the programs were individually downloaded from their respective sites and tested on the platform. The workloads supplied to these benchmarks were also reduced such that the benchmarks finish execution under a minute.

Each benchmark is run on the system, and the result of each benchmarked can be summarized as:

$$Result_{Benchmark\_1} = \frac{(time\ to\ complete\ on\ a\ given\ CPU)}{(time\ to\ complete\ on\ a\ very\ fast\ CPU)} \quad (1)$$

All these results can then be combined to form a metric which represents the hardware performance of the system:

$$Sysmetric_{static} = \frac{\sum Result_{Benchmark\_i}}{num\_benchmarks} \quad (2)$$

2) Measuring System Load

To measure the system load, tools such as vmstat, top, and mpstat were analyzed under various system loads. The various indicators of the system load are: (a) The number of time slices spent idling, (b) The expected number of time slices to be assigned to a new process (c) The amount of free memory

The measurements are taken every five seconds from /dev/proc. A significant measure of the speed of execution of a process is the number of time slices that the process is assigned. Thus, the initial load is determined by the number of time slices that a process is expected to be assigned. This can be approximated from the number of time slices spent idling, as well as the number of time slices that were spent by the user processes. The initial system metric is determined by (3)

$$\mathit{Approximate\ Load} \propto (\mathit{jiffies\ spent\ idling}) + \frac{\mathit{jiffies\ spent\ executing\ user\ processes}}{\mathit{number\ of\ processes} + 1} \quad (3)$$

The effect of memory and memory faults becomes relevant only when there is a shortage in memory. Thus, we account for the memory only when the free memory available becomes less than a threshold.

$$Sysmetric_{dynamic} = \frac{1}{load_{approximate}} * \frac{\left(\frac{memory_{available}}{memory_{total}}\right)}{freemem_{threshold}} \quad (4)$$

3) Distribution Metric

The performance of the hardware is checked at the time of a reboot of the computer. The metric collected by static measurements of the system by using benchmarks is denoted by $Sysmetric_{static}$. Meanwhile, the system load is measured at regular intervals. The final system metric is then a weighted combination of these two metrics:

$$Sysmetric = \frac{(Sysmetric_{static} * 0.3) + (Sysmetric_{dynamic} * 0.7)}{2} \quad (5)$$

The distribution metric is a measure of time required between sending a process to a remote system, and it returning the results back to the host system. Thus, the distribution metric includes the overhead involved in migrating a process, along with the system metric of the remote system

$$dismetric = sysmetric + constant_{network\ delays} \quad (6)$$

B. Preliminary Information Gathering

This stage involves gathering information about the current system load that exists over a network. Since we aim towards a peer-2-peer architecture, notifying each system in the cluster would lead to increased overhead. Hence an approximate algorithm was used for this step. The aims of this algorithm were: (a) Scalability and (b) Minimum overhead

The foundation of the algorithm was the hierarchical load balancing algorithm. In hierarchical load balancing, the nodes in the cluster are arranged in the form of a tree and load balancing is performed from parent to child. To extend this algorithm, a Directed Acyclic Graph was chosen to replace the tree, with appropriate modifications made to the algorithm. While this modified algorithm successfully demonstrated various properties that were useful for the algorithm, embedding a directed acyclic graph within the cluster in a distributed way impaired the scalability of this algorithm, and modifications to this algorithm were required. This algorithm is given in algorithm I

Algorithm I: Preliminary Information Gathering

```
Procedure: Init_measurements()
  1.  Calculate system metric of system.
  2.  Assign send_probe = the system metric
  3.  Send send_probe to all neighbors of the current system
  4.  Set max_dist = 0
  5.  Set rec_time = Current system time
  6.  If ((current system time) - rec_time) > delay, then go to step
      9
  7.  For each neighbor i:
        a.  Get a message from i containing send_probe(i)
        b.  Set dist_metric(i) = send_probe(i) - constant_overhead
        c.  if dist_metric(i) > max_dist, assign max_dist =
            dist_metric(i)
  8.  Go to step 6
  9.  Get current system metric
  10. If current system metric > max_dist, set send_probe =
      current system metric. Else, set send_probe = max_dist
  11. Go to step 3
```

In algorithm I, the cluster is represented by a simple undirected graph. Each system in the cluster corresponds to a single vertex in the graph. A system can be linked with another system by using a pairing procedure. Each paired link between two systems can be represented by an edge in the graph. When a new system wishes to join a cluster, it must know one system that is present in the cluster. This can directly pair with the new system, without informing its neighbors. Similarly, when a system wishes to be removed from the cluster, it simply unpairs all its neighbors.

