# Evaluating the Local Ratio Algorithm for Dynamic Storage Allocation *

Kirk Pruhs[1] and Eric Wiewiora[2]

[1] Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA. 15260 USA
kirk@cs.pitt.edu
http://www.cs.pitt.edu/~kirk

[2] Computer Science and Engineering Department
University of California – San Diego
cwiewior@cs.udsd.edu

**Abstract.** We empirically compare the local ratio algorithm for the profit maximization version of the dynamic storage allocation problem against various greedy algorithms. Our main conclusion is that, at least on our input distributions, the local ratio algorithms performed worse on average than the more naive greedy algorithms.

## 1 Introduction

The paper [1] presents a general framework, the local ratio technique, for solving many scheduling and resource allocation problems. The input for these problems consists of tasks/tasks that must be scheduled. These tasks have some resource requirement, and some profit that is obtained if they are scheduled. The most obvious naive algorithms only consider one parameter of the tasks, and thus these naive greedy algorithms can then be arbitrarily bad in the worst case. For example, one possible greedy algorithm would be to schedule the most profitable tasks first, which would be bad if the slightly more profitable tasks have much higher resource requirements. For many of these problems it was not at all clear how to construct simple reasonable algorithms. The local ratio technique can be viewed as "sophisticated" greedy algorithm that balances these competing demands. For some of these problems, the local ratio algorithm is the first known constant approximation algorithm. So in summary, for these problems the local ratio algorithm is generally vastly superior, in the worst case, to the more naive greedy algorithms.

Our goal in this paper is to empirically test whether the local ratio algorithms are also superior on average to the more naive greedy algorithms. The application from [1] that we have selected for our experiments is the dynamic storage allocation problem with the objective function of maximizing the profit of the

* Supported in part by NSF grants CCR-9734927, CCR-0098752, ANIR-0123705, and by a grant from the US Air Force.

scheduled tasks. This problem seemed to us to be one of the more applicable and harder problems in [1].

We implemented several different variations of the local ratio algorithm, and many different greedy algorithms. We implemented a random instance generator, and a graphical user interface to view the outputs produced by the algorithms. All of these programs can be found at:

`http://www.cs.pitt.edu/~kirk/dynstore`.

An example of the graphical user interface showing a particular schedule is shown in figure 1. The graphical user interface proved to be quite a useful tool. The performance of the various algorithms is quite obvious from a visual inspection of their schedules for a few inputs.

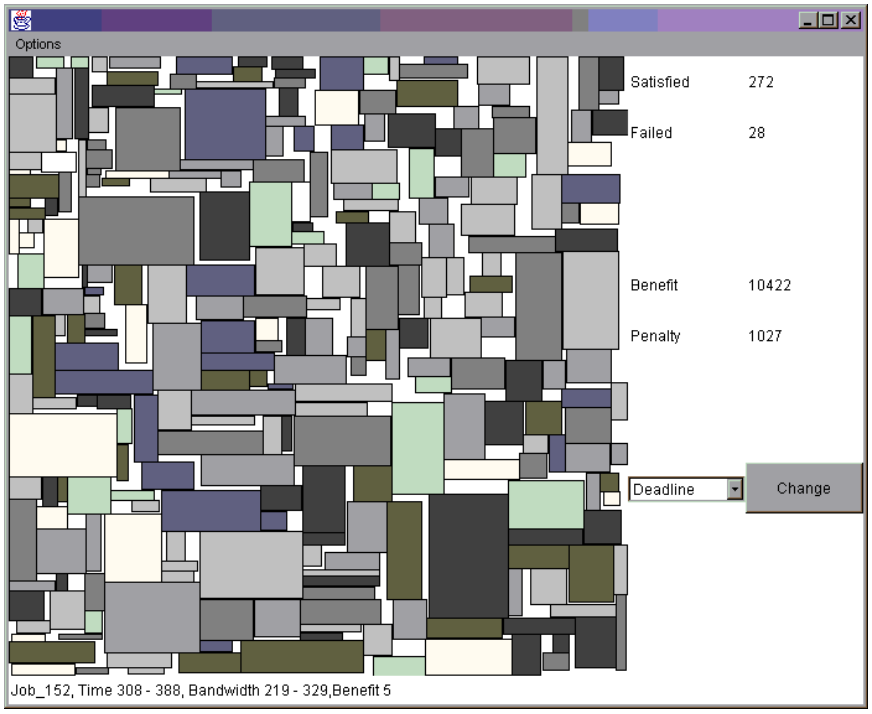


**Fig. 1.** An example of a flight schedule using a greedy placement

Our main conclusion is that, at least on our input distributions, the local ratio algorithms performed worse on average than the more naive greedy algorithms.

