Computing and Informatics, Vol. 31, 2012, 1255–1278

CYCLIC: A LOCALITY-PRESERVING LOAD-BALANCING ALGORITHM FOR PDES ON SHARED MEMORY MULTIPROCESSORS

Antonio García-Dopico, Antonio Pérez
Santiago Rodríguez, María Isabel García

Department of Computer System Architecture and Technology (DATSI)
Technical University of Madrid
Facultad de Informatica
Campus de Montegancedo s/n, 28660 Madrid, Spain
e-mail: {dopico, aperez, srodri, mgarcia}@fi.upm.es

Communicated by J. C. Cunha

Abstract. This paper presents a new load-balancing algorithm for shared memory multiprocessors that is currently being applied to the parallel simulation of logic circuits, specifically VHDL simulations. The main idea of this load-balancing algorithm is based on the exploitation of the usual characteristics of these simulations, that is, cyclicity and predictability, to obtain a good load balance while preserving the locality of references. This algorithm is useful not only in the area of logic circuit simulation but also in systems presenting a cyclic execution pattern, that is, repetition over time, making the future behavior of the tasks predictable. An example of this is Parallel Discrete Event Simulation (PDES), where several tasks are repeatedly executed in response to certain events. A comparison between the proposed algorithm and other load-balancing algorithms found in the literature reveals consistently better execution times with improvements in both load-balancing and locality of references that can be of help on current multicore desktop computers.

Keywords: Parallel algorithms, shared memory systems, load balancing, locality of references, multicore, VHDL, PDES

1 INTRODUCTION

Digital design needs to describe the functionality of the implemented system. Several approaches are used to ensure that algorithms meet specifications, from the register transfer level (RTL) to the circuit logic level. Hardware Description Languages, such as VHDL (VHSIC Hardware Description Language), are used to describe and test the circuits prior to their actual implementation. Given the growing complexity of current designs, simulation has proven to be a useful tool for checking design functionality. This may involve testing a circuit with a wide range of input data or even subjecting it to stress, the latter often being difficult to perform on a real circuit.

Simulations have to execute as quickly as possible to improve productivity and to allow the designer to put the final product on the market as soon as possible. Current circuit designs are so complex that simulating them in uniprocessor systems is a very time consuming task. Current workstations are multicore processor-based parallel systems that allow the designer to run the VHDL simulator on desktop computers. Simulator parallelization is therefore required if the designer wants to use these architectures efficiently. The most used parallelization techniques in Parallel Discrete Event Simulation (PDES) are shown in [1]:

- Synchronous simulations. There is a global clock for time-measuring and all logical processes tagged with the same time stamp are executed in parallel, but cannot proceed with the next simulation step until all logical processes have completed the previous one. This synchronous behavior requires all processors to synchronize at a barrier before carrying on with the simulation [2].
- Asynchronous simulations. They differ from synchronous approaches in that they remove the barrier synchronizations between time steps in order to allow more parallelism in the simulator, allowing each logical process to execute the simulation by using a local time [3]. Thus a logical process executes events in chronological order until the events queue is empty. This approach allows different parts of the simulation to progress independently. However, it may introduce errors by executing events with time tags previous to already executed events, thereby violating the local causal constraint.

On the other hand, when parallel simulations are executed on shared memory multiprocessors, load balancing and data locality are two of the most important issues to obtain an efficient use of computer resources as well as to minimize execution times. Load balancing avoids idle processors alongside busy processors, whereas preserving data locality avoids moving the same data again and again through the memory hierarchy.

Traditional load balancing techniques do not exploit the main characteristic of PDES applications. The execution of these applications is cyclic and repetitive, i.e., they execute the same code with different data sets in every iteration thus making future behavior predictable.

A new load balancing algorithm for parallel synchronous simulations is proposed in this paper that exploits these features so as to efficiently distribute the workload among the available processors and to try to locate tasks close to their data to improve data locality. This is an important issue in current multicore systems as it helps avoid conflicts in the shared cache and reduces the simulation time on synchronous PDES.

The rest of the paper is organized as follows: Section 2 offers an overview of several load balancing techniques found in the literature. In Section 3, the proposed load balancing algorithm is described in detail, including considerations about data locality. Section 4 contains an example of applying the new algorithm to an unbalanced system. The results that have been obtained are described in Section 5, comparing the proposed algorithm with other well-known load balancing algorithms. Finally, the conclusions of this work are presented in Section 6.

2 LOAD BALANCING AND DATA LOCALITY

After selecting one of the PDES parallelization schemes mentioned in the introduction, processes have to be allocated to processors. This task is extremely important for system performance because if the load of the processors is not balanced, there will be processors with a lot of work pending while others remain idle waiting for the former to finish. However, this task must be accomplished without neglecting the locality of references, which has a major influence on execution time.

Load balancing has also been discussed in the context of scheduling for distributed systems. In centralized scheduling, the scheduling task resides in one processor of the system. In distributed scheduling, the task of allocating processes to processors is physically distributed among the processors. In both approaches process migration is a complex task, except in shared memory multiprocessors, where the cost is negligible.

Static load balancing is the simplest algorithm and has been used for loop parallelization. In this case, the loop is distributed among the available processors assuming that the load of every iteration is similar. This algorithm is easy to implement and there is no contention to obtain more work.

Static scheduling has many disadvantages as stated in [4]. Usually the amount of computation of every iteration is not the same and the load of the processors may vary with time. Data references are irregular and dynamic and, in centralized scheduling, data locality does not take into account data movement among processors. The overall system load may vary and is not predictable.

Dynamic scheduling offers an alternative avoiding load imbalances, but it has to be implemented taking into account the computation involved in scheduling. This has even led some authors to suggest using a processor fully dedicated to scheduling [5]. In dynamic scheduling, work is assigned to every processor in the system. When the work is completed the processor requests more work. In centralized approaches, a processor is dedicated to executing a scheduling function and the tasks are assigned to processors by overloading a processor in the system which in many cases is completely dedicated to executing the load balance function.

Distributed scheduling is an alternative option to avoid synchronization with the centralized scheduler: the processors monitor and update the whole system load information. The least loaded processor steals work from the most loaded ones. In [6] a toolkit for dynamic load balancing is presented that implements a well-known algorithm for distributed scheduling. In [7, 8] the adaptive factoring algorithm dynamically estimates the means and variances of loop iteration execution times and uses a probabilistic model to dynamically allocate chunks of loop iterations to a processor.

When a process is migrated, data have to be maintained as close as possible to the processes that use the data [9]. If data locality is not preserved, communication overheads are added when accessing data, and the benefits obtained by load balancing are lost. In [4] data chunks are assigned to tasks which are in turn assigned to processors. When a processor needs to steal work a task within that processor's boundary is selected (i.e. close to its data).

