Gang Scheduling with Memory Considerations*

Anat Batat Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

A major problem with time slicing on parallel machines is memory pressure, as the resulting paging activity damages the synchronism among a job's processes. An alternative is to impose admission controls, and only admit jobs that fit into the available memory. Despite suffering from delayed execution, this leads to better overall performance by preventing the harmful effects of paging and thrashing.

1. Introduction

A major issue for parallel systems is how to divide the system resources among a number of competing jobs, giving a reasonable quality of service to all the system's users. This allocation of resources is done by a central operating system which runs on a host that manages the whole system. The system scheduler determines when and on which nodes a job will be executed.

One way of scheduling parallel jobs is *gang scheduling* [10, 5]. The idea is to map the threads of a parallel job to distinct processors, and then schedule them to be executed simultaneously on their respective processors. Time slicing is used for interactive response times, and this is coordinated across the processors. Most studies find gang scheduling to be very efficient.

A major drawback of current implementations of gang scheduling is that they do not take memory requirements into account. Today, when there is a tendency to run a complete Unix operating system on every node of a parallel system, paging can be had for free. However, such paging is undesirable, because the paging mechanism is not synchronized by nature [3]. Using it may harm the synchronization of the parallel program's threads. In order to avoid paging all the program's used address space must be memory resident.

The simplest way to prevent memory pressure is to run only a subset of the jobs, and delay the rest until the needed memory becomes available. The main question is whether the queueing delay will be bigger or smaller than the delays caused by the paging inefficiency. Our research indicates that the answer is that delaying processes execution is more efficient then running them all together, using paging.

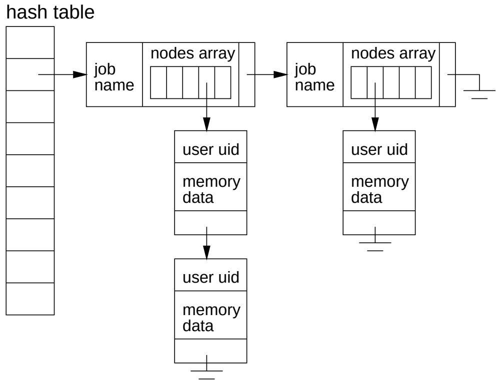

Figure 1. Data structure used to store and access the jobs' memory information.

2. Memory Usage Estimation

To use addmission controls, one has to know in advance how much memory a job will use. This can be done in two ways: estimation by using knowledge on previous executions of the job, or using information on the static memory which exists in the executable file.

2.1. Using previous information

2.1.1. Identifying the job

In order to use data on previous executions of a job, first we have to define how a job is identified; or in another words, we have to define when two jobs are identical enough in a way that there is a high probability they will consume the same amount of memory. We use the following three characteristics: file name (the executable path), number of nodes the job uses, and the name of the user [7]. If we don't have information on previous executions of a job according to this triple identification we can use information that exists for partial identification of the job: path + number of nodes, or even path only.

The data structure used to store job-related information is shown in Figure 1. It is based on hasing the path, an array of possible node values, and a record for each user. This data structure's total size (in the ParPar implementation) is 320K, where the maximum number of different job's paths is 2048, and the maximum number of different jobs (path + user) is 4096.

2.1.2. Data collection

The memory consumption of every process is measured on its node and then sent to the host. In principle, the memory usage of each process can be found from the `rusage` struct returned by `wait4` after it terminates. Unfortunately, these fields are not filled in all the versions of Unix that we checked. Therefore we resort to reading the kernel data structures periodically (once a second) as is done by the `top` utility. Additional measurements are done immediately after a process is created, so as not to miss short processes. As memory usage is non-decreasing, only the last value is stored. When the process ends its run, the stored value is sent to the host with other information about the process and its run.

In order not to clog the system with stale data, it is necessary to get rid of old information. To do so, each jobs' memory information can be divided into two: current month information and previous month information (any other time period can also be used). At the end of a month, the current month data becomes the previous month data and the collection of the new month's information is started. In case the job hasn't been run for two months, its entry is removed. This way, the information that is used to estimate a job's memory usage is not older than two months and at least the last month executions' information, if such exists, is available.

