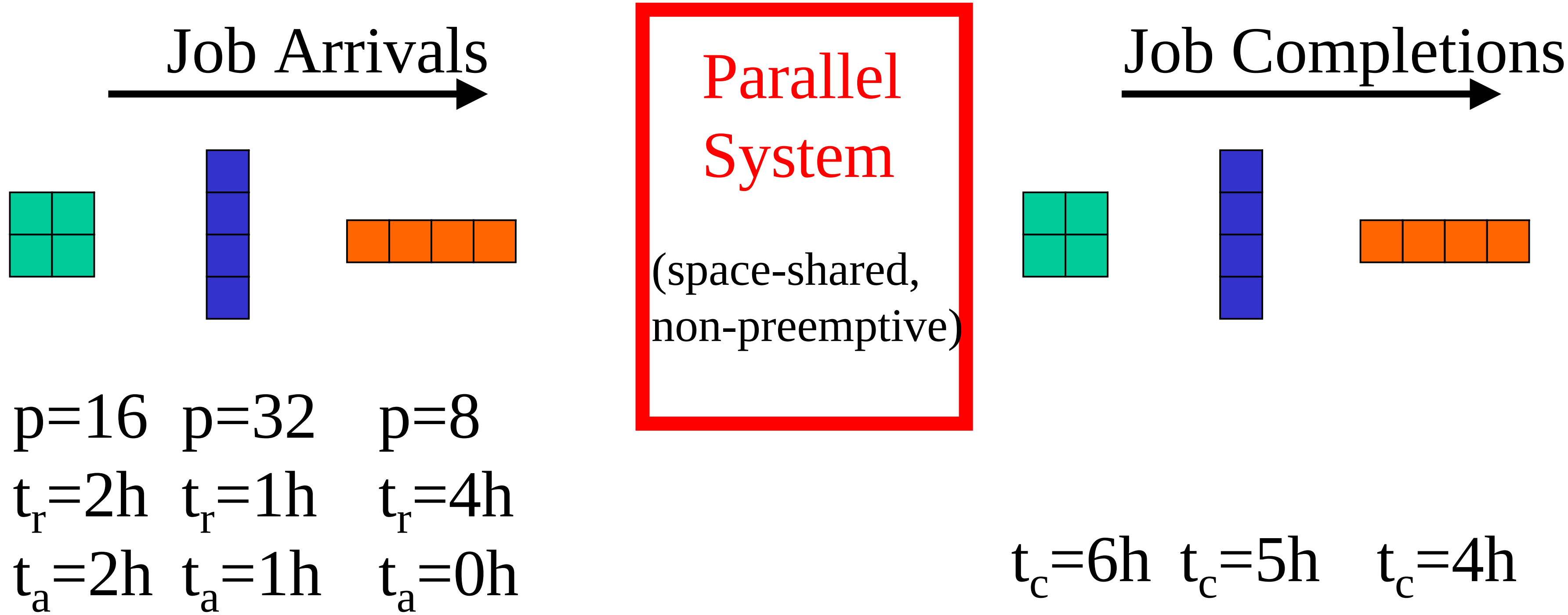
A Robust Scheduling Strategy for Moldable Jobs

Sudha Srinivasan, Savitha Krishnamoorthy and P. Sadayappan

The Ohio State University, USA

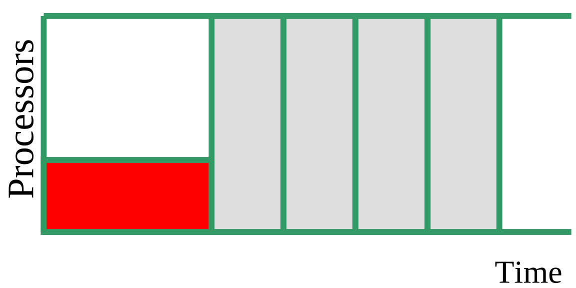
Background: Job Scheduling



- Goals:
  - Minimize job response time (t_c-t_a)
  - Avoid job starvation; Provide fairness
  - Maximize system utilization
- First-Come-First-Served policy is fair but utilization is low
- Back-filling is used to improve utilization and response time

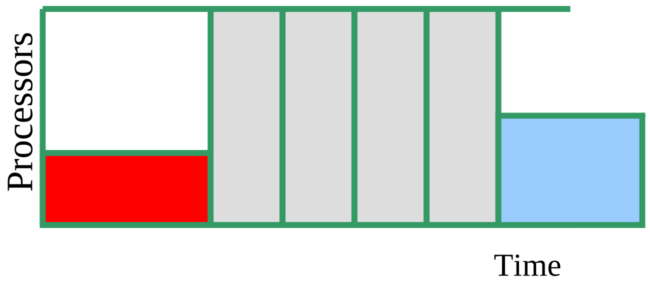
Backfilling

- A later arriving job is allowed to overtake previously queued jobs if its early execution will not delay others



