CS270 Course Project

Resource Constrained Cache-Affinity Scheduling

Feng Zhou(zf@cs), Li Zhuang(zl@cs)

Abstract

The Resource Constrained Cache-Affinity Scheduling problem is defined. It is practically useful in computer operating systems, where the problem models the scheduling of segments(tasks) from different threads(jobs) to minimize cache-filling cost. The problem is shown to be NP-Complete. A dynamic programming algorithm is presented that solves the optimization problem in $O(n^2r^n)$ time, where n is number of jobs and r number of tasks per job. This is a significant improvement over the naive $O(\frac{(rn)!}{(r!)^n})$ algorithm. Three greedy heuristic algorithms are presented and evaluated via simulation. In the experiments, the heuristic algorithms perform significant better than traditional non-cache-affinity scheduling methods.

1 Background

Suppose we are able to produce a graph of the structure of each thread in a program. The graph can hopefully be used to help the process-local thread scheduler to do better scheduling. The environment we are targeting at is highly concurrent CPU and/or I/O intensive applications which process a lot of repetitive tasks. Typical examples include Web/application/email servers. In this graph, edges represent non-preemptive code segments, while vertices represent blocking points due to I/O calls or voluntary yields. Each task starts at s and ends at t. Assuming uni-processing, at any moment there is at most one thread running on one of the edges. When it reaches the next vertex, it blocks for a certain period of time and the scheduler gets a chance to reschedule a new thread to run. At that moment, there can be thousands of runnable and un-runnable threads waiting at the vertices. The scheduler has to choose one runnable thread now waiting at one vertex to run. We call this graph a *blocking graph* and a scheduler exploiting this graph a *graph-based thread scheduler*.

Figure 1 is a simple blocking graph, where edges are annotated with branching probability. Figure 2 is a graph we obtained from Apache 2 web server, using runtime profiling. It represents the main part of processing a HTTP GET request.

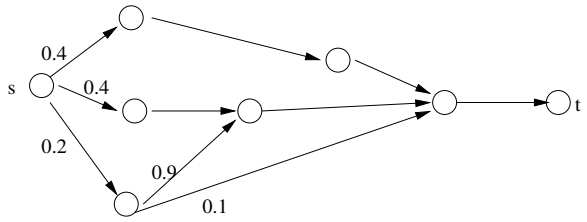

Figure 1: A blocking graph

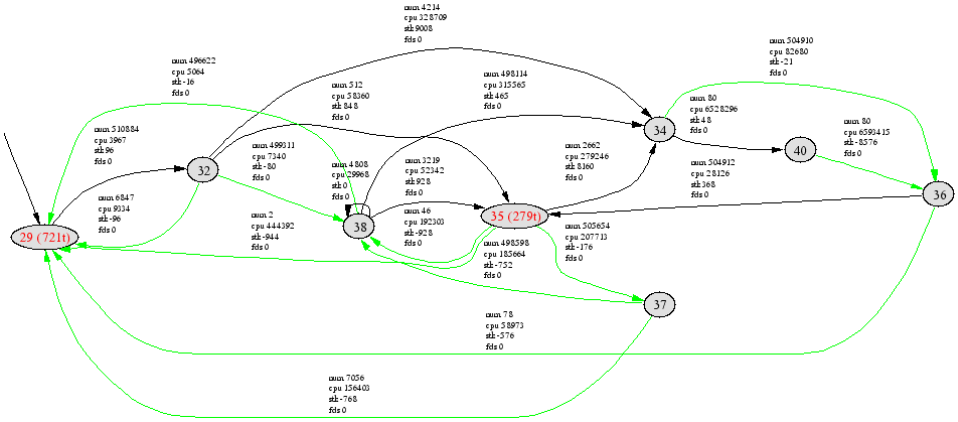

Figure 2: Blocking graph of threaded Apached 2 web server

Given the graph G and various knowledge about the system, our goal is to find a **scheduling policy** that **maximize throughput** and **maintain acceptable response time**. We can possibly obtain the following knowledge about the graph to aid the scheduler, either through static analysis or runtime profiling. The behavior of the system is random. So these values are averages.

