

Dynamic Load Distribution with the WINNER System*

Olaf Arndt, Bernd Freisleben, Thilo Kielmann, Frank Thilo
University of Siegen, Dept. of Electrical Engineering and Computer Science
Hölderlinstr. 3, D-57068 Siegen, Germany

{arndt|freisleb|kielmann|thilo}@informatik.uni-siegen.de

Abstract

Networks of workstations offer large amounts of idle computing capacities. Exploiting these resources imposes the problem of efficiently distributing a given workload without disturbing the workstations' primary (console) users. Recently, specific resource management systems have been developed for this purpose [1, 4, 6, 8, 9, 11, 14].

This paper presents WINNER, a resource management system which has been designed in order to utilize the maximal amount of computing power, imperceptible even by active console users. WINNER automatically assigns tasks to suitable machines, supporting sequential as well as parallel applications, the latter based on PVM [3]. WINNER's modular structure as well as its mechanisms for monitoring and distributing workload will be presented. Performance measurements of a sample PVM application indicate the suitability of our approach.

1 Introduction

Workstation clusters are economically attractive system architectures for parallel processing applications. Because workstations are typically idle for significant fractions of the time they are running, their idle times can be exploited for parallel evaluation of computation-intensive tasks without costly investments in dedicated parallel computing hardware. Success and widespread use of systems like PVM [3] and MPI [5] indicate the feasibility of the workstation-cluster approach to parallel computing.

Workstation clusters are typically evolving systems. This is due to the fact that, for financial and technological reasons, it is not feasible to completely replace existing clusters whenever new components become available. Instead, cluster environments evolve and hence form constantly changing collections of heterogeneous components with different capabilities and requirements. Furthermore, the fraction of workstations in an existing cluster that is available for parallel computations is changing frequently. This is due to (1) individual machines being switched on or off by their primary users, (2) failures of single machines, and most importantly (3) by the workload of their primary users who should not be disturbed by parallel computations additionally operating in the cluster.

*Proc. Workshop ALV'98, March 1998, Munich, Germany

These properties of workstation clusters convert task assignment and load distribution into non-trivial tasks. Traditional strategies known from dedicated parallel machines can no longer be applied because they rely on exclusively accessible and furthermore identical processing elements. For solving these problems, so-called resource management systems have recently been introduced [1, 4, 6, 8, 9, 11, 14]. Such systems aim at exploiting idle computing resources of workstation clusters without disturbing the workstations' primary users.

In this paper, we present WINNER, a resource management system designed for exploiting the maximally available computing power of workstation clusters. It does so by carefully co-scheduling computing tasks with the processes of the workstations' console users while minimizing interferences between both task groups. In the following, WINNER's modular structure as well as its mechanisms for monitoring and distributing workload will be presented. Performance measurements of a sample PVM application indicate the suitability of our approach.

2 WINNER System Design

WINNER has been designed for typical Unix workstation cluster environments, consisting of a central server and several workstations. For a WINNER cluster, the server is required to provide shared file systems and user accounts for all connected workstations. The various tasks of the WINNER system are distributed over three kinds of so-called *manager* processes. Similar to the structure of the Prospero Resource Manager [9], WINNER consists of system managers, node managers, and job managers. The distribution of these manager processes across the workstations is shown in Figure 1. Additionally, there exist several user-interface tools e.g. for status reports and for influencing the usage of a user's workstation. Manager processes and user-interface tools communicate with each other by message exchange over UDP sockets. They hence directly rely on the operating system without the need for additional software products.