Symmetric multiprocessors (SMPs) are nowadays very popular in the area of high performance computing in stand-alone systems. Commodity hardware is improving based on deeper memory hierarchies and chip parallelism, turning current personal computers into symmetric multiprocessors. In the last years, each processor has itself become a multiprocessor by introducing several cores into the processor. The main difference with respect to previous SMPs is that the cores usually share the cache levels above L1, changing the behavior of the memory hierarchy. All these improvements allow high performance applications to be executed on a desktop computer; however, several changes have to be made in these applications and in the operating systems [10, 11]. Parallel simulators can also make use of these improvements, but they have to be modified. SimK [12, 13] is a parallel simulation engine that uses high efficiency synchronization techniques and dynamic task migration. It implements work stealing, and processor affinity is considered for task migration by using CPU affinity system calls.

Traditionally, scheduling in SMPs was done using a single, globally accessible task queue and every process could be executed in every processor in the system. As this scheduling ignores data locality, a collection of local task queues with a simple load balancing scheme was already proposed in [14]. Task migrations in these systems are cheaper than in distributed systems, but their cost in terms of data locality is known [15]. However, always executing the same tasks on the same processors does not in itself guarantee good performance and data distribution must be considered to improve the use of the memory hierarchy. Effective use of memory hierarchy is important and some schedulers use processor-cache affinity information to avoid cache-reloads and to improve the performance of local task queues as in [16]. [17] uses a combination of a global queue and several local queues, known as hybrid queues, to exploit the advantages of each scheme, where the "n" first tasks associated to each processor are located in its local queue, and the rest of the tasks are located in a global queue. These hybrid queues can be combined with a dynamic threshold, where the threshold changes according to the amount of pending work [17], as it is impossible to estimate a unique and optimum value for "n". In SimK [12, 13] two separate lists per processor are used; the run_list and the mig_list where tasks are moved when the length of the run_list reaches a certain threshold. If there are tasks in the run_list they are dispatched, otherwise the mig_list is consulted for candidates. If the mig_list is empty, work stealing occurs from remote mig_lists. Some scheduling systems consider the negative effect of the memory hierarchy negligible, e.g. in real-time system schedulers, as the main objective is for the tasks to meet their deadlines while system performance is secondary [18]. Also several attempts have been made at data partitioning in order to increase data locality [19], e.g. by minimizing data region sharing among processors or by modifying the scheduler of the operating system to obtain better performance through reduced memory contention [20, 21, 22].

Load balancing and locality of references have a major impact on the execution time of PDES in SMPs, however improving the one tends to worsen the other [23]. Vee and Hsu [24] study a similar problem to the one exposed in this paper: load balancing in synchronous simulations on SMPs. Their approach is to solve load imbalance by applying work stealing between processors. However, this results in high contention in the bus due to the fine grain of parallelism. Another implementation for general simulations is proposed in Cilk [25]. This runtime system is designed to exploit dynamic, highly asynchronous parallelism, focusing on load balancing, communication protocols and data locality.

When loop scheduling is considered different algorithms are also used. [26] compares five different loop scheduling algorithms: static scheduling, self scheduling, uniform-size chunking, guided self-scheduling, and affinity scheduling. The latter considers the loop scheduling problem to have three dimensions: load imbalance, synchronization overhead, and communication overhead due to nonlocal memory accesses.

All these scheduling policies have to be combined with a second level of scheduling when jobs share a computer, i.e. when several jobs run simultaneously and the scheduler must distribute the jobs among the processors, each job with several threads [27].

3 CYCLIC: A NEW LOAD BALANCING ALGORITHM

In this section, a new load balancing algorithm aimed at synchronous PDES is presented which obtains very promising results in terms of load balancing and enforces the locality of references.

To measure the improvement this algorithm will be compared in Section 5 with the four algorithms referred in the previous section: the global queue, the local queues, the hybrid queues, and the hybrid queues with a dynamic threshold.

As this algorithm was initially designed to be applied to logic circuit simulations, the main characteristics of these simulations were exploited. Logic circuit simulation involves components (gates, multiplexors, registers, etc.) and input-output logic signals. The components are sensitive to the changes produced in the values of their input signals, that is, when their input signals change they react by executing and recalculating the values of their output signals. VHDL has been chosen as an extended, well-known and standard hardware description language widely used in industrial and academic environments. In VHDL the components are simulated using VHDL processes and the signals are implemented using VHDL signals, VHDL processes and signals being the only tasks of the simulation. The simulation algorithm consists of two phases. In the first phase, the components are executed to obtain a new value for their output signals taking into account the values of their input signals. In the second phase, the values of the changed signals are updated and propagated. This fact introduces new changes at the input of other components, causing the simulation cycle to start again.

This cyclic feature, which allows for the prediction of future behavior, and the fact that the time employed in updating a VHDL signal or executing a component (VHDL process) is usually the same for every iteration, allow estimating the workload of each phase in advance. This time is constant for two reasons:

1. the steps followed by the simulation kernel to update a signal are always the same and
2. the steps to compute the output signals from the input signals in a VHDL process are usually the same and do not involve a lot of user code per VHDL process, even in the case of behavioral models.

These features help better distribute the load among the available processors.

The workload of each phase can be determined because all active tasks (VHDL signals and processes) are known at the beginning of each cycle. All that is left to do is to assess the granularity of each task, that is, measuring the time employed by each component or by each signal. Thus, before starting each phase, the active tasks are known, as well as their previous execution times. In this way, the total amount of work for each phase can be estimated allowing the workload to be balanced between the different processors before starting the execution of the phase. This effectively results in an adaptive algorithm that balances the current workload in every phase.

Moreover, the proposed workload distribution algorithm takes into account the locality of references and seeks a reasonable trade-off between both factors: good workload distribution and good locality of references.

To obtain the above-mentioned trade-off, each processor has its own queue with pending work, employing a system which only steals from those queues with unbalanced work. To determine the amount of pending work, each local queue has a counter showing the total estimated work present in the queue.

Every time a task is activated, it is incorporated into the queue of the processor that executed it previously to obtain a good locality of references. Before starting each phase, once the total workload and the workload assigned to each processor are known, only the minimum necessary tasks are migrated to obtain a good load balance.

Moreover, by using only local queues for each processor, the contention that could arise by accessing a global queue is totally suppressed, because each processor inserts and removes tasks from its local queue without any need for mutual exclusion to manage the global queue, since load balancing and task stealing are done at the end of each phase at the synchronization barrier.

At the beginning all the tasks are placed in a global queue and the processors obtain the work to do from this global queue. Later, each task is placed in the local queue of the processor that executed it previously. This ensures a good initial load distribution using local queues, to improve the locality of references. As the cache memories do not initially contain any footprints, the penalty of using a global queue in the first iteration is minimal.