- $p(e)$: The probability of the thread taking each branch (edge) at each vertex.
- $t(v)$: Average blocking time of threads at a certain vertex.

- $M(e)$, $m(e)$: Memory usage of an edge, with $M(e)$ being the maximum resource used during the edge, and $m(e)$ being the relative resource usage change after the edge.
- $S(e_1, e_2)$: Data cache / instruction cache affinity, represented by the cost to fill the cache when switching from one edge to another edge to run.
- M_0: Total available memory.

We consider only the **single processor** version of the problem. In this case, the edges from different threads are executed one after each other on the processor. If we take each edge as a separate task, the constraints that limit our selection of schedule are the precedence ordering of these tasks and the memory consumption limit. Our goal in the problem would be to minimize the total run time of a fixed amount of jobs. That goal directly translates to minimizing the cost of cache misses, because that is the only point where the ordering matters. The problem is formulated mathematically in the next section.

In the operating systems context, recent work on batch scheduling in staged event-driven architecture[6], *cohort scheduling* [4] and *affinity scheduling*[2] approaches the same problem in a coarser-grained way. Only the different costs between scheduling the same edge and a different edge are considered. So the scheduler schedule as many threads as possible that are running the same piece of code one after each other. In [5], Philbin et al. proposed a practical scheduling algorithm to improve cache locality of sequential programs by scheduling fine-grained threads. However, their targeted area was mostly cpu-intensive scientific applications.

2 Problem Formulation

Here we formulate the problem mathematically. We discuss assumptions and their implications in the next sub-section.

Definition 1 Resource Constrained Cache-Affinity Scheduling *(RCCAS) There are n independent jobs $R_1, ..., R_n$ to be processed on a single machine. Each job R_i consists of $|R_i|$ tasks $r^i_1, r^i_2, ..., r^i_{|R_i|}$, which have to be completed sequentially. A* schedule $\pi = \pi(1), ..., \pi(\sum_{i=1}^{n} |R_i|)$ *is a permutation of all r^i_j that satisfies,*

(a) Resource constraint, $\forall k_0, \sum_{k=1}^{k_0} m(\pi(k)) \leq M$. *$m(r_j^i)$ is the amount of system resource allocated by that task, negative if the task frees resource. Naturally, a job frees all resource when it finishes, i.e.* $\forall i, \sum_{j=1}^{|R_i|} m(r_j^i) = 0$.

(b) Ordering constraint, $\forall i, j_1 < j_2$, *if* $\pi(k_1) = r_{j_1}^i$ *and* $\pi(k_2) = r_{j_2}^i$, *then* $k_1 < k_2$.

The cost of the schedule is defined by,

$$cost(\pi) = \sum_{k=2}^{|\pi|} C(\pi(k-1), \pi(k))$$

where C is a $|\pi| \times |\pi|$ matrix.

Question*: does there exist a schedule that has a cost at most c?*

Basically this is a static and limited version of the problem stated in section 1. Each task corresponds to an edge in the blocking graph and a job is a path from s to t in the blocking graph. Instead of representing the composition of different threads using branching probabilities in the graph, we statically specifies the tasks in each job(thread). Therefore we do not have the blocking graph in this problem formulation. Notice two tasks from certain two jobs can possibly be the same task, i.e. $r_j^i = r_{j'}^{i'}, i \neq i'$. This corresponds to the case when two threads go throught the same edge in the blocking graph.

An alternative and more general way of representing the structure of jobs is to treat each task as a separate job and use a precendence graph to constrain their ordering. But for this particular problem, the two level job-task approach seems more natural.

The task-switching cost defined herein is called *sequence-dependent setup times* in scheduling terminology. In the notation of Graham, Lawler, Lenstra and Rinnooy Kan [1], the optimizing version of our problem is a more contrained version of $1|prec, S_{jk}|C_{max}$, i.e. a single machine(1 in the notation), precedence constrained scheduling($prec$) problem with sequence-dependent setup time(S_{jk}) and the goal being minimizing *makespan* (C_{max}), i.e. the time between the start of first job and the finish of last job. However, the resource constraint in our problem is not categorized by [1] or other work known to use.