2.1.3. Estimation function

The value we produce using estimations is most likely to be inaccurate. Estimating a value too high or too low, compared to the correct one, has disadvantages. A too high value may cause a postponement of the execution of our job or later arriving jobs due to the misconception that the available memory space is insufficient. The system resources utilization will be damaged as a result of such an estimation. On the other hand, a too low value is likely to cause excessive paging and swapping, either because the job will consume more memory than the nodes can supply, or because new jobs will be scheduled according to an over evaluation of the free memory space, which (in both cases) will slow down the node and along with it, the whole system.

The estimation function we chose is the minimum between the maximum previous value and the average of previous values plus 3 standard deviations. To check the quality of the different estimation functions, we checked their performance using a log file of 9 months (January to September 1996, with approximately 50000 jobs) activity on the LANL CM-5. We predicted the memory size of every job according to previous similar executions and compared this with the actual usage.

The results are shown in Figure 2. We divided the jobs into buckets, according to the difference between the predicted value and the real value. The left graph is a histogram of the number of jobs in each bucket (the y axis) where the x axis shows the absolute difference between the predicted values and the real values; each bucket has a range of 1MB. In the second graph the x axis shows the differences between predicted and the real memory size as a percentage of the real size, and the range of each bucket is 50 percentage points. Observing the graphs, we can see that a high number of jobs concentrated near 0 indicating very good predictions. In about 87% of the jobs the absolute difference is smaller then 5MB, and in 70% of the jobs the absolute difference is smaller then 1MB. We can also see there are less jobs on the negative part of the graph, where the estimated value is lower than the real value (this happened to only 6% of the jobs).

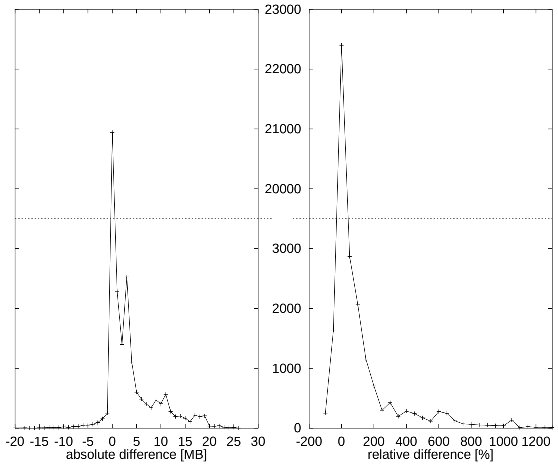

Figure 2. Quality of estimation function (note the split linear scale).

2.2. Static memory information

We don't always have information from previous runs. The alternative is to find static information from the executable. Every executable file in `a.out` [1] format contains in its beginning a struct with compiler-known information on the executable. This includes information about the sizes of the text segment, the data segment, and the bss.

To check the accuracy of `a.out` data as a predictor of the program memory usage, we measured the actual memory usage of processes and compared it with their `a.out` information. We measured three different classes of processes:

- *Students' programs* gave very poor results: the static memory size depended on the student's course rather then on the actual memory size. This is probably because most of the programs were written in C++, and therefore included lots of dynamic allocations.
- *System utilities* (applications like: netscape, csh, latex

etc.) gave a better correlation.

- *Scientific programs*, represented by the SPEC benchmark programs, showed a strong connection between the static memory and the actual memory. However, it should be remembered that the results from the previous section indicate that in real parallel systems the situation is not so rosy — if jobs always used the exact amount of memory indicated in their executable, the predictions would always be perfect.

3. Scheduler Implementation

The idea of queueing jobs that do not have enough memory available was implemented in the framework of the ParPar system [4]. This section first describes the system, and then the implementation of the memory considerations.

3.1. The ParPar system

The ParPar prototype cluster is built from 17 PCs: A host and 16 nodes. The PCs have an Intel Providence motherboard with a Pentium Pro 200 processor, 64 MB DRAM, and a 2.1 GB SCSI disk. The nodes are connected by two independent networks. One is a switched Ethernet which serves as a control network, using the conventional TCP/IP protocol and a reliable multicast protocol developed locally [8]. The other network is a 1.28 Gb/s Myrinet dedicated to the users' applications communication using MPI over FM [11].