In Section 3.1 the load balancing algorithm, based on the cyclicity and predictability of every simulation iteration, is explained. Later in Section 3.2, some changes are included in the tasks, i.e. VHDL processes and VHDL signals, to make better use of the memory hierarchy and improve data locality.

3.1 Description of the Algorithm

To balance the whole system the scheduling algorithm is executed for every VHDL simulation phase by the last task that arrives at the synchronization barrier.

During the first iterations of the algorithm (initialization), the execution time of each task is measured several times. To avoid taking more measures than necessary, this measure is only taken the first "n" times the task is activated, and the average time is computed. The first two measures taken during initialization are discarded, since they do not follow the same pattern as the following executions. Spurious measures, i.e. measures that are at a variance with the average values, are also discarded; this variance is usually due to CPU preemption during execution. After the initialization the amount of work assigned to each processor can be estimated as the granularity of each task is known.

Once the initialization has been completed, the workload is balanced every time the last task reaches the synchronization barrier. The main objective of the load balancing algorithm is to balance the workload with minimal migration to improve data locality. The algorithm steps are as follows:

1. The least busy CPU is determined.
2. The busiest CPU is determined.
3. `unbalanced_work` is determined by:

$$\sum_{i=1}^{n\_cpus} (\text{work\_of\_CPU}_i - \text{work\_of\_least\_busy\_CPU}) \tag{1}$$

that is, the sum of all the pending work to do once the least busy processor has finished.

4. The amount of work to migrate is computed as

$$work\_to\_steal = unbalanced\_work / number\_of\_CPUs$$

The idea behind this division is that the least busy CPU should do its fair share of the unbalanced work.

5. The algorithm avoids migrating more tasks than necessary in order to preserve the locality of references, i.e. to reuse the data present in the cache memory of the processor where the task was previously executed. To do so, the algorithm ensures that the least busy CPU does not end up with more work than the busiest one, i.e., the workload of the least busy CPU plus the workload of the migrated tasks does not exceed the final workload of the busiest one. If this is the case, the amount of work to migrate is recomputed as maximum workload minus the sum of the minimum workload plus the previous amount of work to migrate.

```
if ((least_busy_CPU+work_to_steal) > (busiest_CPU-work_to_steal))
   work_to_steal = busiest_CPU - (least_busy_CPU+work_to_steal);
```

Figure 1 shows a situation where the above-mentioned problem appears. The amount of migrated work is too big and the least busy CPU could end up with more work than the busiest one.

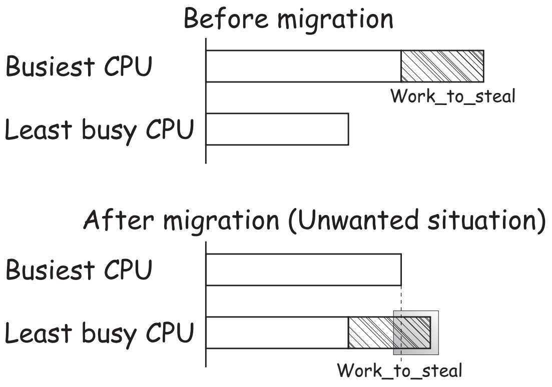

Fig. 1. Incorrect task migration

6. Tasks are migrated from the busiest CPU to the least busy CPU while the amount of work migrated is less than or equal to `work_to_steal`.
7. If all the tasks in the queue of the busiest CPU have an execution time greater than `work_to_steal`, the smallest one is migrated, as long as this migration

does not increase the imbalance, that is, the workload of the least busy CPU plus the workload of migrated tasks does not exceed the previous workload of the busiest CPU.

8. The previous steps are repeated in the following iterations distributing any unbalanced work between the other CPUs until the system is totally balanced, i.e., as long as any task has been migrated the system reverts to step 1. The load balancing algorithm ends only when no further task can be migrated because the imbalance would increase, i.e., the system is balanced.

Each iteration reduces the load imbalance as the least busy CPU will always have a lower workload than the busiest CPU at the end of each iteration. Thus, the algorithm converges until there are no tasks left to move and the load distribution cannot be improved any further.

This algorithm introduces little overhead, because the time taken to update the component or the signal is only measured the first "n" times. It also spends time on the balancing process, but as the simulation advances the initial imbalance decreases, because each task is assigned to the queue of the processor where it was previously executed, which also decreases the time needed to balance the system. Thus, the number of tasks that have to be migrated to obtain a good balance decrease over time. As the algorithm quickly converges to a load-balanced state, and the VHDL simulations usually take thousands or even millions of iterations to simulate the digital circuit, this minor overhead is negligible.

3.2 Data Locality

As opposed to other load balancing algorithms, such as those used by SimK or other VHDL simulators, data locality has been improved in the code generation phase, when translating the VHDL code to C code. This has been achieved by introducing additional changes, most of which are easy to implement and make far better use of the memory hierarchy. These changes pursue two objectives: the first is to avoid false sharing of internal data between different tasks and the second is to increase proximity between dependent internal data of a task.

False sharing occurs when two independent tasks without any shared data have some of their data located in the same cache block, which causes constant invalidations and unnecessary reloads every time one of these tasks updates its data.

Regarding data locality, the objective is to group related data as closely as possible to increase proximity in such a way that when a cache miss or a page fault is triggered, any additional data needed in the close future are obtained together with the required data.

Internal data are not the only problem, shared data between different tasks also need to be considered. VHDL signals have been grouped by their dependencies, as will be explained in Section 3.2.2, and therefore there are no shared data between these signals groups. VHDL processes do not share data because the language definition does not allow direct communication between processes, and communication must always be through the VHDL signals.

Thus, the only shared data are the signals read and written by each VHDL process. The read signals do not present any problem because the other copies present in the memory hierarchy do not have to be invalidated. The written signals have been modified to reduce the penalties caused by the memory hierarchy. Details are explained in Section 3.2.2.

The parallel VHDL simulator has been written in C language, with each VHDL process supported by a thread of the simulator, and several threads – one for each processor of the machine – supporting the simulation kernel which updates the VHDL signals.

Each VHDL process has its own source code, provided by the user, and it can call procedures and functions to describe the behavior of the component being simulated. This fact implies that each VHDL process needs its own stack and therefore a different thread has been used to implement each VHDL process. The size of the components being simulated depends on the abstraction level used by the designer and can vary from gates, multiplexors or registers to a whole CPU.

VHDL signals are a different case because unlike VHDL processes they do not have any associated user code, as they represent input-output logic signals. This means that VHDL signals do not need to have an individual thread with its own stack to execute the user code as VHDL processes do. In fact, VHDL signals are updated by the simulation kernel threads. Thus, a kernel thread can update several VHDL signals.

The simulator therefore has two different types of tasks, VHDL processes and VHDL signals. All the tasks are executed by threads. The next section describes the changes introduced, both in the VHDL processes and in the signals, to improve data locality.