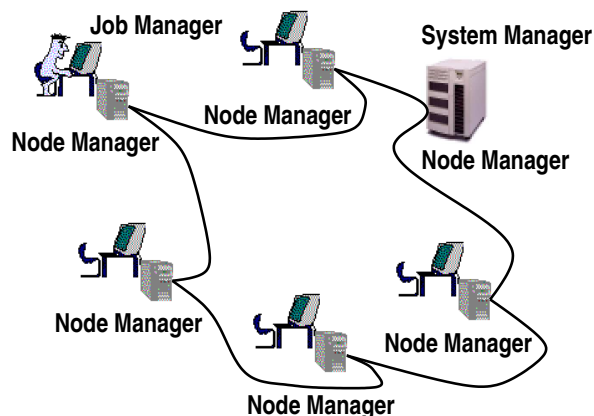


Figure 1: Manager processes in a WINNER cluster.

The system manager is the central server process of a WINNER cluster and as such a single point of failure. It is intended to be located on a cluster's server machine. In the case of a server failure, all machines of a cluster are typically not operative (e.g. due to missing file service). Therefore, the simultaneous failure of the WINNER system can be neglected. The system manager's tasks include

(a) collecting the load information of all respective workstations, (b) managing the currently active jobs, and (c) deciding which hosts are assigned to a particular job request. While each WINNER cluster has exactly one system manager, in the case of multiple, cooperating clusters several system managers interact in order to share workload.

On every host participating in a WINNER cluster, a node manager performs the tasks related to the machine it runs on. First of all, it periodically measures the host's utilization and reports these data to the system manager. Furthermore, node managers are responsible for starting and controlling WINNER processes on their node. Processes of WINNER jobs run at two different priority levels. While a console user is actively working, WINNER processes run at a low priority level in order to avoid disturbing primary users. Without an active console user, WINNER processes run at a slightly higher priority level compared to normally started user processes. This feature has been introduced in order to motivate people to start their jobs via the WINNER system instead of using the traditional Unix tools like `rsh`. Therefore, the node manager is responsible for switching the process priorities of the WINNER processes on its machine accordingly.

System and node managers run as daemon processes. In contrast, job managers are invoked by a WINNER user in order to execute a sequential or parallel job. Thus, job managers are part of WINNER's user interface. Their duties are acquiring resources from the system manager, starting processes on the acquired nodes via the respective node managers, and controlling and possibly redirecting input and output of the started processes. WINNER's simplest job manager is called `wrun`. It allows the execution of a sequential (possibly interactive) job. It works almost like the standard UNIX command `rsh`, except that the job is started on the currently best suited workstation in a cluster.

Because one of its basic design goals is to avoid disturbing the primary (console) users, WINNER provides a user-interface tool by which a primary user can determine how WINNER makes use of his or her workstation: By default, a workstation is only used if the primary user has reached a particular idle time (currently 15 minutes). But using this tool, the user is able to release the local workstation indicating that WINNER can start tasks even if the user is not idle. Alternatively, the primary user can also completely block his or her workstation, such that WINNER will not start any task while the user is logged on until the setting will be changed again.

Additionally, the primary user can request to reduce load caused by WINNER tasks. This is done by WINNER by either migrating processes to different machines, or by interrupting or even aborting WINNER processes in case other methods fail.

3 Load Measurement and Distribution

In order to perform suitable task placement decisions, the system manager must have an accurate global view of the current utilization of the workstations. To achieve this, each node manager provides it with the information related to its own workstation. We will now discuss the duties of both kinds of manager processes.

3.1 Node Manager

On startup, each node manager performs a simple benchmark loop, evaluating the machine's speed in integer operations, floating point calculations, and memory access. This benchmark's result is a

single number proportional to the host's overall relative performance. It is reported to the system manager along with the node name, IP address, and other static data like the total amount of main memory and the number of CPUs on multiprocessor machines.

Afterwards, the node manager regularly queries several load characteristics of its local host and reports them to the system manager if necessary. The set of dynamically changing information consists of the following data:

Load average: The most prominent value is the classical Unix load average. It is calculated by the operating-system kernel by averaging the number of processes in the run queue within a certain time interval.