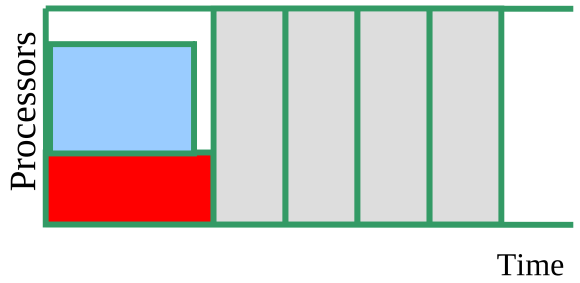
Backfilling

- A later arriving job is allowed to overtake previously queued jobs if its early execution will not delay others



Backfilling

- A later arriving job is allowed to overtake previously queued jobs if its early execution will not delay others



Conservative vs. Aggressive Backfill

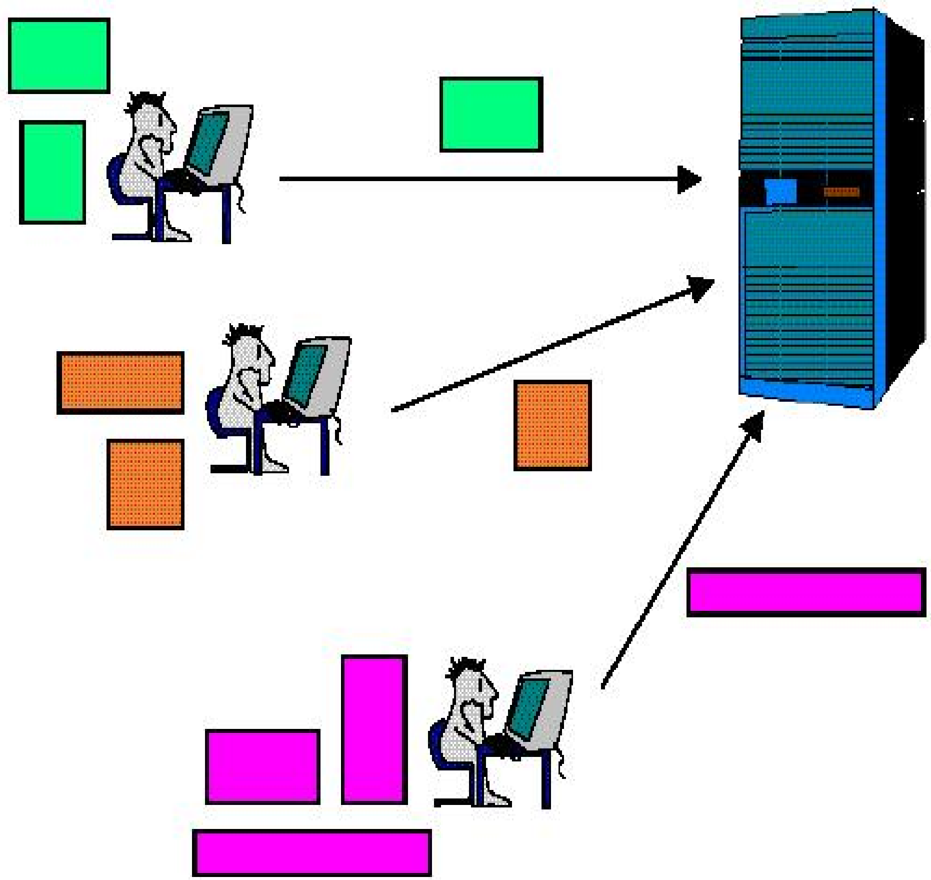
- Conservative
  - Every job is given a reservation when it enters the system; a job is allowed to backfill only if it does not violate any of the reservations.
- Aggressive
  - Only the job at the head of the queue is given a reservation; a job is allowed to backfill if it does not violate this single reservation

Moldable Job Scheduling

- With current job schedulers, each job requests a specific number of processors, e.g. 64
- But most jobs are not so rigid, but are "moldable", e.g. the same job could run on 32 or 64 or 128 processors
- User makes choice at job submission time based on perceived trade-off between run time and wait time:
  - Large number of processors => lower execution time, but wait time will likely be longer
  - Small number of processors => lower wait time, but execution time is higher
- Why not let the scheduler decide the number of processors for each job, to optimize response time?
  - How many processors should each job be allocated?