3.2.1 VHDL Processes

Each VHDL process has its own source code with its own variables, local to the process. These variables have been grouped as closely as possible and separated from the rest of the variables owned by other processes. For this purpose, they have been generated as variables local to the thread of the simulator executing the code of the mentioned VHDL process.

To ensure space locality, during the generation phase, i.e., when the VHDL code is translated to C, the variables are grouped close together in the stack of the thread, and clearly separated from the local variables of other VHDL processes, which are located in the stack of their threads, thereby avoiding false sharing as the variables will not be located in the same cache line.

Together with these variables there are frequently used dynamic data, used to dynamically implement the VHDL data structures in the C simulator that require a different treatment. In this case, a heap has been assigned to each thread, thus differentiating the proposed simulator from other simulators. Again, a similar effect has been obtained, increasing the locality of references and avoiding false sharing. All the dynamic variables of a VHDL process are close together in the heap of the thread, and separated from the dynamic variables of other VHDL processes which are located in the heap of their threads. As the dynamic data are used to implement the VHDL data structures that are built at the beginning, during the first two iterations (where measures are not taken), the memory allocation does not introduce time unpredictability in the task execution time.

An added advantage is that the contention due to the dynamic memory management is avoided because all the requests for assigning or freeing memory are made local to the heap of each thread. This contention can be significant in parallel applications because current operating systems usually use a single list per process to manage dynamic memory and access to that list could give rise to contention, converting the dynamic memory management into a bottleneck.

Thus, the variables of each VHDL process are located in the stack of their thread, while the data structures are located in the heap of their thread, increasing the locality of references and avoiding false sharing.

3.2.2 VHDL Signals

VHDL signals are a different case because they do not have any associated user code and they are updated by the simulation kernel threads instead of having their own thread. Moreover, VHDL signals display mutual dependencies and they usually share data. However, this is not a problem as it does not constitute false sharing.

The first approach to data locality is to treat dependent signals together to avoid several threads updating them simultaneously, which would require accessing the shared data in mutual exclusion. This has to be avoided as it increases the contention and invalidations among the different cache blocks that share the same data.

The signals have been grouped according to their dependencies into so-called signal or activity groups. These groups are defined by the fact that any change in any signal of a group forces to recompute all the signal values of the group, activating all the signals of the group and the group as a whole, i.e., the signals of a group are always activated in the same time-step. Activating all the signals in a group does not mean that they all change but that all of them have to be recomputed.

Activity groups represent the minimum working unit for the kernel threads. Not considering the signal as the minimum working unit is another feature that sets CYCLIC apart from other VHDL simulators. To update the VHDL signals affected by changes, each kernel thread selects an activity group, updates all its signals (all of which are active) and moves on to the next activity group once finished.

VHDL signals can present the following dependencies that have to be considered when grouping them. Their dependencies are important to avoid synchronizations between threads, to avoid false sharing and to increase spatial locality.

- Composed signals. There are no dependencies between their elements. A change in the value of one of its elements does not imply a change in the value of the others. Thus, recomputing is not required and the subelements are not grouped.
- Connected signals. If several signals are connected through ports they are dependent, thus they should be together in the same group in order to propagate their values correctly.
- Implicit signals. If the value of a signal varies, all the implicit signals that use it as a reference (stable, quiet, transaction, delayed) should be recomputed because their value may change. Therefore, they should be in the same group as their reference signal.
- Solved signals. Those are signals which several processes write to. VHDL introduces a data structure called driver for each VHDL process that writes to a VHDL signal. The final driving value of the signal is obtained using a resolution function. Solved signals can be of two types:
  - scalar, that is, they are not composed. There are no dependencies between signals but there are several processes writing to one signal (each process in a different driver). Thus, there are several drivers but a single signal. The solution here is to group the drivers with their signal. The activity search is done using the signal groups that include all the interrelated signals and their drivers.
  - composed. If the signal is solved at element level, the case is similar to an unsolved composed signal, as there are no dependencies between subelements of the signal. If it is solved at composed signal level, the resolution function will return a new value for the "n" bits, requiring all the elements of the signal to be computed again. Since all of them are mutually dependent they should be together in the same group.
- Signals connected through type conversion functions. This case is similar to the above-mentioned solved signals that use resolution functions, but in this case type conversion functions are used to obtain the driving value of the signal.
  - The signal is scalar: it is equivalent to the connected signals.
  - The signal is composed: the situation is the same as with the composed solved signals. Modification of the input value forces calling of the conversion function, which will return a new input value for the "n" bits, forcing to compute all the subelements of the composed signal again. As all the subelements are affected, they should be together in the same group.

These activity groups are built during the code generation phase, that is, when the dependencies between signals in the elaborated model are analyzed. To improve the spatial locality of references, signals belonging to the same group are generated contiguously, to take into account the fact that when one is updated, the rest also have to be updated. To avoid false sharing, the groups are generated aligned to the cache memory block size. In this way, although some memory is wasted, independent data cannot share the same cache block.

To improve memory hierarchy behavior the authors took a closer look at the data shared between the signals and the processes that write to these signals. It should be noted that a process does not modify the signal, but the driver that is associated to both. The driver of each group is generated in its own memory block and all the drivers are linked with their group by means of pointers. In this way, when several VHDL processes write new signal values into a group concurrently, the whole group does not move across the memory hierarchy and only few memory blocks are updated or invalidated. This considerably reduces the overall traffic as the signals group is not moved at all. Instead only the drivers move across the cache memories.

3.3 Input/Output Monitor

An input/output monitor has been included to decouple the input/output operations from the rest of the simulation and avoid having all the tasks waiting in the synchronization barrier for a task that is performing a read or write operation. This is to prevent the simulation tasks from accessing the disk for read or write operations. As with all synchronous simulators with synchronization barriers at the end of each phase, if a task has to read from or write to the disk, the rest of the tasks have to wait in the synchronization barrier for it to finish the I/O operation before continuing with the simulation, introducing an unbalanced situation in a previously balanced simulator. The scheduling algorithm cannot predict the task input/output operations, the only way to maintain the whole system balanced is preventing the tasks from accessing the disk.

The I/O monitor takes care of both tasks: prefetching to store the data required for the tasks in memory and data dumping to write to the disk. The I/O monitor implementation takes into account that all the file descriptors have an associated data buffer, so buffered I/O is used. The tasks read or write data directly to the buffer. They access the file descriptor in mutual exclusion and they read or write the data directly to the buffer associated to the descriptor. It is a quick and simple operation, leaving disk access to the monitor.

The I/O monitor is suspended until a request is received, whereupon it wakes up and takes care of the request. Once all the pending requests have been dealt with, it examines all the file descriptors, looking for buffers that are near their limit (the limit value is 75 % of the buffer size, that is, if 75 % of the buffer has already been read or written, a monitor request is generated). If one of the buffers is being used above 50 % it is considered near the limit and the monitor reads or writes its data before suspending.