The ParPar software is based on the Unix BSDI system, and runs at user level. It includes daemons which run on the host (masterd) and on the nodes (noded), and graphical user interfaces for each running job (job rep). In addition there are the processes (sprocs) which make up the parallel jobs (xprocs) which are executed by the system.

The job scheduling we are concerned with is done by the masterd.

3.2. Adding memory considerations to scheduling

The ParPar system uses gang scheduling, combining time slicing with space slicing. The main idea is that the processes of a parallel job are mapped to distinct processors, and are then scheduled to execute simultaneously on their respective processors. The mapping of processes to processors uses Ousterhout's matrix algorithm, where columns represent nodes and rows are scheduling slots [10]. Packing of jobs into slots is done using a buddy system, as in the Distributed Hierarchical Control scheme [5, 6]. This uses a tree of controllers that preside over nested groups of power-of-two processors.

After we estimated the memory size of a job, we need to add this information to our scheduling decisions. The original gang scheduling algorithm considers only the load factor — the number of running processes on a node. It selects the least loaded controller such that a time slot exists in which all the controller's nodes are free. To add memory load considerations, we also check that each of these nodes satisfies

$$proc\ mem \leq (node's\ phys\ mem) \times C - used\ mem \quad (1)$$

Where C is a constant bigger then 0. C limits the maximum used memory, and is expected to be near unity.

If there are not enough nodes with sufficient memory, the job will be pushed into a queue until the required memory will be available, or until the system total load will be zero (to make it possible to run processes which consume more then the initial total memory).

When a job finishes its execution the queue is scanned for the first job that can be executed, i.e. there are enough nodes with enough available memory (more or equal to its estimated memory usage) to execute all its processes. This procedure is repeated until there is not enough memory to run any additional jobs. To avoid starvation a job in the queue cannot be skipped more then 15 times. If a job was skipped 15 times no job will be executed until this job will.

The queue and its management already existed in the ParPar system. The memory considerations full implementation was added to the ParPar resource management module, in the function that chooses the controller and slot.

4. Experimental Results

In order to check the efficiency of these ideas, we compared the system performance with and without our improvements. We checked the effect of memory pressure on performance, comparing the influence of different degrees of memory pressure. In cases of lack of available memory space, a major factor that influences the system performance is the locality of the program's memory accesses. Therefore we ran programs with several locality levels.

4.1. Workload

Our workload was composed of synthetic applications that were executed on the ParPar system. This is not a simulation but a real execution of multiprocess jobs. Thus all the paging effects are real and measured directly, and we do not need to model them.

We chose to use synthetic programs instead of existing benchmarks for several reasons. The main reason is a desire to check very specific issues. The behavior of real programs is influenced by lots of factors. Using synthetic programs lets us restrict the influence of irrelevant factors and limit the checking to issues that interest us. Another reason not to use benchmarks is that there are no agreed benchmarks, especially for memory usage.

The programs we created use lots of memory, enough to cause memory pressure by executing only a small number

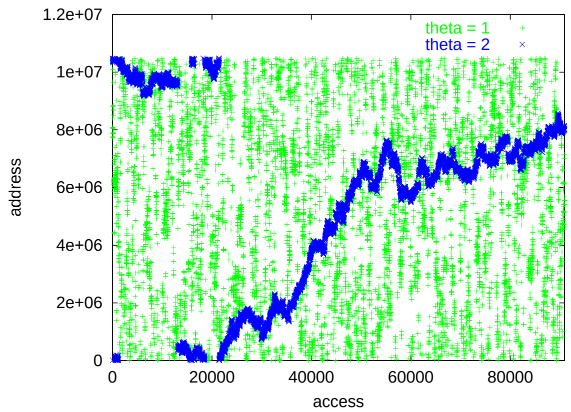

Figure 3. The spread of 90909 random accesses with low and high degrees of locality.

```
char mem[K*RANGE];
for (i=0 ; i<max ; i++) {
   randint = random();
   r = ((double)randint) / maxrand;
   jump = pow( r, theta );
   if (randint & 0x01)
      jump = -jump;
   addr += jump;
   intaddr = (int)addr;
   intaddr = intaddr % RANGE;
   if (intaddr < 0)
      intaddr += RANGE;
   intaddr *= K;
   mem[intaddr] = intaddr % 200;
}
```