Initially, a system in the cluster sets it send_metric variable to its current system metric. It then sends this value to all its neighbors, and then waits for a fixed period of time. During this time, the system acquires the send_metric of all its neighbors. To this value, the system deducts a constant overhead to give the distribution metric of each neighbor system. It then checks the neighbor with the maximum distribution metric, and stores this value in max_dist. Periodically, the system will check its own system metric. It then finds the maximum of the current system metric and max_dist, and stores this in send_metric. The system then sends the send_metric to all its neighboring systems. This procedure proceeds until the system is removed from the cluster.

This algorithm has the following characteristics:

- Linear complexity: Each machine in the cluster performs the algorithm in worst case O(n).
- Convergence: The values of the metrics never converge. The distance of a system from another system is proportional to how relevant the information about the current load is.
- Assumption: The algorithm works effectively only when the increase in load of a system is gradual. If the increase in loading of machines is expected to be more sudden, then the various

parameters of the algorithm, can be modified. We see that reducing these parameters, would improve handling of more sudden load changes, but would induce higher overheads.

- Coherence: The algorithm uses the value of system metric taken at regular intervals of time. Hence, it is possible to have a system which has a system metric much higher than the value of distribution metric stored in its neighbor (and consequently, the one that is transferred by the neighbor). However, by ensuring the delay added in every step is high enough, such a scenario could be avoided.

C. *Probing*

Probing is the procedure to find the exact node to which a process must be migrated to before the process migration. The aims of probing are: (a) to get the node with the optimal load, to handle the process execution and (b) to allow for the transitivity property to hold: A process may be transferred from a source node to a destination node, where the source and destination need not necessarily be directly paired.

Unlike the hierarchical load balancing algorithm which worked on timely pings, the probe algorithm is a sender-initiated strategy. The Probe message is sent just before the process is going to start executing, to find an appropriate destination node to deliver the process to. The probing algorithm uses two kinds of messages: Probe requests and probe replies.

A probe message can be represented as P(i,j,k) where i is the system requesting the probing request, j is the system that is sending the message to the system k. Similarly, the probe reply can be represented as PR(k,i, $sysmetric_K$) which is sent from the system k to the system i which initiated the probing with $sysmetric_K$ being the current system metric of system K.

The algorithm can be divided into three parts: The probe sending, probe propagation, and probe reply

The probe sending algorithm is given in II:

Algorithm II: Sending Probe

```
Procedure: Send_probe()
  1. Get the current system metric, and compare it with highest
     distribution metric of the neighbor.
  2. If the current system metric is greater, then execute the
     process locally on the system itself.
  3. If the highest distribution metric of the neighbor is higher,
     then first get the neighbor.
  4. Send a probe message to the neighbor.
  5. Wait for a reply. If a reply arrives, store it.
```

The probe propagation and reply steps are given in algorithm III:

Algorithm III: Propagation of Probe

```
Procedure: propagate_probe()
  1. Consider that node K receives the probe P(I, J, K).
  2. Check if the current system metric is greater than the
     distribution metric obtained from any neighbor.
  3. If the system metric of K is higher, then send back Probe
     reply PR(K, I, $sys\_metric_K$). Go to end.
  4. If the system metric is lower, get the neighbor with the
     highest distribution metric. Let this neighbor be M.
  5. Send the probe P(I, K, M) to the neighbor M. End
```

D. *Process Migration*

Process migration can be carried out either preemptively and non-preemptively. In non-preemptive process migration, a process is migration only at its creation. Preemptive process migration allows processes which are already executing, to be migrated. Kappa uses the non-preemptive process migration strategy. This is because of the following:

- Additional overhead of saving the state of a process before it is migrated.
- Additional overhead of setting up the same state of the process after migration.
- The bookkeeping involved with preemptive process migration is higher.

Thus, a non-preemptive strategy was chosen.

V. RESULTS

Kappa was tested with a few preliminary programs under various conditions of the CPU

A. *System Load*

To implement the system, systems A and B were chosen, such that A was loaded with 4 processes while B was idling. The programs were run on A. We see that the system metric of A varied from 35 to 45. On the other hand, B's system metric varied from 140 to 150. When these metrics were distributed in the 2nd step, the preliminary information gathering, it was found that: $distmetric_B$ on A varied from 125 to 135, while $distmetric_A$ on B varied from 20 to 30.

When the process is about to migrate, a probe message is sent from A to B, since $distmetric_B > sysmetric_A$. This probe message was received and replied to by B since B contains no neighbors with distmetric higher than $sysmetric_B$. B sends a probe message PR(B, A, 142), where the $sysmetric_B$ varies, back to A. The effect of process migration on the execution times(in microseconds) of the various processes is given in Figure 1.