4 EXAMPLE

The best way to show how the CYCLIC algorithm works is applying it to an example with an initially unbalanced workload. The example is based on a computer with four processors, two of them heavily loaded and the other two with a very light workload.

For every task to execute, the time spent on executing is estimated. The initial situation is shown in Figure 2.

- CPU1: 1 977 μs; CPU2: 579 μs; CPU3: 1 591 μs; CPU4: 330 μs.

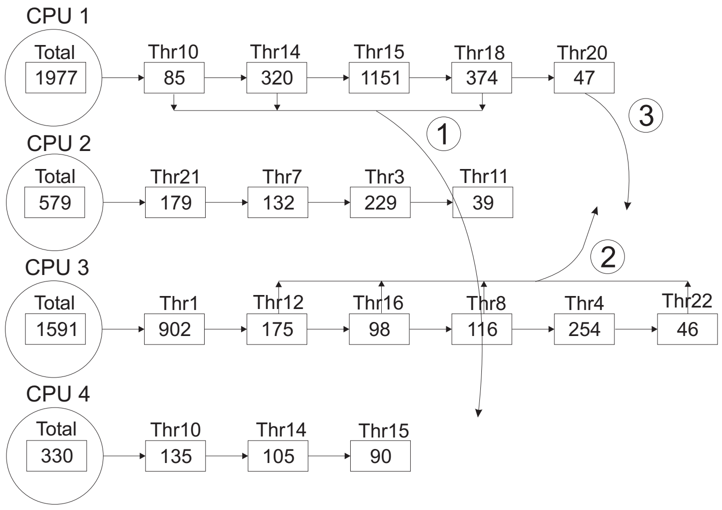

Fig. 2. Migrating tasks to balance the workload

The busiest processor is CPU1, with 1 977 μs, and the least busy processor is CPU4, with 330 μs. The amount of work to balance is:

$$\text{unbalanced\_work} = (1\,977 - 330) + (579 - 330) + (1\,591 - 330) + (330 - 330) = 3\,157.$$

Hence, the amount of work to migrate from CPU1 to CPU4 is $3\,157/4 = 789\,\mu$s.

The local queue of CPU1 is searched to migrate tasks to the queue of CPU4 without exceeding 789 μs of migrated work.

The first task in the queue, Thr10, has an execution time of 85 μs, therefore it can be migrated. The next one, Thr14 (320 μs), is also migrated. The third one (1 151 μs) cannot be migrated because the above mentioned 789 μs would be exceeded. The fourth one, Thr18 (374 μs), is also migrated. The last task, Thr20 (47 μs) is not migrated as it exceeds the limit of 789 μs. Thus, a total workload of 779 μs has been migrated from CPU1 to CPU4. This is shown by the arrow labeled "1" in Figure 2.

The new situation after migrating is:

- CPU1: 1 198 μs; CPU2: 579 μs; CPU3: 1 591 μs; CPU4: 1 109 μs.

Now, the busiest processor is CPU3, with 1 591 μs, and the least busy processor is CPU2, with 579 μs. The amount of work to balance is:

$$\text{unbalanced\_work} = (1\,198-579)+(579-579)+(1\,591-579)+(1\,109-579) = 2\,161.$$

Hence, the amount of work to migrate from CPU3 to CPU2 is $2\,161/4 = 540\,\mu$s. If this amount of work were migrated, CPU3 would be less busy than CPU2 ($579 + 540 > 1\,591 - 540$). As avoiding the migration of tasks (if possible) is preferable for the sake of preserving the locality of references, the amount of work to migrate is the maximum workload minus the sum of the minimum workload and the amount of work to migrate: $1\,591 - (579 + 540) = 472\,\mu$s.

The queue of CPU3 is searched, migrating tasks to the queue of CPU2 without exceeding 472 μs of migrated work. The first task in the queue, Thr1 (902 μs) cannot be migrated because it would exceed the limit. Thr12 (175 μs), Thr16 (98 μs) and Thr8 (116 μs) are migrated because their execution times fit the limit. Thr4 (254 μs) cannot be migrated without exceeding the limit, whereas Thr22 (46 μs) can. The four tasks with a total workload of 435 μs are migrated as shown by the arrow labeled "2" in Figure 2. The new situation is as follows:

- CPU1: 1 198 μs; CPU2: 1 014 μs; CPU3: 1 156 μs; CPU4: 1 109 μs.

Again the busiest processor is CPU1 with 1 198 μs, and the least busy processor is CPU2 with 1 014 μs. The amount of work to balance is 421 μs.

The amount of work to migrate is now $421/4 = 105\,\mu$s. Again this would leave CPU1 less busy than CPU2 ($1\,014 + 105 > 1\,198 - 105$). To avoid this, the work to migrate is recomputed $(1\,198 - 1\,014) - 105 = 79$.

The queue of CPU1 is searched and Thr20 (47 μs) is migrated as shown by the arrow labeled "3" in Figure 2. Now the situation is:

- CPU1: 1 151 μs; CPU2: 1 061 μs; CPU3: 1 156 μs; CPU4: 1 109 μs.

At this stage, the busiest processor is CPU3 with 1 156 μs, and the least busy processor is CPU2, with 1 061 μs. The amount of work to balance now is 233 μs, therefore the amount of work to migrate is 58 μs. Again, this would leave CPU3 less busy than CPU2. Hence, the recomputed work to migrate is 37 μs.

Searching the queue of CPU3, Thr1 (902 μs) and Thr4 (254 μs) are found. Both exceed the limit, leaving the smaller of the two, Thr4, as a possible candidate for migration. If Thr4 were migrated, the load imbalance would rise because
`max_work (1156)` < `min_work(1061)+min_to_steal(254)`.

The algorithm ends as it cannot move any further tasks without increasing the imbalance, i.e., the system is fully balanced. The final result can be seen in Figure 3. The speedup is computed as the total amount of work to do divided by the busiest processor.