Run queue length: Unfortunately, most Unix systems only provide very inert load average values, the fastest of which is averaged over the last 60 seconds. As this index changes too slow to be of good use for load distribution decisions, WINNER transforms these values into an index more sensitive to load changes. To understand how this is done, one has to know the way UNIX kernels calculate their load average every i seconds:

$$a_{n+i} = \beta a_n + (1 - \beta) a$$

where a_n is the last load average i seconds ago, a is the current length of the CPU run queue, a_{n+i} is the current load average to be calculated and β is the smoothing factor determining by which the old average a_n contributes to a_{n+i} . Because this smoothing is performed exponentially, a value of $\beta = e^{-i/60}$ is used for the 60 second average.

Neglecting variations of the run queue length a and assuming an interval of 10 seconds between two consecutive load measurements performed by WINNER's node manager, we get an approximation for the average run queue length a out of two consecutive load values:

$$\begin{aligned} a_{n+10} &= \beta^{10/i} a_n + (1 - \beta^{10/i}) a \\ a_{n+10} &= e^{-10/60} a_n + (1 - e^{-10/60}) a \\ a &= \frac{a_{n+10} - e^{-10/60} a_n}{1 - e^{-10/60}} \end{aligned}$$

Using the above equation, WINNER calculates the average run queue length a during the last period. This value reacts much faster to load changes than a_n itself as it is reported by the operating system. This behaviour is shown in Figure 2 in which a (hypothetical) machine with load 0 starts a compute-intensive task at time $t = 0$, yielding a load value of 1 until this task terminates at $t = 60$, which hence resets the load value to 0. The load average as it is computed by the UNIX kernel slowly increases up to a peak value of 0.7 after 60 seconds and then decreases back to 0.21 after 120 seconds. In contrast, WINNER's load average immediately changes its value to 1 and to 0, as soon as it is recomputed after the real load has changed.

CPU states: The relative amount of time the processor(s) spent in one of the different CPU states during the last interval is tightly related to the load average. The states are differentiated into idle time, user time, system time, and nice time. Idle time reflects the amount of time the processor was idle, i.e. executing the system idle thread, whereas user, system, and nice time

reflect the time the processor was executing user code, kernel code, and user code with reduced priority, respectively.

This data is often more accurate in low-load situations than the load average. However, once the CPU idle time is close to zero, one cannot distinguish between situations in which there is either just one or whether there are more processes in the CPU run queue.

User idle time: The presence of a primary (console) user is detected. If such a user is present, the time of his or her last input is queried in order to calculate how long the user has been idle.

Free memory size: The amount of free virtual memory is queried. This information can be used to exclude workstations which are low on virtual memory from the list of available hosts for avoiding situations where a process cannot execute successfully due to memory shortage.

Free disk space: The node manager measures the amount of free disk space for temporary files. This is used as a simple threshold by the system manager for deciding whether or not a host is suitable for a given job.

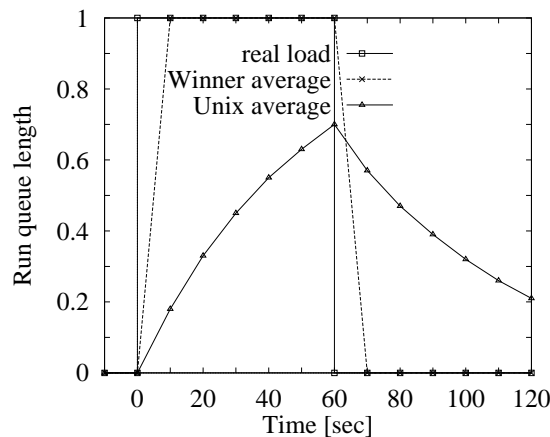


Figure 2: Load averages computed by WINNER and the UNIX kernel.

Finally, the newly collected load information is compared to the latest data set that was sent to the system manager. If these do not differ by more than a certain threshold, the new data is not sent. This policy is used in order to reduce network traffic. However, if no load information has been sent for some time, the data is reported nevertheless for informing the system manager that the node manager is still alive.