In section 2 we survey the related background, including a formal definition of the dynamic storage allocation problem, and descriptions of the algorithms that we implemented. In section 3 we describe the input distributions that we used. In section 4 we give our experimental results.

## 2 Background

### 2.1 Profit Maximization Dynamic Storage Allocation

An instance consists of $n$ tasks, where the $i$th task $R_i$ has a bandwidth requirement $b_i$, and a length/duration $p_i$. Thus each task can be thought of as a rectangle, with vertical height equal to the bandwidth, and horizontal height equal to the length. The goal is to assign the tasks to a scheduling space, which is a larger rectangular space with height normalized to 1, and length equal to some value $T$. No two tasks can be placed in such a way that they overlap in this scheduling space. Further each task $R_i$ has a release time $r_i$, and a deadline $d_i$. Each task $R_i$ must be started after $r_i$, and completed before time $d_i$. It is also convenient to define the laxity of a task $R_i$ to be $\ell_i = d_i - r_i - p_i$. Finally, each task $R_i$ has an associated positive integer benefit $w_i$ that is obtained by the scheduler if $R_i$ is scheduled. In the *unit case setting*, each benefit is 1. The objective function is to maximize the aggregate benefit of the scheduled tasks. We assume that all numbers are integers.

Dynamic storage allocation is classic combinatorial optimization problem with a wide variety of applications. For surveys see [2, 6]. Essentially all interesting versions of the the dynamic storage allocation problem are NP-hard [2]. The most commonly studied objective function is the size of the spectrum used to schedule all requests. That assumes that the scheduling space that does not have an a priori bound on the size of the spectrum. The best competitive ratio known for a polynomial time offline algorithm is 3 [3].

In [1] the local ratio algorithm is shown to have a constant worst-case approximation ratio for the profit maximization version of dynamic storage allocation. In [5] a LP-based polynomial time algorithm with a constant competitive ratio is given. The authors of [5] encountered this problem in a EU research project Euromednet within the context of scheduling requests for remote medical consulting on a shared satellite channel.

There have been some experimental studies of some heuristics such as First Fit for the objective function of minimizing the bandwidth required to schedule all tasks (see [6] for a survey), but to the best of our knowledge there has been no experimental studies of algorithms for the profit maximization version of dynamic storage allocation.

### 2.2 The Greedy Algorithms

All of our greedy algorithm consider the tasks in some order. All of the greedy algorithm use First Fit to place the task under consideration, that is, the greedy algorithms differ only in the order that they attempt to place the tasks. First Fit tries to place the task as early in time as possible, and then as low in the bandwidth dimension as possible. We considered about 12 different ordering methods. We only report on the best 2 here.

**Deadline** : Considers the tasks in nondecreasing order of $r_i + \ell_i$, the latest time that the task can be run. (Note that this is a nonstandard use of the term deadline.)

**Profit Area** : Considers the tasks in nonincreasing order of $\frac{w_i}{p_i \cdot b_i}$, the benefit density of the request.

### 2.3 The Local Ratio Algorithms

We now explain how the local ratio algorithm works on an arbitrary instance. First $\ell_i + 1$ copies of each task $R_i$ are created in the instance, one for each possible starting time of $R_i$. The release time of the $j$th copy, $0 \leq j \leq \ell_i$, is $r_i + j$, and the deadline of the $j$th copy is $r_i + j + p_i$. So each copy has zero laxity.

The local ratio algorithm consists of two stages. The first stage is recursive and determines the placement $S$ of the tasks to be scheduled in time. The first stage has a parameter $\alpha$, which in some sense captures the value of tasks that can be finished quickly versus the value of other characteristics of a task. If the linear resource was fungible (in this case that means that the bandwidth used by each task could be reassigned at each time instant) then $S$ would be feasible. However, in our dynamic storage allocation problem, the resource is not fungible. So a second stage is required to assign particular bandwidths to the tasks.