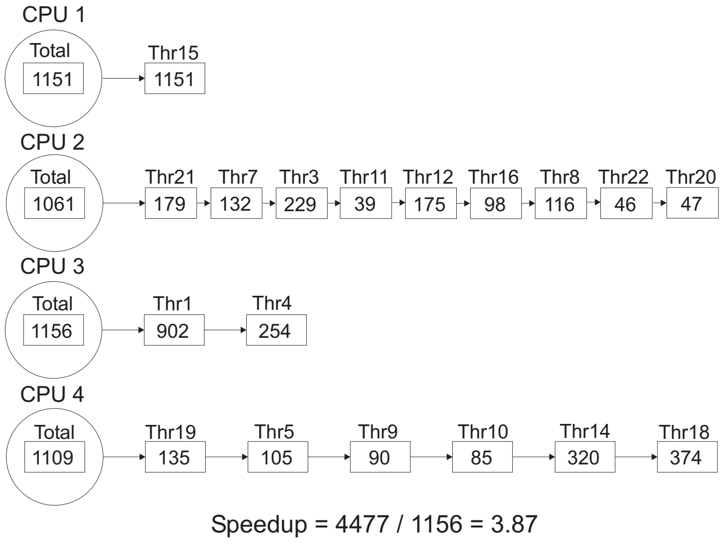

Fig. 3. Load distribution obtained with the CYCLIC algorithm

5 RESULTS

The CYCLIC load-balancing algorithm was performance-tested without task stealing or ordered queues and was compared with four other well-known existing load-balancing algorithms to estimate the benefits obtained. The following algorithms were compared:

Global queue. A single central queue is used which all the processors access to obtain work. This algorithm provides a quasi-optimal load balance, but it ignores the affinity or locality of references. Moreover, it presents contention in accessing the queue, and it is the most used algorithm in thread-based systems.

Local queues. Here, the opposite is true: each processor has its own local queue and the tasks always execute on the same processor, that is, they never migrate. This ensures a very good locality of references but poor load balance.

Hybrid queues. This is a trade-off between the two previous algorithms, with both local queues and a global queue. There is one local queue per processor, where the "n" first tasks associated to each processor are located, and a global queue that contains the rest of the tasks. If a processor completes its local tasks it checks the global queue for further tasks. This algorithm reduces contention when accessing the global queue, increases the locality of references (because a lot of tasks are executed on the same processor as before), and balances the workload with the tasks present in the global queue, avoiding idle processors once they have finished their local work. The problem of this algorithm is to estimate the optimum value for "n".

Hybrid queues with dynamic threshold. In this algorithm, the threshold "n" is variable; it changes according to the amount of pending work.

CYCLIC. As mentioned before, this algorithm is based on one local queue per processor, locating each task in the queue of the processor where it was previously executed. Mainly, this guarantees a good locality of references. To balance the workload, threads are migrated between queues. To determine the amount of work to migrate, the processing time of each component is estimated. Moreover, there is no contention due to the absence of a global queue.

In order to compare the performance of the above-mentioned algorithms, the following commercial register transfer level (RTL) type designs were selected: `cpu_rtl`, `fpa`, `lsi1`, `lsi2`, and `cfg_pid`. The following is a description of the designs used:

- `Cpu_rtl` implements a processor at RTL with a floating point unit and a memory bank. The processor is used for multiplying two matrices. This design consists of 8 processes and 179 signals that generate 452 signal groups.
- `Fpa` implements a floating point adder, which adds double precision floating point numbers represented according to the IEEE754 standard, normalizing the result. It detects and manages the various situations that can arise: overflow, infinities, etc. This design, described at RTL level, has 91 processes and 115 signals that generate 1 735 groups.
- `Lsi1` generates files with input stimuli from a file and a logic vector. It has 61 processes and 191 signals that generate 749 groups.
- `Lsi2` generates files with input stimuli, similar to the previous one, from a file and a logic vector, but it is a larger version. It has 317 processes and 537 signals that generate 5 424 groups.
- `Cfg_pid` implements a proportional-integral-derivative (PID) controller at RTL level. For this purpose it uses 321 processes and 3 116 signals, generating 2 290 groups.

Multiple simulations of these designs were run changing only the load-balancing algorithm. Measuring was performed by executing the simulations on a personal computer using an `Intel Core 2 Quad Q8200` processor (i.e. with four cores), with 4 096 MB of shared memory, a clock frequency of 2.33 GHz and `Linux Ubuntu 10.04 Lucid Lynx` operating system. For every design and load-balancing algorithm, several simulations were performed discarding spurious data and computing the average value. As the measures were taken without other processes executing in the computer, all the measures were quite similar, with very low variation. All simulations were carried out with several thousands of iterations, as can be seen in Figures 4 and 5, as VHDL simulations usually take thousands or even millions of iterations to simulate the digital circuit. Thus, the overhead of the initialization cycles is negligible, as the simulation quickly converges to a load-balanced state and the number of times the task execution time is measured is small ("n'' = 5), discarding spurious data.

Table 1 shows the simulation times in seconds of the sequential and parallel versions, obtained with CYCLIC as well as the speedup.

Design	Sequential version	Parallel version	Speed up
lsi2	42.1	22.6	1.86
lsi1	13.8	11.1	1.24
cpu_rtl	33.4	28.8	1.16
fpa	21.7	14.65	1.48
cfg_pid	94.2	42.3	2.21

Table 1. Simulation times of the sequential and parallel version

Table 2 shows the simulation times of the different designs in seconds and Table 3 shows the improvement of CYCLIC. The traces resulting from the executed simulations with CYCLIC algorithm show very effective load-balancing, except for the first simulation phases in which, due to the low number of measurements, the estimated granularity is not very accurate. In fact, as the simulation progresses, the workload distribution becomes more and more balanced and the number of tasks that need to be moved is reduced to a minimum. Figure 4 shows that a balanced status is quickly reached after which the number of migrated tasks is very small.

Design	Global queue	Local queues	Hybrid queues	Dynamic threshold	CYCLIC
lsi2	23.8	24.5	24.0	24.3	22.6
lsi1	11.6	11.8	11.6	11.2	11.1
cpu_rtl	29.8	29.4	29.1	29.9	28.8
fpa	15.75	15.35	15.5	15.55	14.65
cfg_pid	47.25	45.6	45.6	43.6	42.3

Table 2. Simulation time of VHDL designs with different load balancing algorithms

Design	Global queue	Local queues	Hybrid queues	Dynamic threshold
lsi2	5.3 %	8.4 %	6.2 %	7.5 %
lsi1	4.5 %	6.3 %	4.5 %	0.9 %
cpu_rtl	3.5 %	2.1 %	1.0 %	3.8 %
fpa	7.5 %	4.8 %	5.8 %	6.1 %
cfg_pid	11.7 %	7.8 %	7.8 %	3.1 %
average	6.5 **%**	5.9 **%**	5.1 **%**	4.3 **%**

Table 3. Improvement obtained with the CYCLIC algorithm

Analyzing the results of Table 3, some conclusions can be drawn:

- Both the global and the local queue algorithms show poor performance. This is because the former ignores the affinity between tasks and processors (locality of references) and the latter provides a poor load balance [28].
- The behavior of the two hybrid algorithms is slightly better than that of the global and local queues. Although at the beginning the algorithm with the dynamic threshold seemed to show better performance, it does not clearly outdo the algorithm with the static threshold.
- The proposed algorithm behaves much better than the others. This is because it takes advantage of the digital circuit simulation characteristics: cyclicity and predictability.
- There are two designs where the improvement of CYCLIC is rather small, `lsi1` and `cpu_rtl`. This is due to the implementation of both designs. Neither has exploited the concurrency that VHDL language offers and they are pseudo-sequential implementations with a resulting lack of parallelism. The poor results are not due to a poor load balance but to a low degree of parallelism in the circuit, i.e. to the lack of workload to balance.