Greedy Partition Size Choice

- Earliest proposed approach: Submit-time Greedy Choice + Conservative Backfill (Cirne)
- When job is submitted, earliest feasible reservation is determined for several possible partition sizes
- Partition size with earliest completion time is selected; job shape is frozen and reservation is made



Greedy Moldable Scheduling: Job Profile

Partition size	% Rigid Jobs	% Greedy Moldable Jobs
1	44.46	23.38
2	7.88	3.48
3-4	13.1	4.24
5-8	9.88	1.1
9-16	12.34	0
17-32	6.54	0
33-64	3.44	1.02
65-100	1.44	0.72
129-256	0.68	1.4
>256	0.24	64.66

- Rigid job trace (CTC from Feitelson's archive) used to evaluate Greedy moldable scheduling
- The majority of parallel jobs tend to choose very wide partition sizes with the Greedy scheme
- Light jobs (low processor-time product) suffer due to reduced backfilling opportunities

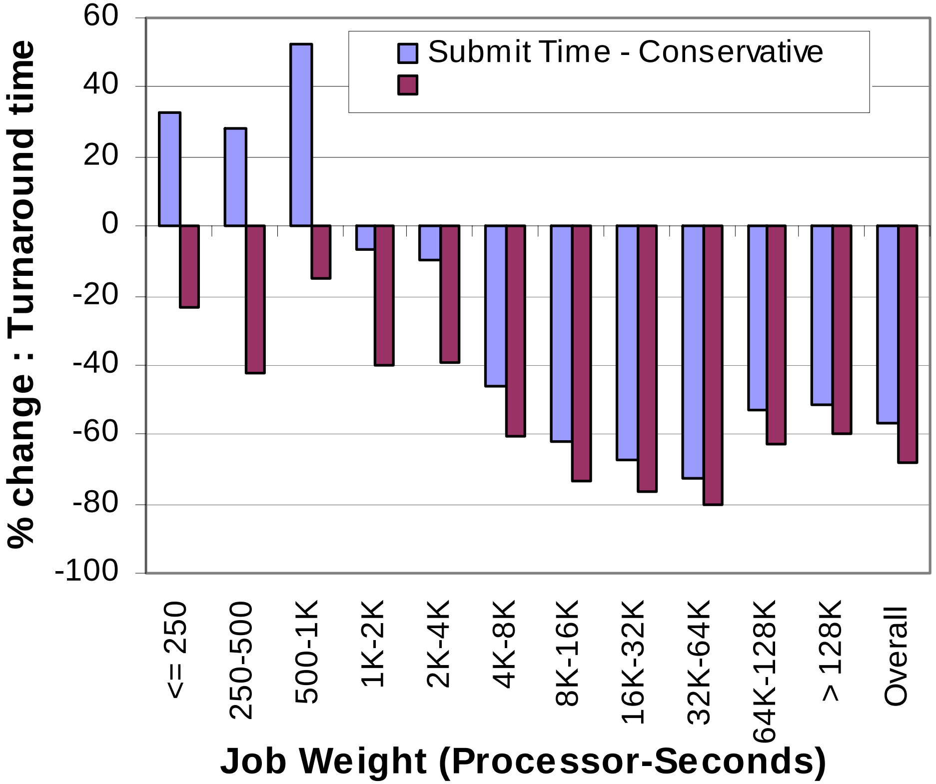
Work-Proportional "Fair-share" Limits

- Locally greedy choice for a job is often very wide:
  - Fills up processors, and reduces backfill opportunities
  - Uses more processor-seconds than narrow choice; lowers efficiency and increases offered load to system
- Use a load-sensitive "fair-share" limit to bound number of processors allowed for a job:
  - High limit at light loads and low limit at high loads

$$FS(i) = Nprocs * \frac{\mathrm{Pr}\,ocessor\_Seconds(i)}{\sum_j \mathrm{Pr}\,ocessor\_Seconds(j)}$$

$$Maxprocs(i) = Min(0.9 \times Nprocs, FS(i))$$

- Submit-time Conservative Fair-share Strategy
  - Similar to submit-time greedy strategy except for limit on max-procs for job
- Schedule-time Aggressive Fair-Share Strategy
  - Defers job's partition choice to start-time; hence can use aggressive back-fill



% change : Turnaround time
Submit Time - Conservative
60
40
20
0
-20
-40
-60
-80
-100
<= 250
250-500
500-1K
1K-2K
2K-4K
4K-8K
8K-16K
16K-32K
32K-64K
64K-128K
> 128K
Overall
Job Weight (Processor-Seconds)