Two similar problem in the literature that we are aware of is the *cache scheduling* problem [3] and *paging* problem. In the cache scheduling problem,

we are given a cache of k pages, a memory of m pages and a number of non-dividable jobs each having a different memory working set that has to be loaded into the cache before it starts. The cache-filling cost is proportional to the number pages to be loaded. The question is whether we can find a schedule of the jobs that has cache filling cost below a given value. In comparison to RCCAS, it treats jobs as undividable wholes instead of a sequence of schedulable tasks, and there is no resource constraint considered. However, it captures the behavior of cache memories in more details than RCCAS in that it uses working set of jobs instead of switching cost between tasks. The paging problem considers a cache of k pages, and a secondary storage of m pages. Then a sequence of memory page access is presented, one at a time. A *paging algorithm* needs to decide at each step which one of the k pages in the cache to discard in order to make room for the new page, if the page in not already in the cache.

2.1 Assumptions

It is clear that Resource Constrained Cache-Affinity Scheduling is not a direct capture of our real-world problem in Section 1. A couple of (unrealistic) assumptions are made to simplify the problem. These assumptions are often not true in real environments. We discuss their implications and possible work-arounds here. This serves as the validation for our problem formulation.

- All jobs are static, deterministic and known by the scheduler at the very beginning. All jobs are assumed to be available at time 0. These assumptions are false for an online scheduler. In other words, in real world we naturally want *online scheduling algorithms* that make decisions based on partial(past and current) knowledge of the system. Obviously that makes the problem much harder. Fortunately, static algorithms discussed here can still be useful or even better in some cases. For example, we can process the continously arriving jobs in batches, i.e. run the static algorithm at certain intervals and schedule all jobs waiting for execution together. In cases where throughput is more important than latency, this can be more favarable than scheduling jobs whenever they arrive because we can possibly achieve lower average cache cost and improve throughput.

- The cost to fill the cache only depends on the previous task scheduled. Actually the cost depends on the current content in the cache, which is determined by the full history of tasks scheduled. However, if the

tasks are "large" enough both in terms of code length and data access amount, this assumption will be a good approximation.

- The resource limit is *hard*. If the resource is memory, in real systems with virtual memory, this limit is indeed soft. Exceeding the normal memory limit will be possible but result in degraded performance because of paging.

- The maximum resource usage of a task is the same as the amount it uses when it finishes. This is not true in general, a task can allocate a lot of resource and free it before finishing. Therefore schedules generated by algorithms here may be actually infeasible. However, one workaround to get a feasible and generally good schedule is to reserve enough resource for all such "temporary" resource usage and rerun the algorithm.

3 Resource Constrained Cache-Affinity Scheduling is NP-complete

Theorem 2 Resource Constrained Cache-Affinity Scheduling *is NP-complete.*

Proof: The problem is a generalization of the classic *scheduling problem with sequence dependent setup times* problem $(1|S_{jk}|C_{max})$. If we set the resource constraints to ∞ and let each job contain only one task, then our problem becomes $1|S_{jk}|C_{max}$. The latter problem is easily shown to be NP-complete by doing a reduction from TSP. Given a TSP problem on graph $G = (V, E)$, construct a scheduling problem with n jobs and let and S_{jk} be the edge lengths in TSP. Then the cost of any schedule will be same as its corresponding tour in the TSP. ■

Knowing that the problem is NP-complete, we do not attempt to find a polynomial algorithm that produces optimal solution to the corresponding optimization problem Min Resource Constrained Cache-Affinity Scheduling. In the following sections, we first present a dynamic programming algorithm that gives the optimal solution. Although having exponential running time, it is still a significant improvement over the naive method. And its results serve as standards to compare other scheduling algorithms against. Then we present a series of heuristic algorithms. We are not able to analyze lower-bound performance of any of them. Instead we conduct

experiments to study their performance for different inputs, compared with the optimal solutions.

4 A Dynamic Programming Algorithm

In order to derive a dynamic programming algorithm for the problem. We observe that an optimal schedule has the following property.