We now summarize the first stage. First all tasks with non-positive benefits are removed. Note that while all tasks initially have positive benefits, the benefits may become zero/negative in the course of the algorithm. If no tasks remain, then the empty schedule is returned. Among those tasks that can be added to the current schedule, select a task $R_i$ that can be finished as early as possible. That is, if one can add $R_i$ to the current schedule so that it is finished at time $t$, then the ending time of every other task that one might add to the schedule would be no earlier than time $t$. In the key step, the benefits of each task $R_j$ are modified in the following manner:

- If $R_j$ is a copy of $R_i$, then $w_j$ is set to $w_j - w_i$. While initially $w_i = w_j$ if these are two copies of the same task, the values may diverge in the course of the algorithm.
- If $R_j$ overlaps $R_i$ and $R_i$ and $R_j$ are not copies of the same task, then $w_j$ is set to $w_j - \alpha \cdot b_j \cdot w_i$. So the more valuable the selected job $R_i$ is, the less valuable the overlapping job $R_j$ is. The higher the bandwidth of $R_j$ the more its weight is lowered. And $\alpha$ balances these two factors versus the factor that $R_i$ may be completed before $R_j$.
- Otherwise, $w_j$ is not modified.

The problem is solved recursively on the modified instance. The task $R_i$ is added to the returned schedule if there is sufficient bandwidth to add $R_i$, and no other copy of $R_i$ has already been added to the schedule. Otherwise, the returned schedule is unmodified.

In [1], it is proposed that Kierstead's algorithm [4], for the standard dynamic storage allocation problem, be used for the second stage. Kierstead's algorithm schedules all the tasks in $S$ using bandwidth at most 6. The final schedule is then the horizontal slice, of height 1, that maximizes the weight of the tasks fully contained within the slice. We call the resulting algorithm *Unified+K*, as the local ratio algorithm is called the unified algorithm in [1].

We also tried various other possibilities for the second stage. Kierstead's algorithm involves rounding the bandwidths up to the next highest power of two. After these tasks are assigned bandwidth in the second phase, this may leave obvious gaps. In *Unified+KC* these gaps are removed, by letting the tasks drop in the bandwidth space as if by gravity, before the best horizontal slice is selected.

The algorithm *Unified+FF* considers the tasks from the first stage in random order, and then assigns the tasks to bandwidth using First Fit. Again the final schedule is the horizontal slice of height 1 that maximizes the weight of the tasks fully contained within the slice.

*Unified+FF2* is the same as Unified+FF, except that bandwidth requirements are rounded to the next higher power of 2. After the assignments are performed, the requests are returned to their original bandwidth. The resulting extra space is removed before the best bandwidth range is found.

It will also be instructive to plot the value of the tasks selected by the local ratio algorithms in the first stage. We will call this schedule simply *Unified.* Note that Unified is not a feasible schedule.

## 3 The Input Distribution

In this section we describe how our inputs were created. Recall that $T$ is the time period to be scheduled. Our input generalization is parameterized by 4 variables:

- $n$: The number of tasks
- $C$: The minimum bandwidth required to schedule all of the tasks
- $L$: The maximum ratio of a task's laxity to its length.
- $W$: The range of benefits of the tasks

We began with one task that has length $T$ and bandwidth $C$. We then iteratively cut this large task into smaller tasks until the desired number $n$ of inputs was achieved. We will think of the $T$ by $C$ task as being in the bottom of the scheduling space, and this placement will be an optimal solution if the total bandwidth measure $C \leq 1$. Generating instances with known optimum solutions is a standard method that allows one to compare algorithms against the optimal solution, as well as against each other.

On each iteration, all of the tasks from the previous round are split into two tasks. On the first round, and subsequent odd rounds, the tasks are cut horizontally. That is, the aggregate bandwidth of the two resulting tasks are equal to the bandwidth of the original task. On even rounds, the cut is vertical, reducing the duration of the tasks. At the end of every round, the task order is randomized. This process will continue until $n$ tasks are created. The point to cut each task is selected uniformly at random from the middle half of the task to be divided. Thus each partition has at least $1/4$ of the original task. Also, if the input is too small to be divided, that is the task is passed up and the next task is divided.

This sets the length $p_i$ and the bandwidth $b_i$ of each task. If $C \leq 1$, then also sets the placement of the tasks in the optimal schedule. We now explain how to set the release time and deadline for each task. The laxity $\ell_i$ of each task $R_i$ is selected uniformly at random from the range $[0, L \cdot p_i]$. We thus assume that, given a fixed $L$, the expected laxity of a task is proportional to its length. The release time $r_i$ of each task is then the start time in the optimal schedule minus a number uniformly at random from the range $[0, \ell_i]$, or zero, whichever is larger. Setting the release date and the laxity fixes the deadline. If the deadline is greater than $T$, then it is rounded down to $T$, and the release time is moved back in time so that the laxity remains fixed.