3.2 System Manager

The system manager stores the information about its nodes in a table consisting of one entry per host. Each entry contains both the static and dynamic information about this host. Upon reception of new load information from a node manager, the system manager updates the corresponding table entry. Thus the system manager always has a global and current view of all the workstations' utilization.

When a job manager requests a new node for its job, the system manager tries to choose the most appropriate machine. It does so by taking three different aspects into consideration:

First, it checks whether a node is at all available for task execution. This depends on the console user's setting and his or her idle time. E.g. if the console user is active and has not changed the default setting, his/her workstation will not be used.

Second, the node's current load information is checked against the resource requirements possibly specified with the job manager's request. Currently, such requirements can be specified by the user via resource description files or on the job manager's command line. If a node does not fulfill all requirements (like e.g. size of main memory or temporary disk space) it will not be used.

If the node passes these two tests successfully, the processing performance it could provide for the new job is estimated using the following heuristic:

Based on the workstation's base speed S_b (which was measured by the node manager), its current speed S_c is calculated as $S_c = S_b/\lambda$, where λ denotes the fraction of available processing power in the presence of the current workload. Assuming a constant load, λ could be calculated as $\lambda = l_a + 1$ (where l_a is the load average as measured by the node manager). This reflects the fact that after starting a new process, there are λ active processes sharing the CPU.

Because the assumption of constant load is unrealistic, using the load average as it is reported by the kernel yields inaccurate results. E.g. this may be the case when many short running processes cause misleading (high) values for the run queue length. Hence, for achieving more accurate values, WINNER's system manager instead calculates λ by using t_i (the percentage of time the processor was idle): $\lambda = 2 - t_i$, yielding $\lambda \approx 1$ for high percentages of idle time and $\lambda \approx 2$ for higher CPU usage.

In the case of small values of t_i , it can be assumed that the actual workload consists of more than one process. In this case, the load average value should be used for calculating λ in order to reflect the higher load. A threshold of $t_i = 0.15$ is used for choosing between the two cases. This threshold value has been determined empirically as a feasible compromise between possible inaccuracies of both possibilities for computing λ . Its value is hence computed as follows:

$$\lambda = \begin{cases} l_a + 1, & t_i < 0.15 \\ 2 - t_i, & t_i \geq 0.15 \end{cases}$$

Another problem arises when the system manager receives several requests for available hosts within a short time interval. Without further measures, the system manager would choose the same node for all these requests resulting in the fastest host being swamped with new processes. This is due to the fact that it takes some seconds for the selected workstation's load to change and some additional time for this information to be reported back to the system manager. In order to avoid this problem, a bias procedure was introduced to the system manager's calculations which assigns a temporary penalty to recently allocated nodes. This penalty is withdrawn when the system manager receives new load information from the respective node.

Figure 3 shows the output of WINNER's status tool which illustrates a cluster's status based on the system manager's information. On the left side of the status window, the names of the machines are listed. Available machines are printed in black whereas unavailable workstations (with active console users) are printed in green (light grey).

On the left side of Figure 3, the relative speed of the given machines is indicated by the size of the corresponding bars. The total size of a bar corresponds to the statically measured machine speed.