Claim 3 *Suppose the optimal schedule is* π*. Then any prefix of* π*,* $\pi(1...k) = \{\pi(1), ..., \pi(k)\}$ *is also the optimal schedule for the same set of tasks with the additional constraint that* $\pi(k)$ *must be the last task scheduled. We call the corresponding optimization problem of the subset a* sub-problem*.*

Proof: From the definition of cost, we see that $cost(\pi) = cost(\pi(1...k)) + C(\pi(k), \pi(k+1)) + cost(\pi(k+1...|\pi|))$. If we fix the second part of the schedule from $\pi(k)$ to $\pi(|\pi|)$, then the last two terms in the equation above remain the same. We know that for the optimal schedule, any change will only increase or do not change the cost. Notice that any schedule of the sub-problem satisfying the resource constraint will result in a valid schedule of the whole problem, because resource consumption after the sub-problem is done does not depend on the internal schedule of the sub-problem. Therefore any valid schedule of the sub-probblem will have cost larger or equal to $cost(\pi(1...k))$. Thus by definition, $\{\pi(1)...\pi(k)\}$ is the optimal schedule for the sub-problem. ■

With this definition of sub-problems, we can iterate over k and do dynamic program. During each step, we generate optimal schedule for all possible schedule prefixes of length k with all possible last task, based on results of the last step. With precedence constraints of our problem, we do this by build a table of $cs(k, S, p)$ for each k, defined as follows,

$cs(k, S, p) =$ optimal cost of schedule in which job i has finished S_i tasks (S is a vector n non-negative integers, $S = \{S_i\}_{i=1,\cdots,n}$) and job p is the last job just scheduled. We have $\sum S_i = k$ and $1 \leq p \leq n$.

This table essentially lists optimal costs for all cases, because each S vector corresponds to a possible state of progress made so far for all jobs. The values of $cs(k, S, p)$ are obtained from the ones in the last step ($cs(k-1, S, p)$

as follows,

$$cs(k, S, p) = \text{Min}\{cs(k-1, S', p') + C(r^{p'}_{S'_{p'}}, r^{p}_{S_p}) \mid p' = 1..n\},$$

$$\text{where } S'_i = \begin{cases} S_i, & i \neq p \\ S_i - 1, & i = p \end{cases}$$

When the table for the last task ($k_0 = \sum |R_i|$) is obtained, the minimal cost c_0 of the whole problem can be obtained by,

$$c_0 = \text{Min}\{cs(k_0, S, p) \mid 1 \leq p \leq n\}$$

Now consider the running time of the algorithm. The total number of cases we have to generate is $n \prod_{i=1}^{n}(|R_i| + 1)$. This is because we are generating all prefixes in which job R_i can complete $0, 1, .., |R_i|$ tasks. Thus total number of prefixes is $\prod_{i=1}^{n}(|R_i| + 1)$. And for each prefix, there are at most n cases, with the last job scheduled being $1..n$. In order to generate each case, we look at n previous cases and find a minimum. Thus it needs n step. Suppose r is the average length of each job. Then, the algorithm takes $O(n^2 \prod_{i=1}^{n}(|R_i| + 1)) = O(n^2 r^n)$ steps. For the naive approach of enumerating all schedules, the running time is $O(\frac{(rn)!}{(r!)^n})$, because it is equivalent to the experiment of putting n different sets of balls, each containing r balls in the same color, in a line. Compared with the naive algorithm, the dynamic programming algorithm is a significant improvement.

5 Heuristic Algorithms

Here we present three simple and similar heuristic algorithms for this problem. Although they are still designed for the static version of the problem, it is easy to adapt them for the dynamic version in which jobs arrive continuously, because these algorithms make decision for each time independently.

The algorithms work greedily: at each time, they examine the situation and select a next job to schedule, go to the next time point and never backtrack. The selection of the next job is done with the simple heuristic of least cache-filling cost from current task[1]. If multiple jobs have the same cost, the winner is selected randomly to avoid always selecting the same one. Although simple, it turns out that this heuristic can still perform much better than traditional non-cache-affinity-aware thread/process schedulers.