We should note that we also tried input distributions where release times, length and laxity were generated uniformly at random over fixed ranges. This input distributions produced essentially the same results as for the input distribution that we report on here.

## 4 Experimental Results

We only report on the quality of the solutions produced by the algorithms because all algorithms can be implemented efficiently. The bottleneck in the local ratio algorithms is the first stage. And despite some reasonable effort to implement the first stage efficiently (taking some suggestions from [1]), the local ratio algorithms were clearly slower than the greedy algorithms. We report on a representative sample of the many experiments that we ran.

### 4.1 Selecting $\alpha$

Our first goal was to determine how to select the parameter $\alpha$ in the local ratio algorithms. We ran the following tests in order to gauge different $\alpha$ values.

- $n = 300, 400, 500$. These input sizes were sufficiently large for the asymptotic behavior of the algorithms to manifest itself.
- $C = 0.8, .9, 1.0, 1.1, 1.2$. Note that loads near 1 are the most interesting case. If the load is much less than 1, then the problem is easy to solve optimally. In our experiments, inputs with loads below .8 could be solved optimally by at least some of the greedy algorithms. If the load is much more than 1, then, regardless of the scheduling algorithm, the system will reject many tasks, which will probably be unacceptable.
- $L = 0.0, 0.5, 1.0, 1.5, 2.0$
- $W = 1, 40, 80, 120, 160$
- $\alpha = 0.25, 0.50, 0.75, 1.00, 1.25, 1.50, 1.75, 2.00$
  Recall that $\alpha$ is a parameter of the local ratio algorithms.

We generated 10 instances at each point in the experimental space.

The horizontal axis in figure 2 represents the various choices of $\alpha$. The vertical axis in all of the figures is the average percent of total benefit obtained by the

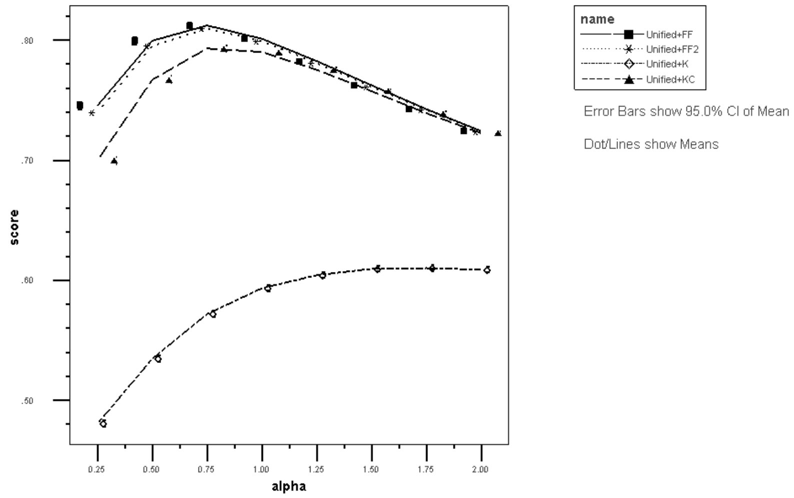


**Fig. 2.** $\alpha$ versus score

plotted algorithm (not the actual benefit obtained by the algorithm), where the average is taken over all inputs in our experimental space. The vertical bars in all plots are .95 confidence intervals.

From figure 2 one can see that the best choice for $\alpha$ is about .75 for our sampling space. Thus for the rest of the paper, we report on the local ratio algorithms with $\alpha$ fixed to be .75. We also concluded that Unified+K proposed in [1] is not competitive with the other three local ratio variations, which are all quite similar (at essentially all points in the experimental space). Thus from here on we will only plot Unified+FF.

After having set the best overall value for $\alpha$, we conducted another round of tests looking into other factors that affect performance in more depth. The rest of the findings of the paper are based on these tests

- $n = 300$
- $C = 0.8, .9, 1.0, 1.1, 1.2$
- $L = 0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0$
- $W = 1, 20, 80$
- $\alpha = 0.75$

We generated 100 instances at each point in the experimental space.

### 4.2 The Effect of Weights

The horizontal axis in figure 3 represents values of the weight parameter $W$, and the vertical axis is the average percent benefit.

Figure 3 illustrates that for all of the algorithms that we tested that the asymptotic effect of the weights kick in for relatively small values of $W$. So for ease of presentation, from here on, we only present results for the unit weight case $W = 1$ and for $W = 80$, which represents the asymptotic case.