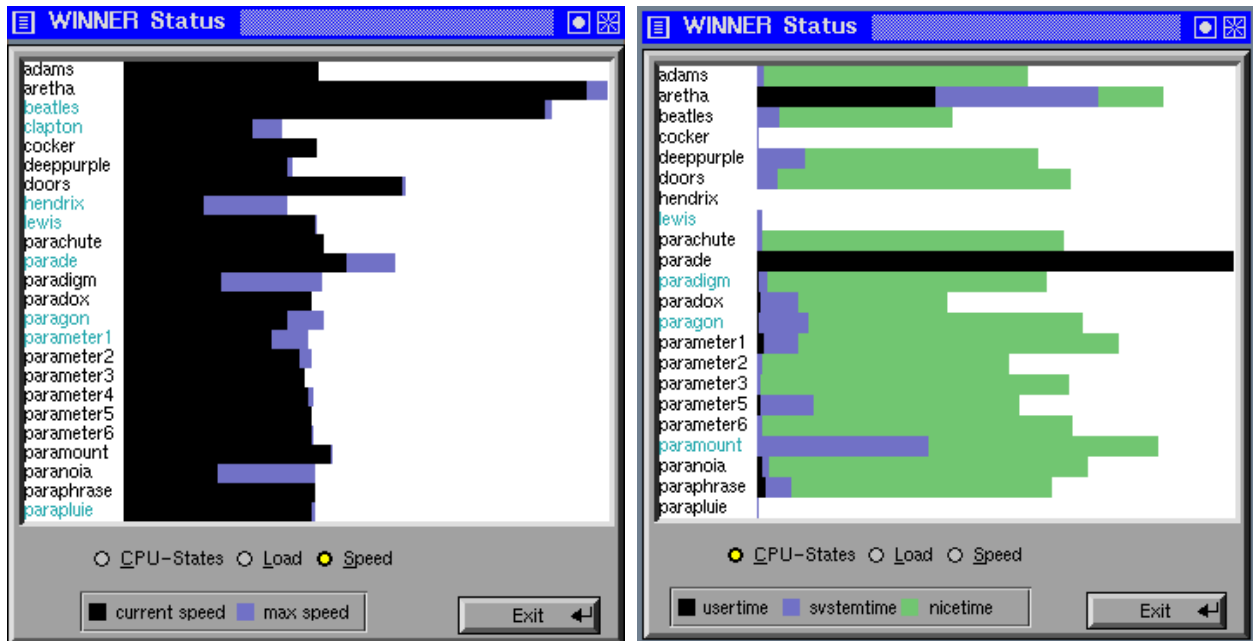


Figure 3: The WINNER status tool, displaying relative speed and processor states.

The black fraction of the bar indicates the currently available fraction of the processor speed ($1/\lambda$). The bar's blue fraction (printed in grey) is currently in use by other processes. In this snapshot, almost all available CPU power is usable for WINNER jobs.

On the right side of Figure 3, the CPU states (of an unrelated snapshot) are shown. For each workstation, the user time fraction is shown in black. System time is shown in blue (dark grey) while nice time (allocated by WINNER jobs) is shown in green (light grey). Finally, white fractions indicate idle CPU time. In this snapshot, almost all machines run WINNER jobs.

4 Automatic Load Distribution for Sequential Tasks

Once a user decides to start a job via WINNER in order to automatically utilize a cluster's spare computing capacity, he or she does so by invoking the proper job manager. The job manager, in turn, queries the local node manager (by contacting it via its well-known port) for the address of the system manager. After being contacted, the system manager determines a suitable target machine (as described in Section 3.2) and authorizes the job manager at the target machine's node manager. The job manager finally requests program execution at the target host.

During program execution, signals received by the job manager are forwarded to the remote process by sending a message to the controlling node manager. Reversely, the node manager informs the job manager about important events regarding the process, like most importantly, its termination. For interactive jobs, standard input and output are redirected to the job manager's terminal. Similarly, the job manager's display is addressed for X-window applications.

In order to reduce the impact on the target machine's console user, the node manager influences the WINNER processes running on its node. Besides the reduction of process priority and temporarily suspension, processes may also be migrated to different hosts. WINNER can do so with

(sequential) processes that are linked with the `libckpt` checkpointing library [10]. In this case, the node manager forces the process to write a checkpoint file which is later used for process restart after the system manager has selected a new target host.

5 Automatic Load Distribution for Parallel PVM Tasks

For parallel applications, WINNER provides a job manager for use with the popular PVM system. This job manager automatically selects a number of appropriate hosts for the PVM virtual machine (usually a tedious task when performed manually), starts the user's parallel application, and improves PVM's scheduling by utilizing WINNER's load distribution mechanism.