Enhancing Robustness

- Previously proposed moldable scheduling schemes
  - showed improved overall performance, but
  - sometimes improved some job categories (e.g. heavy jobs) while degrading other categories
- A robust moldable scheduling strategy should:
  - improve performance of all (most) job categories without significant detriment to any category
  - work well at low system loads as well as high loads
  - work well for jobs with high as well as poor scalability
  - work well under restrictions on partition sizes allowed for jobs (e.g. minimum number due to memory requirements or max. # due to poor scalability)
- Focus of this paper: making moldable schemes robust

Assessing Robustness of Scheduling Strategies

- We used Downey's model for job scalability:
  - Parameter σ models the coefficient of variance in parallelism:
    - $\sigma = 0$ => perfect job scalability; $S(n) = n$
    - Higher the value of σ, poorer the scalability
  - Given a σ for a job, and the run-time and number of processors from job trace, the run-time for a different processor count is found using Downey's model
- Range of allowed processor counts: limit by Range Factor (RF):
  - RF=1 => all partition sizes between 1 and Nprocs allowed
  - RF=2 => for a N-proc trace job, only one-half the range allowed, i.e. [(N+1)/2 .. (N+P)/2]
- Load variation: scaled all job run-times by a Load-Factor (LF):
  - With LF = 125%, each job's run-time is increased by 25%

Modifying the Fair-Share Strategy

- The Weight-Proportional Fair-Share (WPFS) model:
  - tends to assign #procs α job's weight, i.e. proc-seconds product
  - thus it attempts to equalize the run-times of all jobs
  - run-time dominated heavy jobs benefit while light jobs suffer
- Another extreme possibility: Equi-Processor Fair-Share (EPFS):
  - equal number of processors to all the jobs, light or heavy
  - would benefit light jobs but severely hurt heavy jobs
- These two schemes "equalize" jobs along orthogonal dimensions:
  - WPFS equalizes job run-time; #procs proportional to job weight
  - EPFS equalizes #procs; job run-time is proportional to job weight
- Considered a more "balanced" fair-share scheme that "split" a heavier job's weight across both the processor and time dimension:
  - A job with 4 times another job's weight is given twice as many processors, so that its run-time is twice as high