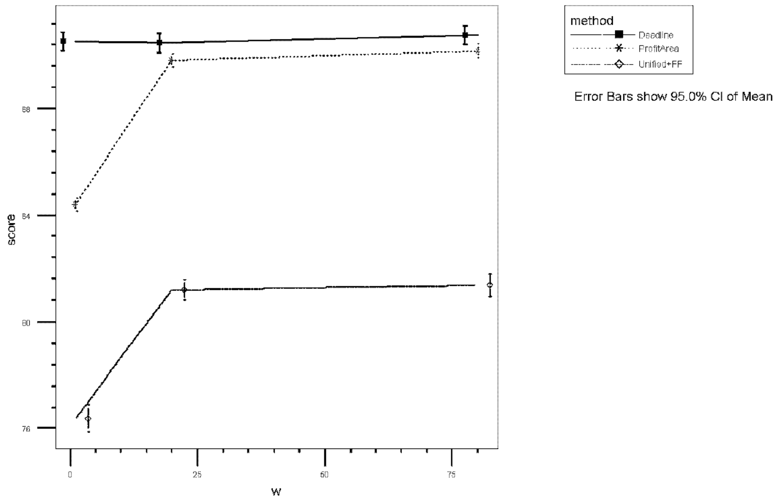


**Fig. 3.** The Effect of Weight

Since most of the experimental space contains relatively large values for the $W$ parameter, one might think that our choice of $\alpha = .75$ might not be optimal for small $W$, but in fact this is not the case; $\alpha = .75$ also nearly optimizes the performance of the local ratio algorithms when $W$ is fixed to 1. Fore-shadowing slightly, one can also observe in figure 3 that the local ratio algorithm do not perform as well as the best greedy algorithms.

### 4.3 The Effect of Laxity

The horizontal axis in figure 4 represents values of the laxity parameter $L$, and the vertical axis is the average percent benefit. Figure 4 shows illustrates that for all of the algorithms that we tested, laxity didn't have an appreciable effect on the benefit produced. This is somewhat surprising as increasing $L$ increases the flexibility of the scheduler.

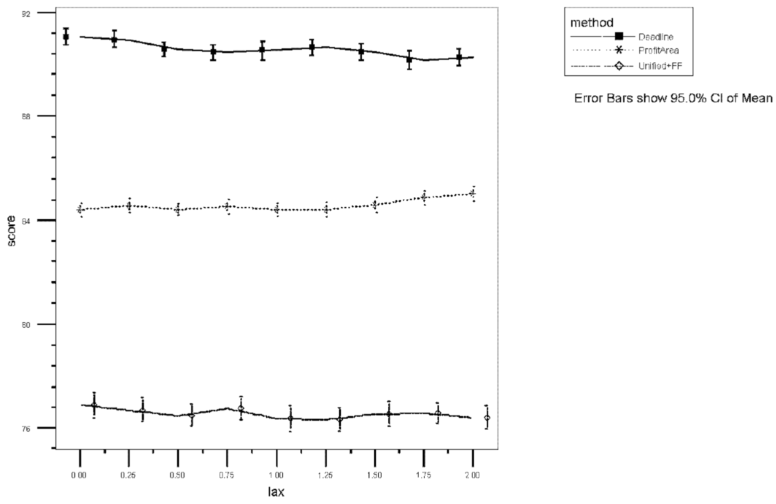


**Fig. 4.** Laxity $L$ versus Score for $C = 1$ and $W = 1$.

To simplify exposition, we will fix the laxity parameter $L$ to be 1 for the rest of the paper. Note that one can also again see that the greedy algorithms are outperforming the local ratio algorithms.

### 4.4 Performance Evaluations

We now plot the performance of the two best greedy algorithms, the best unified algorithm, and the value of tasks selected in the first stage of the local ratio algorithm for the best choice of $\alpha = .75$, for a representative choice of $L = 1$, and for $W = 1, 80$.

One can see from figure 5 the best algorithm for $W = 1$ is the Deadline greedy algorithm. In fact, for all but very high loads, the Deadline algorithm accrues even more benefit than the local ratio algorithm accrues in the first stage. Further the greedy algorithms do better than all unified algorithms. Note that for input with local parameter $C < .8$, the Deadline algorithm is quite near optimal.