Even though the results of the proposed algorithm are better than the results of the other algorithms, a greater improvement was initially expected. In a deeper analysis, the commercial RTL designs used in the tests turned out to have a low degree of activity, with a significant number of cycles with insufficient parallelism to exploit the benefits of the CYCLIC algorithm.

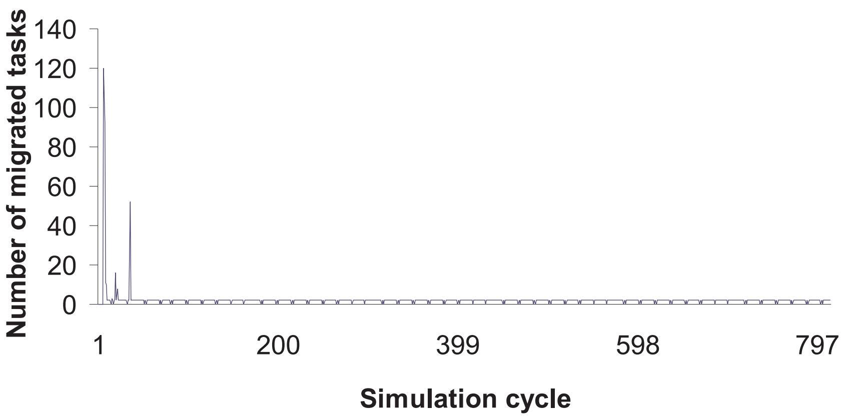

Fig. 4. Task migration without task stealing

5.1 Alternatives That Have Been Analyzed

Two modifications have been considered to improve the CYCLIC algorithm. The first one is to order the tasks by time to complete. The second one is to introduce task stealing to improve unexpected imbalance.

5.1.1 Ordered Queues

An interesting addition to the CYCLIC algorithm was to order the tasks by time to complete. This simplifies the search for candidate tasks for migration and allows easy implementation of the best fit algorithm, thus moving the minimum number of tasks to balance the load and improving the locality of references. Then, new traces were collected and analyzed using the CYCLIC algorithm with ordered queues.

Though a lower number of tasks are migrated and therefore the locality of references is improved, execution times increase because the overhead of inserting all the tasks in their respective queues in the appropriate order is not compensated. To explain this behavior Figure 4 shows the task migration without task stealing when simulating the Cpu_rtl design. This figure shows that the system quickly reaches a balanced status. From that moment on, only a very reduced number of tasks have to be migrated, making the benefit of ordering the queues so negligible that it cannot offset the overheads introduced by ordered insertion. Therefore the option of ordering queues has been discarded.

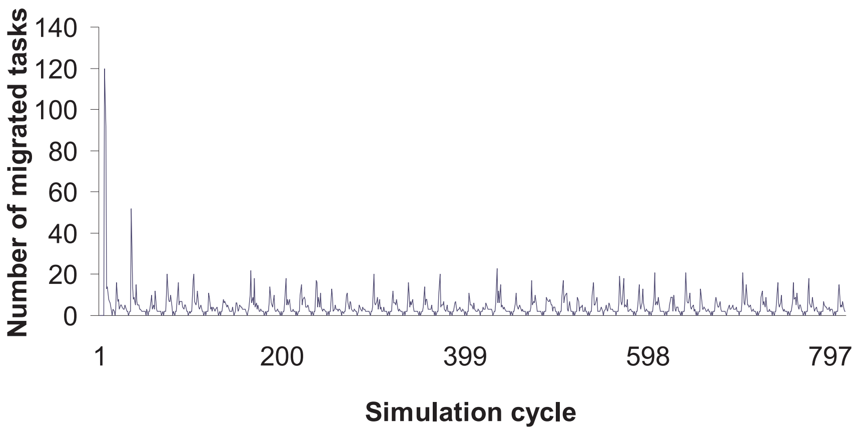

Fig. 5. Task migration with task stealing

5.1.2 Task Stealing

A second addition to the algorithm was explored because there were sporadic situations where a task could be preempted from the processor by the operating system to execute another activity with higher priority. This fact implies introducing unexpected imbalances in a previously balanced system. Moreover, as the balancing operations are only performed at the beginning of each simulation phase, this imbalance is not corrected.

To counter this, task stealing was introduced at the end of each phase. Thus, when a processor completes its assigned work it starts stealing tasks, one at a time, from processors with a higher pending workload.

New traces were then collected and analyzed but this time using the CYCLIC algorithm with task stealing. The results for simulating the Cpu_rtl design can be seen in Figure 5.

As can be seen in Figure 4, the system without task stealing quickly falls back into a balanced status, with very few tasks migrating from processor to processor. Task stealing, although improving some sporadic situations of imbalance, breaks up the distribution previously established, increasing the number of tasks that have to be migrated to recover a balanced status, as can be seen in Figure 5. In view of this poor performance, task stealing has also been discarded.

6 CONCLUSIONS

The proposed algorithm behaves better than the rest of the algorithms studied because it enforces the locality of references and provides good load balance. It guarantees good locality of references because usually each task is executed on the same processor where it was previously executed, reusing the data stored in the cache memory and migrating tasks only if there is load imbalance. It obtains very good workload distribution because the workload is balanced correcting possible imbalances while keeping the number of tasks migrated to a minimum in order to maintain a good locality of references. Moreover, as the time employed to execute each task is known, it provides a much more accurate load balance than if the number of tasks per processor were employed as the balance factor.

The algorithm introduces very low overheads. The measurements to estimate the workload produced by each task are performed few times and, as the simulation progresses, the workload distribution becomes more and more balanced and the number of tasks that need to be moved at the beginning of each phase is reduced to a minimum.

Considering the results obtained, it can be concluded that CYCLIC is an effective load-balancing algorithm, providing good workload distribution and good locality of references. This leads to an improvement of execution times with respect to conventional scheduling algorithms.

This algorithm can be used in other environments with similar performance characteristics to those of PDES, i.e., cyclic and predictable execution.

REFERENCES

[1] Peterson, G. D.—Willis, J. C.: A Taxonomy of Parallel VHDL Simulation Techniques. In VHDL International Users' Forum, Boston, October 1995, pp. 7.11–7.18.

[2] Krishnaswamy, V.—Hasteer, G.—Banerjee, P.: Automatic Parallelization of Compiled Event Driven VHDL Simulation. IEEE Transactions on Computers, Vol. 51, 2002, No. 4, pp. 380–394.