[1] Trying more sophisticated heuristics is left as future work

However, the above discussion does not consider the resource constraint. Because we do not back-track, greedy selection of the most affine task (the one having least switching cost from current task) will possibly lead us to dead ends where we use up all resource and cannot finish the jobs. Even if that does not happen, using too much resource will force us to make bad choices because a more affine task may need too much resource than we have free and cannot be scheduled. This will produce sub-optmal schedules. Therefore we introduce an "admission control" mechanism into some of our algorithms. At any time we only select the next job from a set of "runnable" jobs that have been admitted and are not finished. The other addition is when we select the next task, we use knowledge of current and future resource requirements of jobs to make sure that we do not exhaust resource in the future (discussed shortly).

The first algorithm, *greedy_aggressive*, does not do any admission control. It looks at every job for the next task each time. However, to ensure the generation of a feasible schedule, it "reserves" some resource, with the amount being enough for any single job to complete, i.e. the *maximum* amount of resource any job may possibly allocate in the future. This amount, $m_{res}(k)$ for time k, is calculated as follows,

Let $m_{cur}(i,j)$ be the amount of resource job i is using after completing its jth task. $m_{cur}(i,j) = \sum_{j'=1}^{j} m(r^i_{j'})$. We introduce a new value $m_{max}(i,j)$ to be the *maximum* amount of resource it will use *in the future* after the completion of its jth task. We have $m_{max}(i,j) = \text{Max}\{m_{cur}(i,j')|j' = j..|R_i|\}$. Refer to figure 3 for the relationship between m_{cur} and m_{max}. m_{max} is a useful property of the job in that we can use it to pre-allocate resource for the job. Then the reservation resource amount $m_{res}(k)$ is,

$$m_{res}(k) = \text{Max}\{m_{max}(i, j^i_k) - m_{cur}(i, j^i_k)|i = 1..n\}$$

After getting $m_{res}(k)$, the scheduling decision is made in the following way. 1) If there is any next task that does not need to use the reserved resource, then select the most affine task within all these tasks. 2) If all incompleted jobs need to use the reserved resource, then select the job that currently uses most resource and keep executing it until the first condition holds again or it finishes. Then go to 1 or 2 respectively.

The second algorithm, *greedy_conservative*, admits new jobs very conservatively and make sure resource is always enough. At any time, every job not in "`runnable`" set is examined to see if its maximum possible resource usage added to the current total maximum resource usage of all jobs in `runnable` exceeds total resource. Again, we use m_{max} as the amount of

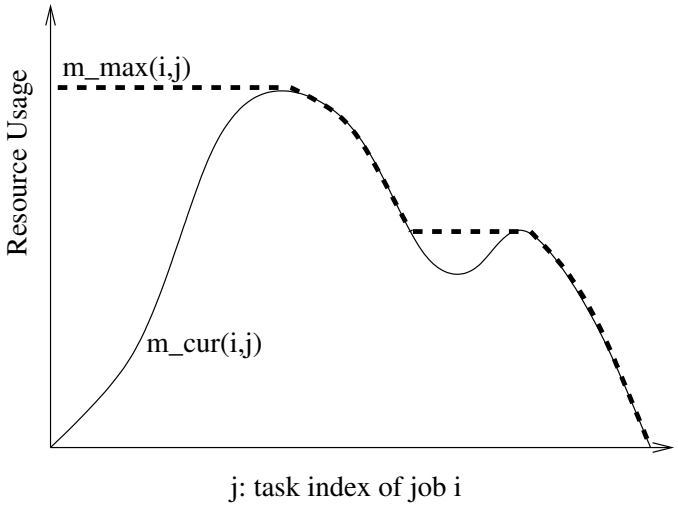

Figure 3: Calculation of m_{max} from m_{cur}

resource each job will need in the future. For each $R_i \notin$ `runnable`, we check whether $m_{max}(i,0) + \sum_{i' \in \texttt{runnable}} m_{max}(i', j_k^{i'}) \leq M$. If it is true, then we admit the job into the `runnable` set. It is easy to see that with this conservative admission control, the algorithm is free to choose any job as next job when making scheduling decisions and "reservation" is not necessary.