One can see from figure 6 that, in the case of weights, the Deadline algorithm is the best choice for an underloaded system ( $C \leq 1$), and that ProfitArea

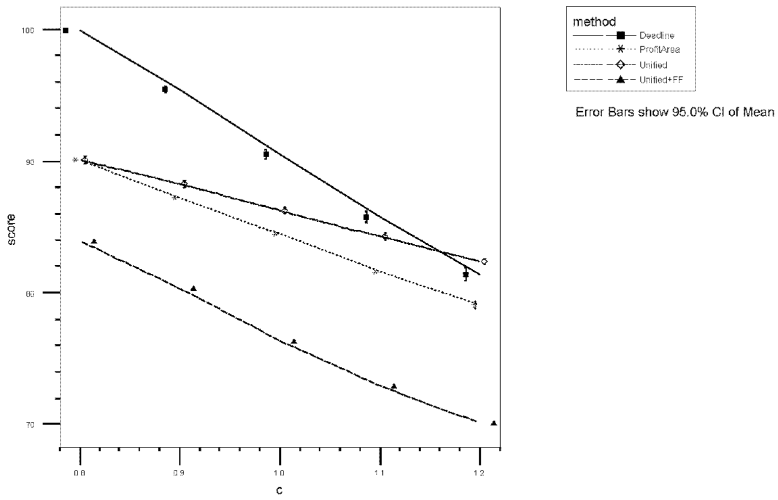


**Fig. 5.** Load $C$ versus Score for $L = 1$ and $W = 1$

algorithm is the best choice for an overloaded system ($C > 1$). Although the first stage of the local ratio algorithm beats the greedy algorithms when the load is high (say $C > 1.05$). Recall though that the first stage of the local ratio algorithm does not produce a feasible schedule.

## 5 Conclusions

Our goal was to compare the more sophisticated local ratio algorithm to more naive greedy algorithms for the profit maximization version of the dynamic spectrum allocation problem. While the local ratio algorithms are better in the worse case, our results suggest that they are inferior on the average case. Further, the local ratio algorithm is harder to program, and and is not as efficient as the greedy algorithms. Our results would suggest against using the local ratio algorithm in practice. It seems that the main reason that the local ratio algorithms does not do as well as the greedy algorithms is that the local ratio algorithms occasionally pick two copies of the same interval in the first stage. While the local ratio algorithms may perform better on highly overloaded systems, but it is not clear how useful that is in practice.

**Acknowledgments:** This research was conducted while the second author was a student at the University of Pittsburgh.

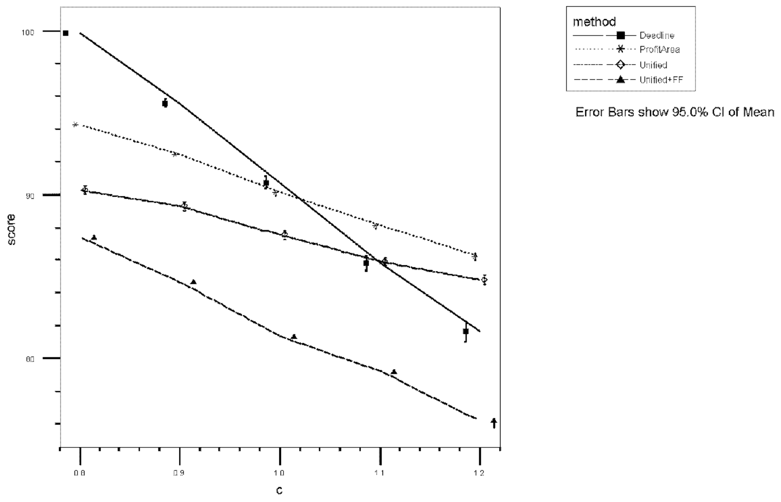


**Fig. 6.** Load $C$ versus Score for $L = 1$ and $W = 80$

## References


1. A. Barnoy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, "A unified approach to approximating resource allocation and scheduling", ACM Symposium on Theory of Computing, 2000. To appear in *JACM.*
2. E. Coffman, "An introduction to combinatorial models of dynamic storage allocation", *SIAM Review*, **25**, 311 – 325, 1999.
3. J. Gergov, "Algorithms for compile-time memory allocation", ACM/SIAM Symposium on Discrete Algorithms, S907-S908, 1999.
4. H. Kierstead, "A polynomial time approximation algorithm for dynamic storage allocation", *Discrete Mathematics*, **88**, 231 – 237, 1991.
5. S. Leonardi, A. Marchetti-Spaccamela and A. Vitaletti, "Approximation algorithms for bandwidth and storage allocation problems under real time constraints", Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2000.
6. P. Wilson, M. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: a survey and critical review", International Workshop on Memory Management, *Lecture Notes in Computer Science*, **986**, 1 – 116, 1995.