Figure 4. The code of the program that was used in the experiments. `max` **and** `theta` **are arguments.**

of jobs. The programs also have a varying degree of locality based on the fractal model [13]. According to this model, the dynamic behavior of a program can be modeled as a one-dimensional fractal random walk. A program's memory accesses are simulated by generating random numbers $X(t)$, where $X(t)$ indicates the jump from the address accessed at time t to the address at time $t+1$. The distribution of $X(t)$ is given by:

$$Pr[X(t) > u] = Bu^{-\theta}$$

Where t is the virtual time marked by the ticks of the accesses to memory, B is a normalizing constant (we chose $B = 1$) and θ is the locality degree of the program's memory accesses and satisfies $1 \leq \theta \leq 2$. θ = 1 produces the lowest locality and θ = 2 produces the highest locality. Figure 3 shows the spread of 90,909 random accesses, when θ = 1 and when θ = 2.

Figure 4 shows the main part of the code of the program. The program uses a static array of 10MB. To ensure that this whole range is covered when $\theta = 2$, about 100000 accesses are needed. Therefore `max` is a multiple of this number. In each iteration, the program produces a random jump and accesses an array cell on the K multiple of the new address (previous address + jump). The size of the jump depends on θ. When θ = 2 (the highest locality) 94% of the jumps' sizes are less than 4. Two successive memory accesses with such a jump will be within a range of 4K, thus in a range of a page. When θ = 1 (the lowest locality), only 75% of the jumps' sizes are less than 4.

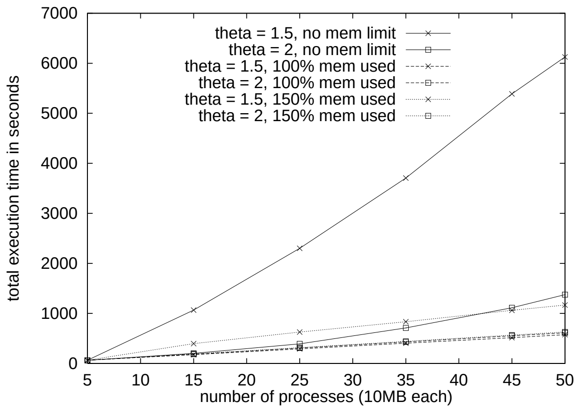

Figure 5. Execution time as a function of number of processes, for different combinations of memory pressure and locality level.

4.2. Memory load and locality

The first experiments used a static workload on a reduced version of the ParPar: masterd and two nodeds.

The first experiment was run once without limiting the memory pressure, and again when limitating the memory load to 100% and 150% (this limit is C from Equation 1). We considered all the non-kernel memory as our free memory. This size is exaggerated, because apart from the nodeds and sprocs some other system processes also run there. These processes' memory usage can reach up to 20% of the non-kernel memory. The size of the main memory of a node machine is 64MB. About 55MB of it is the non-kernel memory, and about 45MB of those 55MB are actually free.

We measured the execution time of different numbers of jobs from 5 jobs till 50. Every job runs on two nodes. Every instance of the job used 10MB of memory space on each node. Thus, the available memory on a node can contain 4 jobs at most. The experiment was repeated several times with different degrees of locality of memory accesses. Figure 5 shows 6 different graphs, one for each combination of memory pressure and locality level. The graphs show execution time as a function of number of jobs (which correlates with memory pressure).