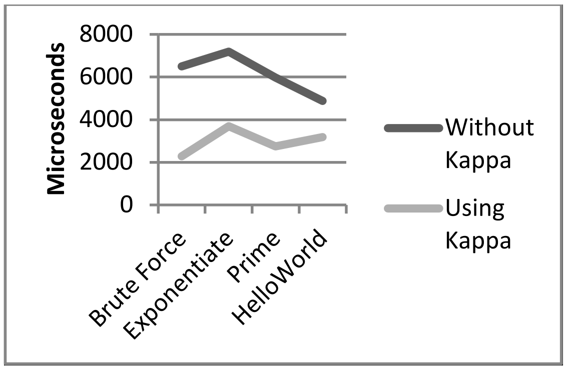

Figure 1. Results: System Load Change

The results can be summarized in Table III

TABLE II. RESULTS OF PROCESS MIGRATION

Process	Sysmetric$_A$	Sysmetric$_B$	Dismetric$_B$ on A	Dismetric$_A$ on B	Exec. Time (ms)
Brute Force	37	148	133	22	2288
Exponentiate	35	144	129	20	3697
Prime	42	145	130	27	2741
Hello World	37	141	126	22	3176

B. Load Balancing

To test the load balancing performance, three systems, A, B and C, were chosen. B was paired with both A and C. A and B were loaded with five processes each, while C was left idling. Then infinite loop processes were created in A to see the effect of increasing the load. The results (and comments) are listed in Table IV.

TABLE III. EFFECT OF INCREASING LOAD

Process No.	Result	Comment
1	No migration	The probe reply wasn't received before the process ran
2	Migrated to System C.	After 2 "delays" to pass for convergence.
3-7	Migrated to System C	Fork and execve separated by some jumps to allow probe replies to arrive.
8-9	No Migration	The System C load is higher than System B, but B's cache is not updated. Probe reply finds out that migration is not needed
10	Migrated to System B	System B has the least load

VI. CONCLUSIONS AND FURTHER SCOPE

Kappa successfully improved the overall system performance by effective load balancing and process migration. Results show an improvement of around 90% for CPU intensive applications. We see that the load balancing scheme is scalable, efficient, and without a central point of failure.

Kappa can be easily incorporated in a system. It allows transparent process migration, and hence no changes to programs are necessary to utilize the process migration facility. The system also does not modify the usability of the system, since it seamlessly integrates into the system. Hence, we see that Kappa bridges the gap between Home computing and cluster computing.

A few improvements that could be introduced to the system are:

- Higher piggybacking of commands between migrated process and stub and use of Migratable Sockets for migrated processes
- Preprocessing of ELF string table and estimation of the files required by the process, so it could be sent along with the process.
- Preprocessing a process to determine if it is CPU and IO intensive, and accordingly migrate process

REFERENCES

[1] D. Milojicic, F. Douglas, Y. Paindaveine, R. Wheeler and S. Zhou, "Process Migration," ACM Computing Surveys (CSUR), vol.32 n.3, pp. 241–299, September 2000.

[2] J.M. Smith, "A survey of process migration mechanisms," ACM SIGOPS Operating Systems Review, vol. 22 n.3, pp. 28-40, July 1988.

[3] M. Bubak, D. Zbik, D. Albada, K. Iskra and P. Sloot, "Portable library of migratable sockets," Scientific Programming, vol.9 n.4, pp. 211-222, December 2001, IOS Press.

[4] M. Claypool, D. Finkel, "Transparent Process Migration for Distributed Applications in A Beowulf Cluster," Proceedings of the International Network Conference 2002, pp. 459-466, July 2002.

[5] N. Vasudevan, P. Venkatesh, "Design and Implementation of a Process Migration System for the Linux Environment," 3rd International Conference on Neural, Parallel and Scientific Computations, Aug 2006.

[6] T.L. Casavant, J.G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," IEEE Transactions on Software Engineering, vol. 14 n.2, pp. 141-154, Feb 1988, IA: IEEE.

[7] R. Wolski, N. Spring, J. Hayes, "Predicting the CPU availability of time-shared Unix systems on the computational grid," Cluster Computing, vol. 3 n.4, pp. 293-301, 2000, MA: Kluwer Academic Publishers.

[8] A. Phansalkar, A. Joshi, L. John, "Subsetting the SPEC CPU2006 benchmark suite," ACM SIGARCH Computer Architecture News, vol. 35 n.1, pp. 69-76, 2007, NY:ACM.