The third algorithm, *greedy_predictive*, trys to be neither too conservative nor too aggressive. We observe that "admission control" is generally useful because getting into resource scarcity and be forced to execute a single job, as in 2nd case of algorithm 1, is sub-optimal. But the admission control of the second algorithm is too conservative in a lot of cases, because maximum resource usage is only achieved at one or a few points in the job's life time. Intuitively, the $\hat{k}-median$ of a job's resource usage will be better estimation of the average resource requirement of the job than the maximum value. We use this in algorithm 3, with $\hat{k}$ being a parameter that we set to $\frac{1}{3}n$ in our experiments. Notice that with this admission control policy, we need to do reservation of resource just as in algorithm 1 because now we are not guaranteed to generate a feasible schedule by greedily next job selection.

The time complexity of these algorithms are low. Basically to make greedy decision for each step, $O(n)$ steps are needed. It also takes $O(n)$ to calculate the resource reservation amount, with m_{max} and m_{cur} precomputed once in $O(rn)$ time. Therefore, overall time complexity for all three algorithms are $O(rn^2)$.

6 Experiments

In this section we present simulation results of the algorithms on several synthesized problems. An alternative way of evaluating the algorithms is to run them on problems extracted from real programs and environments. However, due to time limit, we have not been able to get these potentially more useful problems. Instead, we hand-picked several problems corresponding to extreme or typical "topology" of the blocking graph described in Section 1, in the hope that these experiments will show overall performance and pecularities of the algorithms.

We ran all algorithms on 3 problems,

(a) *Single:* 6 copies of a single kind of job containing 6 tasks (numbered task 1 to 6). Switching cost between two tasks is set up as: 0 between the same task, 1 between adjacent tasks (1 to 2, 5 to 4 etc) and 2 otherwise. Resource allocation amount of the tasks are 1, 2, 4, 13, -13, -7.

(b) *Multiple:* 3 copies of job A and 3 copies of job B. A is longer (7 tasks) and uses more resource (max 20). B is shorter (4 tasks) and uses less resource (max 4). The switching cost set up is the same with *single*.

(c) *Random:* 6 different jobs. Each task has random total resource consumption uniformly distributed between 0 to 20. The switching cost is the same with *single* except that for not adjacent/same tasks the cost is not 2, but a random value uniformly distributed between 1 and 3.

Intuitively *single* models scheduling a lot of jobs of the same kind. *Multiple* models a mixture of longer and shorter jobs. *Random* models more complex situation when every job is different.

Figure 4 shows the results of different algorithms on *single*. X axis is the amount of total resource. Y axis is the cost of the schedule. We compare the three heuristic algorithms to *run-to-completion*, in which the next job starts after the current one finishes all its tasks, and *round-robin*, which is a simple round-robin scheduler that always execute the next task of the next job after the current task finishes, without knowledge of cache-affinity. *Optimal* shows the result of the global optimal schedule obtained with the dynamic-programming algorithm.

For *single*, it is shown that all three heuristic algorithms generate near-optimal schedules. This is partly because the problem matches our cache-affinity heuristic well. Apart from same tasks and adjacent tasks, all other

