

Load Balancing by Remote Execution of Short Processes on Linux Clusters *

M. Kačer, P. Tvrdík

Department of Computer Science and Engineering
CTU FEE, Karlovo nám. 13, 121 35 Prague, Czech Republic
{xkacer,tvrdik}@fel.cvut.cz

1 Introduction

PC clusters typically operate under continuously changing conditions and load balancing is critically important for efficient utilization of their resources, for maximizing their performance, and for minimizing the process response times. The average process response time is usually considered the most important value for measuring the actual performance of a multitasking system.

Clearly, load balancing is critical for clusters running long-term processes (run times of hours). The main motivation of this paper is to study the suitability of the load balancing of short-term CPU-intensive processes, e.g., compilers, compression utilities, etc. Such processes represent a large part of a typical workload of a Linux workstation and load balancing can improve the process response time considerably in such cases. This topic appears to be neglected in the literature.

The load balancing can be implemented by *process migration* (when a running process is stopped, checkpointed, checkpoint data are transferred to another node, and the process is restarted there) or by *remote execution* (when a process is transferred to another node when it is started and has no allocated memory). While the former solution is more universal (the process can be transferred in any moment), the latter one causes less overhead, which is very important especially when short processes are considered.

A load-balancing algorithm is composed of three basic parts: a *transferring mechanism*, which implements remote execution or process migration, a *load-balancing strategy*, which makes decisions which process is to be moved where, and an *information policy*, which collects the information about the cluster state and distributes it among all the nodes.

In Linux, a new process is created by `fork` system call, which makes an exact copy of the parent process, includ-

ing its memory mapping. The child process shares memory segments with its parent. The proper moment to perform a remote execution is `execve` system call, when a new program is loaded instead of the old one, the memory mapping is lost, but other process attributes are retained (e.g., the identifier, open files and their descriptors, environment variables). Thus, the process can be transferred to another node during the `execve` call without the need of transferring memory contents, which forms the largest overhead of process migration.

The contribution of this paper is twofold. First, we argue that load-balancing strategies and transferring mechanisms should be separated by a suitable interface. Second, we give evidence that a simple remote execution outperforms a process migration in case of a typical Linux cluster heavily but non-uniformly loaded with short-term (order of seconds) processes. The main conclusion is that the load balancing in Linux clusters should be performed by combining both the process migration and remote execution and the load-balancing strategy should be represented by user-level programs independent of the transferring mechanisms.

2 Proposed Solution

The problem of efficient process scheduling is very hard in general and no load-balancing strategy is optimal for all purposes. A cluster user should have a possibility to try various heuristic algorithms to find the best solution for a particular situation. To provide a user and a system administrator with such a flexibility, we propose to separate the transferring mechanisms implemented in the kernel from the load-balancing strategies implemented on the user level. A proper interface must be defined between them. The kernel mechanisms inform the load-balancing module about the node state and local process statistics. On the other hand, the load-balancing module delivers its decisions to the transferring mechanism. Similar arguments hold for the information policy module, which collects global statistics

*This research has been supported by FRVŠ under grant #1888/2001 and by MŠMT under research program #J04/98:212300014.

using collective communication operations.

We have implemented a transparent kernel-level mechanism that is able to execute a Linux process remotely whenever a new program is run. When an `execve` system call is invoked, the mechanism intercepts the call and asks (using the simple interface described above) the user-level load-balancing daemon whether the calling process should be executed locally or remotely. Depending on the answer, the process is either left on the local node or transferred to the node determined by the daemon. After the remote execution has been performed, location-dependent system calls are transparently forwarded to the home node, to preserve their semantics. For example, reads and writes from/to files opened before the execution are sent to the home node and their results are returned to the remote node.

3 Experimental Results

Due to the complexity of clusters and various factors that have impact on the performance, it is not possible to construct a simple realistic model. Thus, only experimental measurements of process response times on real systems give reasonable data describing the load balancing.

We used a synthetic load in our experiments. The benchmark program was Floyd-Warshall algorithm for solving all-pairs-shortest-path problem. Depending on the number of vertices, a single process working with a single graph took from almost zero time to tens of seconds (up to one minute). This roughly corresponds, for instance, to run times achieved by the C language compiler `gcc`. The memory needed by a single process was approximately 0.5 MB.

In each experimental setting, several virtual users were simulated by submitting batches consisting of a random number of consecutive benchmark processes. Batches arrived in the system randomly, with the uniform distribution.

A small cluster of 3 equivalent nodes was used for all of the experiments. The nodes were PC's with 200 MHz Pentium II processor, 64 MB of memory and Linux kernel version 2.2. The kernel module providing remote execution mechanism (described before) was inserted into the kernel.

Several well-known load-balancing strategies were used in measurements [4]: **Random** (a destination node is always chosen randomly), **Round Robin**, **Threshold** (a process is executed remotely if the load of its source node exceeds a given source threshold), **Limited** (similar to Threshold, but the load of the destination node is checked), and **Shortest** (execution on the least loaded node whenever the source node is overloaded). A very simple information policy was used, based on all-to-all broadcast performed every second.

The first experiment compared the load-balancing strategies with each other. The Limited and Shortest strategies performed best in most situations. An interesting fact is that

Random and Round-Robin strategies performed very badly in our experimental settings, Random was the worst one.

For comparison, we also measured the performance of the load-balancing strategy implemented in Mosix, which is solely based on process migration. The Mosix load balancing approximately matched our best results for lower loads. For higher loads, Mosix performed considerably worse due to the larger overhead of process migration.

Other experiments were conducted to make more detailed comparison of our remote-execution-based solution with the process migration implemented in Mosix. They have shown that the simple remote execution outperformed the process migration in many cases and achieved at least comparable results in most other cases. Limited, Threshold, and Shortest strategies were used in these experiments.

4 Related Work

There are many existing systems that implement either remote execution or process migration. We mention just two of the most interesting ones: Mosix and Rhodos.

Mosix [1] is an operating system for Linux clusters. It implements a transparent preemptive process migration mechanism on the kernel level. The load-balancing strategy of Mosix uses only the process migration, not the remote execution. Another disadvantage of Mosix is the integration of the load-balancing strategy into the kernel, thus it cannot be changed easily.

Rhodos [2] is a microkernel-based distributed operating system which has many common points with our solution. It implements both remote execution and process migration, the transferring mechanisms are separated from the load-balancing strategies with an appropriate interface. The main difference is the microkernel architecture, which makes it much easier to implement the transparent process migration and to integrate it with the rest of the system. The achieved results are not automatically applicable to Linux clusters and our experiments with Linux have resulted in slightly different conclusions than those stated in [2].

References

- [1] A. Barak, O. La'adan, A. Shiloh: *Scalable Cluster Computing with MOSIX for Linux*. Linux Expo '99, pp. 95–100.
- [2] D.De Paoli, A.Goscinski, M.Hobbs, P.Joyce: *Performance Comparison of Process Migration with Remote Process Creation Mechanisms in RHODOS*. Proceedings of the IEEE 16th ICDCS '96, Hong Kong, 1996, pp. 554–561.
- [3] A. Downey, M. Harchol-Balter: *A Note on "The Limited Performance Benefits of Migrating Active Processes for Load Sharing"*. TR, University of California at Berkeley, 1995.
- [4] B.A. Shirazi, A.R. Hurson, K.M. Kavi, eds.: *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, California, 1995.