From Figure 5 it can be seen that when the memory pressure is limited by our mechanism the graph is linear. When

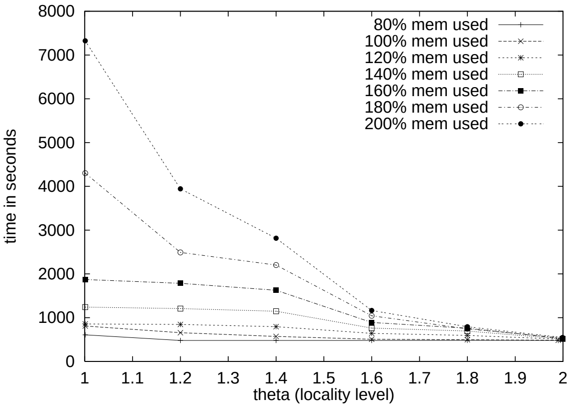

Figure 6. Execution time as a function of locality level, for different memory pressures.

there is no memory limit, the graph's gradient grows superlinearly with the number of jobs. It also can be seen that the locality degree has a big influence on the results when the system uses more memory than available. The most important result is that for a given θ the run time of the jobs is consistently smaller when the improved version of the system is used, that is, when the maximum used memory is limited and the jobs are put in a queue until enough memory becomes available (in contrast to the case when the used memory is not limited and the UNIX operating system has to deal by itself with the memory overload).

In another experiment we investigate the interaction of the locality level with different limitations on the maximum memory pressure, from 80% till 200% (as explained above, the size we refer to is the initial free memory, which might be 20% higher then the actual size, so 80% is roughly the actual free memory). We measured the execution time of 50 jobs at each locality level. Each job runs on two nodes, and every instance of the job uses 10MB of memory space. Figure 6 shows different graphs for the different memory pressures. Each graph shows execution time as a function of locality (theta).

From Figure 6 it can be seen that low locality coupled with a high degree of memory overallocation causes long execution times. On the other hand, if either locality is high, or memory is not overallocated, the jobs' runtime remains low.

4.3. Paging vs. queueing

In the experiments described above, all the jobs are run together and every one of them uses all the nodes in the system. Thus, the system is fully used all the time. Under these conditions, the influence of memory management on job scheduling is not checked, and a possible delay in the execution of jobs is not brought into consideration. Such a delay can damage the scheduling and decrease the system performance.

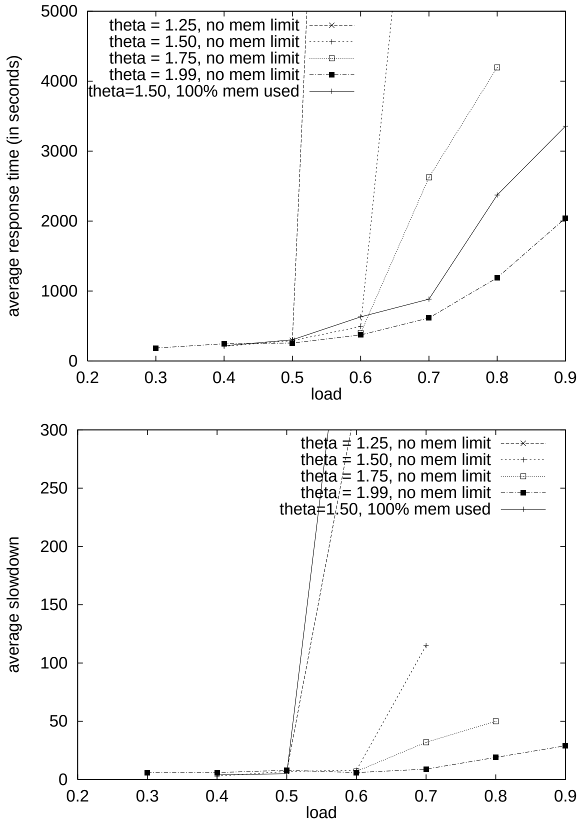

Figure 7. Average response time and average slowdown as a function of load, for different locality degrees.

We therefore also checked the efficiency of our ideas under more realistic conditions — on a system where the arrival time, the number of nodes, and the run time of the jobs change. We used the full size system: masterd + 16 nodeds. We ran 1000 jobs of 10MB each. The distributions of the run times, numbers of nodes, and arrival times were based on the workload model developed by Uri Lublin [9]. We checked different degrees of locality in memory accesses. The experiment was repeated for different loads, once when the memory load was not limited and once with limitation of memory load to 100% of the real memory. The system load is defined as:

$$load = \frac{avg\ run\ time \times avg\ node\ num}{avg\ interarrival\ time \times total\ nodes}$$

The results are presented in figure 7. The graphs present average response time and average slowdown as a function of system load. Response time is the total time the job spent in the system, from its arrival time till its execution is completed (including the time it spent in the queue). Slowdown is the ratio of actual run time (response time) to expected run time.