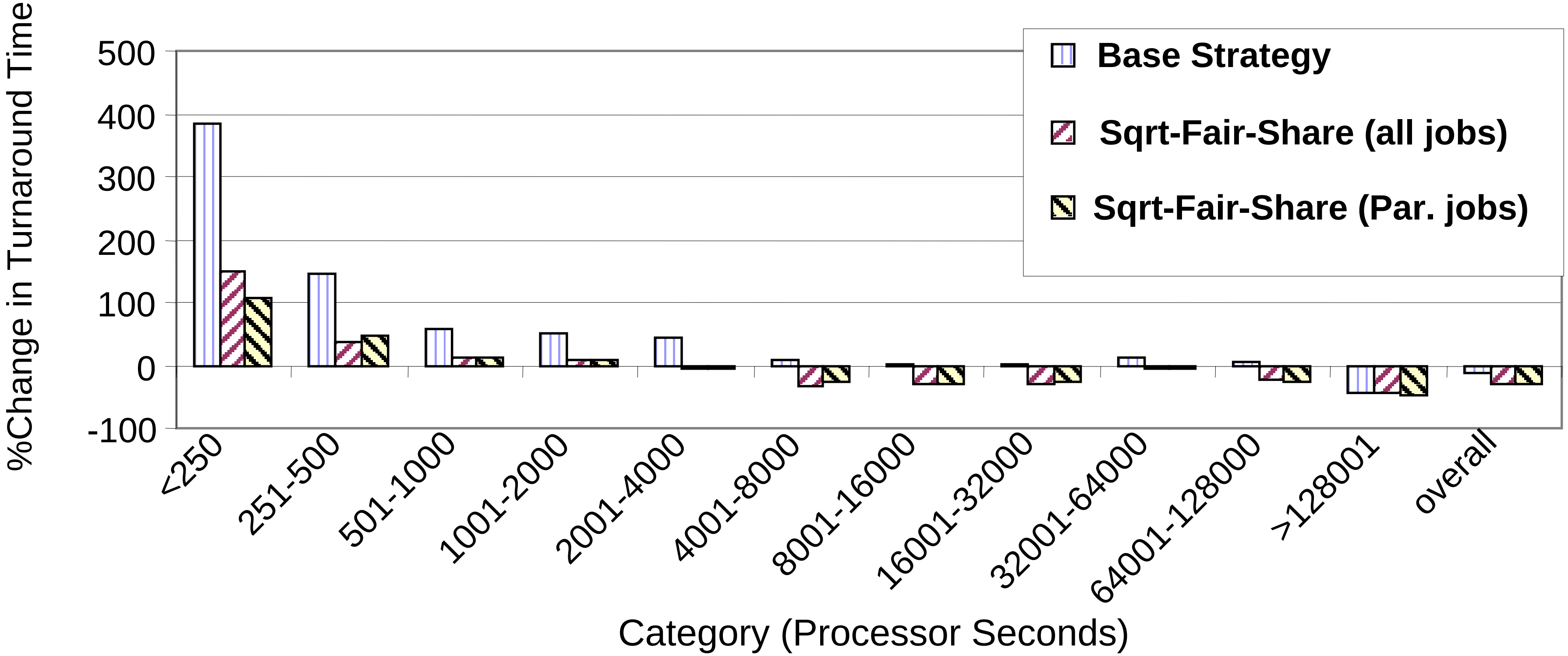
Square-Root Fair-Share Allocation

- A balance between the two extremes of WPFS and EPFS is achieved by the following square-root fair-share formula:

$$FairShare(i) = \left(\frac{\sqrt{Weight_i}}{\sum_j \sqrt{Weight_j}} \right) Numprocs$$

- Two variants:
  - Consider all jobs in determining job's fair-share limit
  - Consider only parallel jobs for computing fair-share limit

Square-Root Fair-Share Allocation



Performance with Sqrt-FS (SDSC log): Sigma=0, LF=100; RF=1

- Enhancements compared with Base Strategy: Schedule-time aggressive, Weight-Proportion Fair-Share; LF=100%, RF=1, σ=0
- All but the heaviest category improves; but lightest four categories are still worse than with rigid scheduling