[3] Fujimoto, R.: Parallel and Distributed Simulation Systems. In Proceedings of the 21st Winter Simulation Conference, Arlington, December 2001, pp. 147–157.

[4] Liu, P.—Wu, J. J.—Yang, C.: Locality-Preserving Dynamic Load Balancing for Data-Parallel Applications on Distributed-Memory Multiprocessors. Journal of Information Science and Engineering, Vol. 18, 2002, No. 6, pp. 1037–1048.

[5] Hamidzadeh, B.—Kit, L. Y.—Lilja, D. J.: Dynamic Task Scheduling Using Online Optimization. IEEE Transactions on Parallel and Distributed Systems, Vol. 11, 2000, No. 11, pp. 1151–1163.

[6] Banicescu, I.—Cariño, R.—Pabico, J.—Balasubramaniam, M.: Design and Implementation of a Novel Dynamic Load Balancing Library for Cluster Computing. Parallel Computing, Vol. 31, 2005, No. 7, pp. 736–756.

[7] Banicescu, I.—Velusamy, V.: Load Balancing Highly Irregular Computations with the Adaptive Factoring. In Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Washington DC, April 2002, pp. 195.

[8] Banicescu, I.—Velusamy, V.: Performance of Scheduling Scientific Applications with Adaptive Weighted Factoring. In Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, April 2001, pp. 791–801.

[9] Milojičić, D. J—Douglis, F.—Paindaveine, Y.—Wheeler, R.—Zhou, S.: Process migration. ACM Computer Surveys, Vol. 32, 2000, No. 3, pp. 241–299.

[10] Frachtenberg, E.: Process Scheduling for the Parallel Desktop. In Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks ISPAN'05, Las Vegas, December 2005, pp. 132–139.

[11] Antonopoulos, C. D.—Nikolopoulos, D. S.—Papatheodorou, T. S.: Scheduling Algorithms With Bus Bandwidth Considerations for SMPs. In Proceedings of the 2003 International Conference on Parallel Processing (ICPP '03), Kaohsiung, October 2003, pp. 547–554.

[12] Xu, J.—Chen, M.—Zheng, G.—Cao, Z.—Lv, H.—Sun, N.: SimK: A Parallel Simulation Engine Towards Shared-Memory Multiprocessor. Technical report, ICT, 2008.

[13] Xu, J.—Chen, M.—Zheng, G.—Cao, Z.—Lv, H—Sun, N.: SimK: A Large-Scale Parallel Simulation Engine. Journal of Computer Science and Technology, Vol. 24, 2009, No. 6, pp. 1048–1060.

[14] Rudolph, L.—Slivkin-Allalouf, M.—Upfal, E.: A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91), South Carolina, July 1991, pp. 237–245.

[15] Torrellas, J.—Lam, M. S.—Hennessy, J. L.: False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Transactions on Computers, Vol. 43, 1994, No. 6, pp. 651–663.

[16] Squillante, E. D.—Lazowska, M. S.: Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, 1993, No. 2, pp. 131–143.

[17] Konas, P.—Yew, P.: Parallel Discrete Event Simulation on Shared-Memory Multiprocessor. In Proceedings of 24th Annual Simulation Symposium, New Orleans, April 1991, pp. 134–148.

[18] Bertogna, M.—Cirinei, M.—Lipari, G.: Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. IEEE Transactions on Parallel and Distributed Systems, Vol. 20, 2009, No. 4, pp. 553–566.

[19] Yan, Y.—Zhang, X.: Cacheminer: A Runtime Approach to Exploit Cache Locality on SMP. IEEE Transactions on Parallel and Distributed Systems, Vol. 11, 2000, No. 4, pp. 357–374.

[20] Suh, G. E.—Rudolph, L.—Devadas, S.: Effects of Memory Performance on Parallel Job Scheduling. In Proceedings of the 7th International Workshop on Job Scheduling Strategies for Parallel Processing (in LNCS 2221), London, July 2001, pp. 116–132.

[21] Suh, G. E.—Devadas, S.—Rudolph, L.: Analytical Cache Models with Applications to Cache Partitioning. In Proceedings of the 15th international conference on Supercomputing (ICS '01), Sorrento, June 2001, pp. 1–12.

[22] Chandra, A.—Shenoy, P.: Hierarchical Scheduling for Symmetric Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, Vol. 19, 2008, No. 3, pp. 418–431.

[23] Gan, B. P.—Low, Y. H.—Jain, S.—Turner, S. J.—Cai, W.—Hsu, W. J.—Huang, S. Y.: Load Balancing for Conservative Simulation on Shared Memory Multiprocessor Systems. In Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS '00), Bologne, May 2000, pp. 139–146.

[24] Vee, V. Y.—Hsu, W. J.: Cache-Aware Load-Balancing Mechanisms for Synchronous Computations on Shared-Memory Multiprocessors. In Proceedings of the TENCON 2000, Kuala Lumpur, September 2000, pp. 4–9, Vol. 2.

[25] Blumofe, R. D.—Joerg, C. F.—Kuszmaul, B. C.—Leiserson, C. E.—Randall, K. H.—Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, July 1995, pp. 207–216.

[26] Markatos, E. P.—LeBlanc, T. J.: Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, Vol. 5, 1994, No. 4, pp. 379–400.

[27] Feitelson, D. G.—Rudolph, L.: Parallel Job Scheduling: Issues and Approaches. In Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, Santa Barbara, April 1995, pp. 1–18.

[28] Markatos, E. P.: Scheduling for Locality in Shared-Memory Multiprocessors. Ph. D. Thesis, Department of Computer Science, University of Rochester, 1993.

Antonio García-Dopico received the M. Sc. degree in computer engineering and the Ph. D. degree in computer science from the Technical University of Madrid (UPM), Spain, in 1993 and 2001, respectively. He is currently an Associate Professor in the Department of Computer Systems Architecture and Technology at UPM. His research interests include computer architecture and parallel and distributed computer systems.

Antonio Pérez received the M. Sc. degree in telecommunication engineering and the Ph. D. in computer science from the Technical University of Madrid (UPM), Spain, in 1979 and 1982, respectively. He is currently a Full Professor in the Department of Computer Systems Architecture and Technology at UPM. His research interests include computer architecture, fault tolerant computers, and microprocessor systems design.

Santiago Rodríguez received the M. Sc. degree in computer engineering and the Ph. D. degree in computer science from the Technical University of Madrid (UPM), Spain, in 1990 and 1996, respectively. He is currently an Associate Professor in the Department of Computer Systems Architecture and Technology at UPM. His research interests include real time systems and fault tolerant computers.

María Isabel García received the M. Sc. degree in computer engineering and the Ph. D. degree in computer science from the Technical University of Madrid (UPM), Spain, in 1982 and 1985, respectively. She is currently an Associate Professor in the Department of Computer Systems Architecture and Technology at UPM. Her research interests include computer architecture and instruction level parallelism architectures.