As can be expected, when the system is run without limiting memory load, there is a critical load at which it becomes saturated and cannot handle the job stream any more. If programs have a high degree of locality, this happens at a higher load.

Limiting the memory load consistently decreases the response time. The slowdown on the other hand, is highest when the maximum memory load is limited. The reason for these high values is a high slowdown of short jobs which are forced to wait in the queue. The percentage of queued jobs starts to grow when the load becomes more than 0.5, and reaches 80% when the load is 0.8.

5. Conclusions and Future Work

Thrashing by nature damages the performance of a system, but for a system that runs parallel jobs the damage is much worse. The main question we faced was how to add memory considerations to the system's scheduler, which would prevent thrashing and also would not harm the system's utilization.

Our solution was to estimate the memory usage of newly arrived jobs, using information on their previous runs or on their static memory consumption. A job is executed only on a node that has enough available memory for its estimated memory usage. If there are not enough nodes that fulfill this demand for all the job's processes, the job is inserted into a queue until the needed memory will be vacated.

This idea was fully implemented on the ParPar testbed. Experimental results showed that the system performance improved significantly. When the maximum memory load of the system was not limited the system couldn't stand high loads and thrashed. Limiting the maximum memory load prevented the thrashing and the system managed to handle higher loads. The best performance was obtained when the maximum used memory was exactly the available physical memory. These results show that our technique of adding memory considerations is efficient and can significantly improve the system performance.

Still, there is further work to be done. The biggest disadvantage of our proposed method is not specifically handling limitation of the time short jobs wait in the queue. The slowdown of a job depends on the relation between its pure run time and the whole time it spends in the system, including the time it spends in the queue. There are two possible solutions to limit the queue wait time:

1. To use swapping. If a job runs for too long time we can swap it out and load other job, which might be shorter [2].
2. To add run time considerations to the scheduler similarly to the memory considerations [7], and give priority to short jobs.

Another improvement to our work can be handling very big jobs whose memory usage was underestimated. Such processes can thrash the system. These jobs should be detected, swapped out of memory and their memory usage should be re-estimated.

Acknowledgements

Many thanks to the System Group, and especially Tomer Klainer, for their technical help in carrying out this research. Thanks also to Curt Canada of LANL for the CM-5 log, which is now available on-line [12]. This work was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences & Humanities, and by the Ministry of Science Basic Infrastructure Fund, Project 9762.

References

[1] a.out man page.

[2] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, LNCS 949 pp. 19–44, Springer-Verlag, 1995.

[3] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood, "*Paging tradeoffs in distributed-shared-memory multiprocessors*". *J. Supercomput.* **10(1)**, pp. 87–104, 1996.

[4] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. A. Volovic, "*The ParPar system: a software MPP*". In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, R. Buyya (ed.), pp. 754–770, Prentice-Hall, 1999.

[5] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[6] D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.

[7] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, LNCS 1291 pp. 58–77, Springer Verlag, 1997.

[8] A. Kavas, D. Er-El, and D. G. Feitelson, "*Using multicast to preload jobs on the ParPar cluster*". *Parallel Comput.* (to appear).

[9] U. Lublin, *A Workload Model for Parallel Computer Systems*. Master's thesis, Hebrew University, 1999. (In Hebrew).

[10] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[11] S. Pakin, V. Karamcheti, and A. A. Chien, "*Fast messages: efficient, portable communication for workstation clusters and MPPs*". *IEEE Concurrency* **5(2)**, pp. 60–73, Apr-Jun 1997.

[12] *Parallel workloads archive*. URL http://www.cs.huji.ac.il/labs/parallel/workload/.

[13] D. Thiébaut, "*On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio*". *IEEE Trans. Comput.* **38(7)**, pp. 1012–1026, Jul 1989.