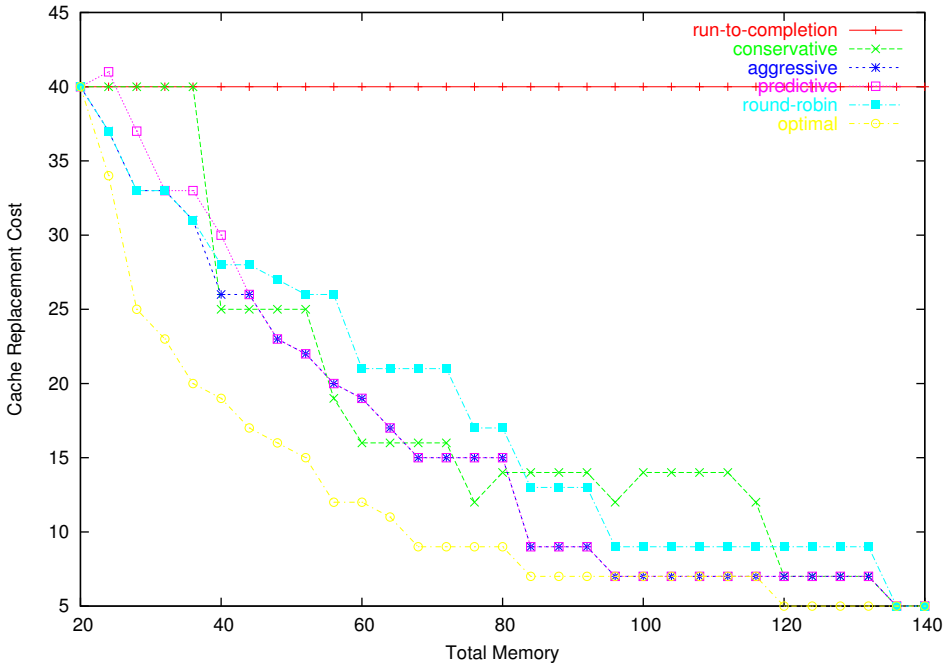

Figure 4: Cost of different algorithms on single

tasks have same switching cost. There are few cases when a greedy choice is a terrible choice in this problem. Also shown is that *aggressive* and *predictive* perform better than *round-robin* and all three much better than *run-to-completion*. This is not surprising. Moreover, *round-robin* is somewhat favored here because the fact that all jobs are available for scheduling at the same time results in *round-robin* generating pretty similar schedules to the heuristic algorithms. Without this unreal assumption, we expect the heuristic algorithms to perform much better than *round-robin*.

Multiple, shown in figure 5, gives similar results showing the heuristic algorithms perform better than *run-to-completion* and *round-robin*. One surprising thing shown in this figure is that *conservative* tends to perform better than *aggressive* and *predictive* when total resource is low. Although both this experiment and the next one show this pheonomenon, we conjecture that this is only an artifact of our experiment setup and will not be true for longer (containing more tasks) jobs whose tasks have more diversified resource requirement, because in those cases accepting jobs by their maximum resource requirement will result in too much loss in concurrency. This is left as future work to be done after we optimize our dynamic pro-

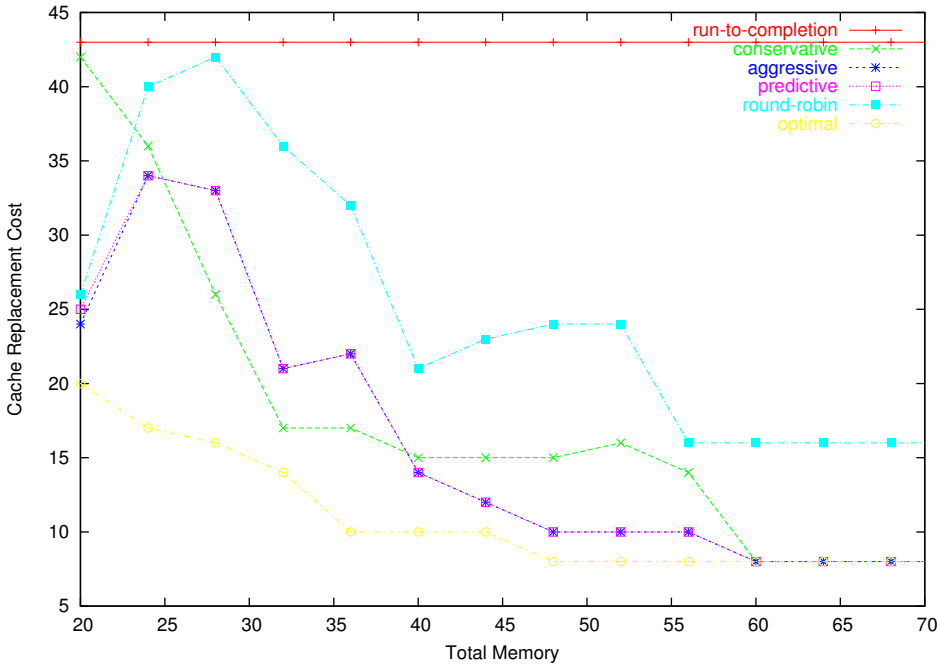

Figure 5: Cost of different algorithms on multiple

gramming algorithm to solve larger problems.