In order to avoid changes to the PVM system and to application programs, WINNER follows the approach introduced with the CARMI system [11] for integrating a resource management system by implementing the following specific PVM tasks:

PVM resource manager: One task registers as the resource manager for the virtual machine. It hence intercepts certain PVM calls from other tasks in order to influence the scheduling policy. The resource manager receives requests from other tasks out of which the ones regarding the spawning of new PVM processes are of main interest, because they allow the resource manager to establish its own scheduling policy using WINNER's workload distribution capabilities. This way, PVM's default round-robin scheme can be made obsolete.

PVM hoster: This is a single process (per virtual machine) responsible for starting slave PVM daemons on remote hosts (normally done by PVM itself via `rsh`). This task is necessary when creating or expanding the virtual machine. WINNER registers a hoster process in order to be able to start the PVM daemon processes via the WINNER infrastructure. Hence, the PVM daemons become part of the current WINNER job and can be controlled by the respective node managers.

PVM tasker: Without a dedicated tasker process, new PVM tasks are started directly by the PVM daemon which is running on the respective host. WINNER's PVM job manager registers one tasker on every host that is part of the virtual machine for the current job. This enables the job manager to start newly spawned PVM processes via the WINNER interfaces and hence enables to control them via the node managers.

A user invokes the job manager (`wpvm`) specifying the desired number of hosts for the virtual machine, as well as the name of the user application's master program and its command line parameters. Just like a job manager for sequential jobs, `wpvm` contacts the system manager in order to get assigned as many hosts as the user specified. If possible, the system manager will select the most suitable (e.g. the fastest available) workstations. If not enough machines are available, as many hosts as possible are acquired.

The fastest of these hosts is selected to act as the master host for the virtual machine. `wpvm` spawns the resource manager task on the master host. This task initializes the PVM virtual machine by starting the master PVM daemon and registering itself as the resource manager. In turn, it reports back to the job manager, signaling that the PVM master daemon is up and running.

Next, the hoster task is started on the master host, the set of workstations out of which the VM should be composed of is transmitted to the resource manager, and the virtual machine is expanded to include all these hosts. Finally, one tasker process is started on each workstation.

After all taskers have successfully registered themselves to the VM, `wpvm` spawns the application's master task on the master host. The master task then will presumably spawn some PVM child processes (via `pvm_spawn()`). The task placement decisions for these processes are then automatically redirected to the resource manager which uses `WINNER` for choosing a suitable host for each task.

When all user tasks have terminated, the parallel job has obviously finished. `wpvm` will now shut down the PVM virtual machine, wait for all auxiliary tasks (resource manager, hoster and taskers) to finish, and finally terminate itself.

6 Example: Parallel Knapsack Packing

For illustrating `WINNER`'s effect on application runtimes, a parallel algorithm for solving the *knapsack problem* has been implemented using the PVM system. The knapsack problem is defined as the problem of finding a set of items each with a weight w and a value v in order to maximize the total value while not exceeding a fixed weight limit. In the implemented *divide-and-conquer* algorithm (taken from [2]), the problem for n items is recursively divided into two subproblems for $n - 1$ items, one with the missing item put into the knapsack and one without it.

Whereas the first subproblem is handed over to another processor, the second one is recursively computed within the same node, yielding a dynamically shaped task tree. The assignment of tasks to processors and the related workload distribution is the primary problem of such tree computations. In order to avoid prohibitive runtime overheads, this task tree has to be pruned efficiently. For this purpose, we rely on the heuristic scheme proposed in [2].

Besides efficient pruning, the dynamically created tasks have to be assigned to the available processors. This can directly be done by the PVM system. But because PVM itself has no information about the speed of the individual processors (and the dynamically occupied fractions thereof), PVM typically yields only sub-optimal tasks assignments. Opposed to this, a resource management system like `WINNER` is able to provide the necessary information and thus contributes to much better task assignments.