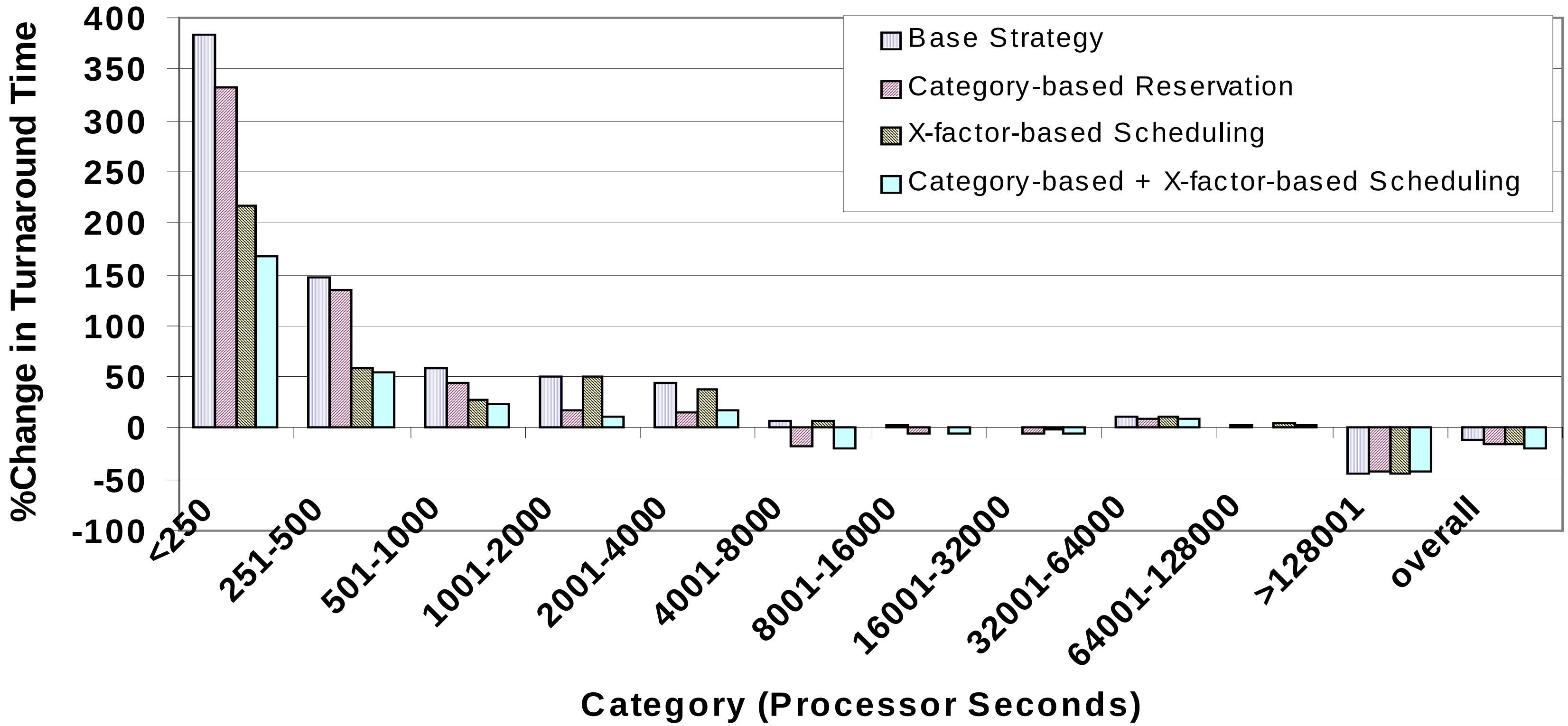
Enhanced Reservation Policies

- Category-based reservation:
  - Jobs partitioned into categories based on their weight
  - Multiple queues; reservation for job at head of each queue
- Xfactor-based reservation:
  - eXpansion-factor computed for each job, based on Expected Sequential Run Time (ESRT):

$$Xfactor = \frac{ESRT + Queue\_Time}{ESRT}$$

  - Job is given reservation when XFactor exceeds threshold

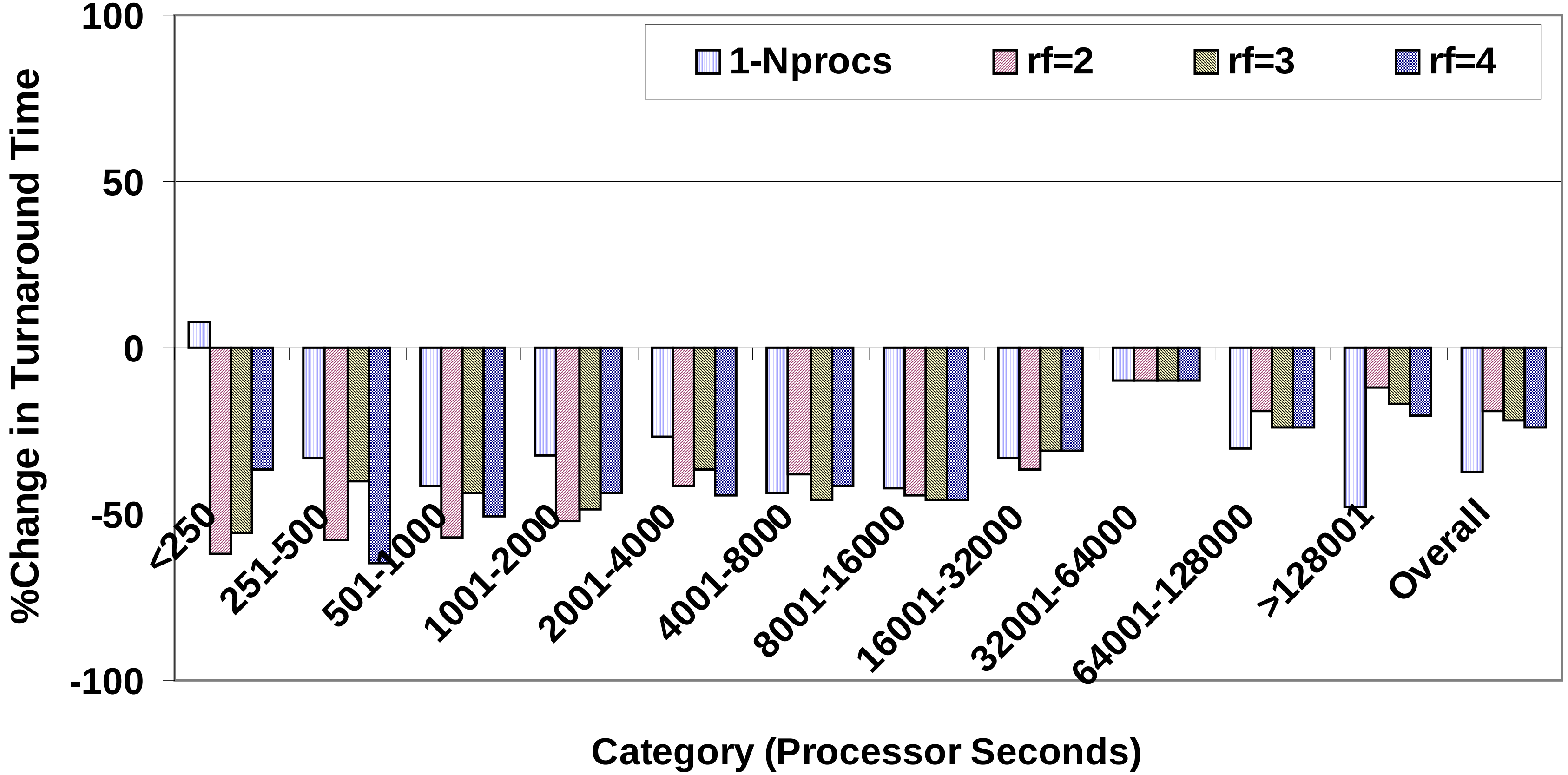
Enhanced Reservation Policies: Performance