The results for *Random* is shown in figure 6. Here *round-robin* performs much worse because each job is different and the round-robin schedule has no more locality than *run-to-completion*. When resource is enough, the costs of schedules generated by the heuristic algorithms are 63% of that of *round-robin* and 131% of the optimal. Therefore, for the random case, the algorithms provide a pretty good improvement over traditional non-cache-affinity scheduling algorithms.

7 Conclusion

We defined the RCCAS problem and showed its NP-completeness. We presented a dynamic programming algorithm producing optimal solutions for the optimization problem. We also presented three greedy algorithms which are shown to perform better than traditional non-cache-affinity aware schedulers in our simulations.

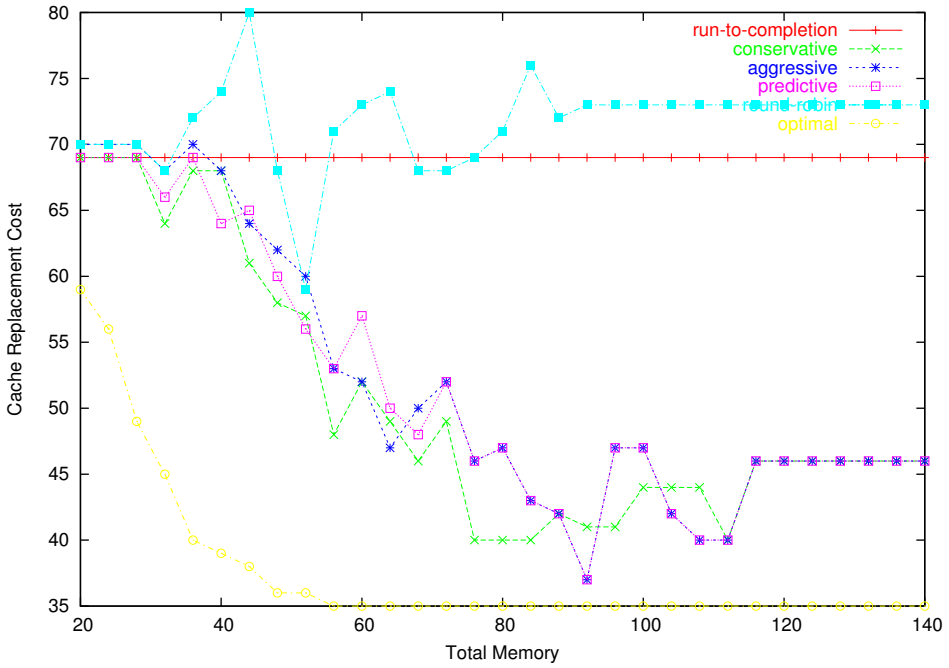

Figure 6: Cost of different algorithms on random

References

[1] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. H. G. R. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Ann. Discrete Math* (1979), pp. 5:287–326.

[2] Harizopoulos, S., and Ailamaki, A. Affinity scheduling in staged server architectures. Tech. Rep. CMU-CS-02-113, Carnegie Mellon University, 2002.

[3] Hoffman, T. L. Cache scheduling. Master's thesis, University of Victoria, 1997.

[4] Larus, J. R., and Parkes, M. Using cohort scheduling to enhance server performance. In *Usenix Annual Technical Conference* (June 2002).

[5] Philbin, J., Edler, J., Anshus, O. J., Douglas, C. C., and Li, K. Thread scheduling for cache locality. In *Architectural Support for Programming Languages and Operating Systems* (1996), pp. 60–71.

[6] Welsh, M., Culler, D. E., and Brewer, E. A. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles* (2001), pp. 230–243.