Figure 4 shows runtime results of the parallel knapsack program for two different problem sizes, comparing task assignment performed by PVM with the assignment performed by `WINNER`. The measurements have been performed on a cluster of Digital AlphaStations, running Digital UNIX 4.0 and PVM 3.3.11. In order to provide a fair comparison, all machines have not been used by other users during the runtime measurements. Hence, the given machines have been fully available to the parallel application and `WINNER` only had to take speed differences into account. Furthermore, the PVM tests used the same machines that had been (automatically) chosen by `WINNER` before. As can be seen from the figure, the task assignments performed by `WINNER` lead to faster program execution with both tested problem sizes and with all three numbers of workstations used for our experimental setup. With additional workload caused by interactive users, even better results for `WINNER` can be expected. The verification of this hypothesis by comprehensive runtime measurements is subject to ongoing work.

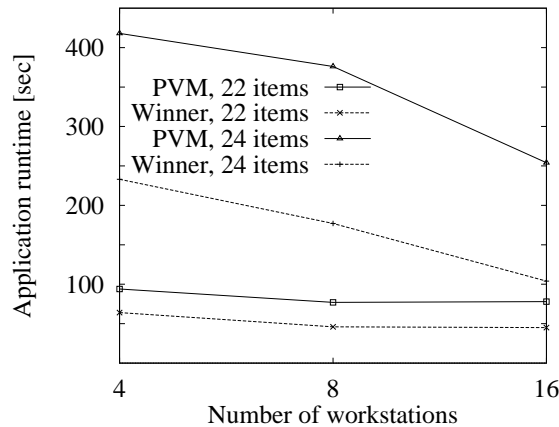


Figure 4: Comparison of task assignment between PVM and WINNER.

7 Related Work

As described above, WINNER's most prominent features include an elaborate and adaptive gathering and evaluation of load characteristics suitable for clusters of workstations with heterogeneous computing speeds. Presence or absence of primary (console) users influence the host selection strategy as well as the execution priorities of active jobs in order to minimize the impact on the console user's work. WINNER's modular structure yields a system which is easily extensible and adaptable to further workstation platforms or kinds of jobs. In the following, we will briefly compare WINNER's features to some related systems.

Piranha [1] provides very good adaptive load distribution for parallel applications. Unfortunately, it only supports a very narrow range of applications, implemented with the Linda programming paradigm. Therefore, it is not suitable for managing general workload.

YALB [14] lacks a proper consideration of console users. Its distributed architecture directly relies on the ability to send low-bandwidth broadcast messages between the participating hosts. Moreover, it does not directly support parallel applications at all.

The Condor system [8] uses a very simple load evaluation strategy for job placement decisions. In fact, it just considers the current load average and the number of active Condor jobs as a threshold. Furthermore it has a very stringent way of reacting to console users (stopping and migrating active jobs). CARMI [11] operates on top of Condor, providing a resource management API to PVM applications. CARMI allows such applications to exploit dynamically changing sets of hosts. However, existing PVM applications have to be rewritten which can be rather tedious as programmers have to explicitly handle the addition and removal of hosts. This problem is somewhat alleviated by the accompanying WoDi library, which provides a comfortable interface for implementing manager/worker applications on top of CARMI.

The Prospero Resource Manager (PRM) [9] lends its modular structure to WINNER. However, PRM's evaluation of utilization information is rather simplistic in that it is only used to decide whether or not a workstation is available for job assignment. Moreover, while a host is assigned to a job, this machine is not available for further jobs. PRM supports PVM applications by providing its own version of PVM, yielding compatibility problems in the advent of future PVM releases.

ALDY [4] is a library of functions providing basic mechanisms and strategies for load distri-

bution of parallel programs. ALDY is not restricted to a single communication platform. Instead it can be adapted by implementing several platform-dependent callback functions, which are automatically invoked by the ALDY scheduler. Application programmers have to define virtual objects along with a mapping to application objects, namely processes. Load balancing is performed by migrating a special kind of active, virtual objects called agents. ALDY determines when an agent should be migrated by considering the number of active agents within a process as its load index. External load information (i.e. memory constraints or active console users) are not taken into account. However, ALDY is useful for balancing the workload of a single parallel application.