LF=100%, RF=1 and Sigma=0 (SDSC log)

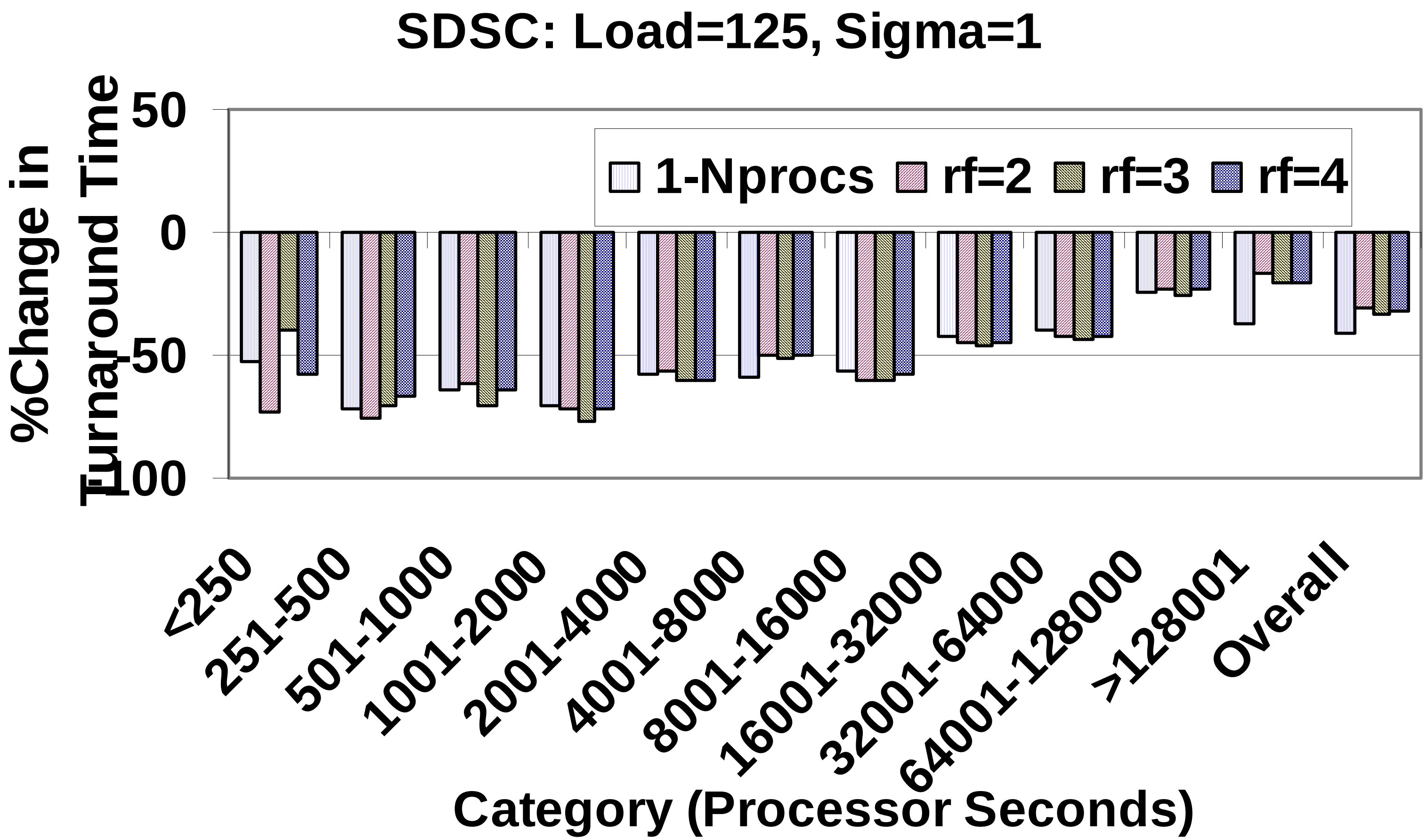
- Category-based reservation improves all but the heaviest category
- X-factor-based reservation provides good improvement for lightest (three) categories; little/no improvement for others
- Combining category-based and X-factor-based reservations provides overall improvement corresponding to the better of both