Finally, the load balancing extension to PVM introduced in [6] collects real load information which it presents to PVM applications by means of a load vector as well as a dedicated, load balancing spawn function. Its implementation consists of several load daemons, the structure of which resemble WINNER's design. Unlike WINNER, this system is restricted to PVM and therefore only performs load balancing for single PVM applications. In contrast, WINNER works on the middle-ware level and is hence capable of simultaneously scheduling several (sequential as well as parallel) applications without undesirable interferences between their load distribution decisions.

8 Conclusions and Future Work

The WINNER system constitutes a modularly constructed system for dynamic load distribution in workstation cluster environments. It elaborately gathers and evaluates load characteristics and employs this information for adaptive assignments of workload to available machines. Presence or absence of primary (console) users influence the host selection strategy as well as the execution priorities of active jobs in order to minimize the impact on the console user's work.

WINNER's modular structure yields a system which is easily extensible and adaptable to further workstation platforms or kinds of jobs. Currently, WINNER runs on top of Digital UNIX, Linux, and Solaris. Job managers are available for sequential tasks; one for batch execution and another one for interactive programs (including X-Window applications). Furthermore, a parallel version of GNU-make has been developed acting as a job manager for distributing its tasks via WINNER. Finally, a job manager for parallel PVM programs has been presented.

Work on the WINNER project is still in progress. Currently, we investigate the integration of virtual memory utilization into the scheduling algorithm of our node managers in order to further reduce impacts on console users. Other future developments include the introduction of multiple clusters that exchange workload via a queueing system as well as support for shared-memory multiprocessor machines. Additionally, we are working on improvements of our PVM support by providing load information on the application level as introduced by [6], by implementing task assignments based on the ideas of SED-scheduling [13], and we also plan to integrate checkpointing and migration facilities like to ones implemented in the CoCheck system [12]. Finally, we plan to support MPI [5] and Objective Linda [7] as additional parallel programming platforms.

References

- [1] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive Parallelism with Piranha. Technical Report 954, Yale University, Dept. of Computer Science, 1993.

- [2] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.
- [3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.
- [4] C. Gold and T. Schnekenburger. Using the ALDY Load Distribution System for PVM Applications. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Proc. EuroPVM'96*, number 1156 in Lecture Notes in Computer Science, pages 278–287, Munich, Germany, 1996. Springer.
- [5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.
- [6] D. J. Jackson and C. W. Humphres. A simple yet effective load balancing extension to the PVM software system. *Parallel Computing*, 22:1647–1660, 1997.
- [7] T. Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. In *Proc. ICCI'96, 8th International Conference of Computing and Information*, Waterloo, Ontario, Canada, June 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).
- [8] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 104–111. IEEE, 1988.
- [9] B. C. Neuman and S. Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, 6(4):339–355, 1994.
- [10] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proc. Usenix Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, 1995.
- [11] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In D. G. Feitelson and L. Rudolph, editors, *Proc. IPPS'95 Workshop*, number 949 in Lecture Notes in Computer Science, pages 259–278, Santa Barbara, CA, USA, 1995. Springer.
- [12] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In D. G. Feitelson and L. Rudolph, editors, *Proc. IPPS'96 Workshop*, number 1162 in Lecture Notes in Computer Science, pages 140–154, Honolulu, Hawai'i, 1996. Springer.
- [13] B. Schnor and M. Gehrke. Dynamic-SED for Load Balancing of Parallel Applications in Heterogeneous Systems. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, Nevada, USA, 1997.
- [14] S. Stille. Lastbalancierung in verteilten Systemen. Diploma Thesis, Technische Universität Braunschweig, Germany, 1993. (in German).