Combining Sqrt-FS & Enhanced Reservations



- All job categories do well when enhancements combined
- Benefit for light categories better than "sum of parts"
  - Sqrt-FS limits heavy jobs' procs: more back-fill for light jobs
  - Light jobs benefit from category-wise and Xfactor reservations

Evaluation under Imperfect Scalability

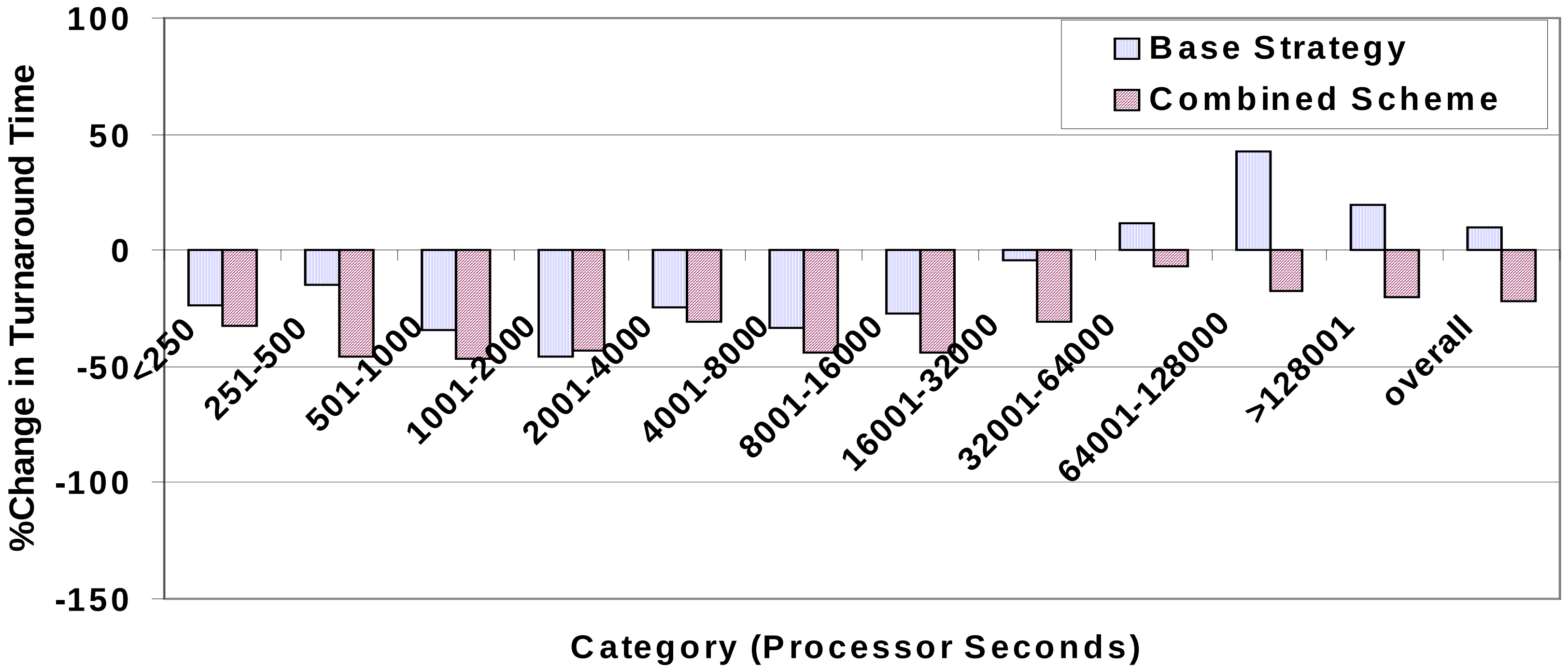


- Performance of all job categories improves, for various values of range factor

Evaluation with "Real" Scalability Data

- Evaluations so far used real job workload traces from Supercomputer Centers, but hypothetical scalability parameters (σ and RF)
- We also used NAS Parallel Benchmarks (NPB) performance data to model scalability
  - 8 applications' speedup characteristics on CRAY-T3E
  - Processor request-range is assumed to be the range represented by the available data for each application
  - Each job in SDSC job trace assumed to have scalability characteristic of one of the 8 NAS Parallel Benchmarks
  - Interpolation used to determine the execution times on different numbers of processors

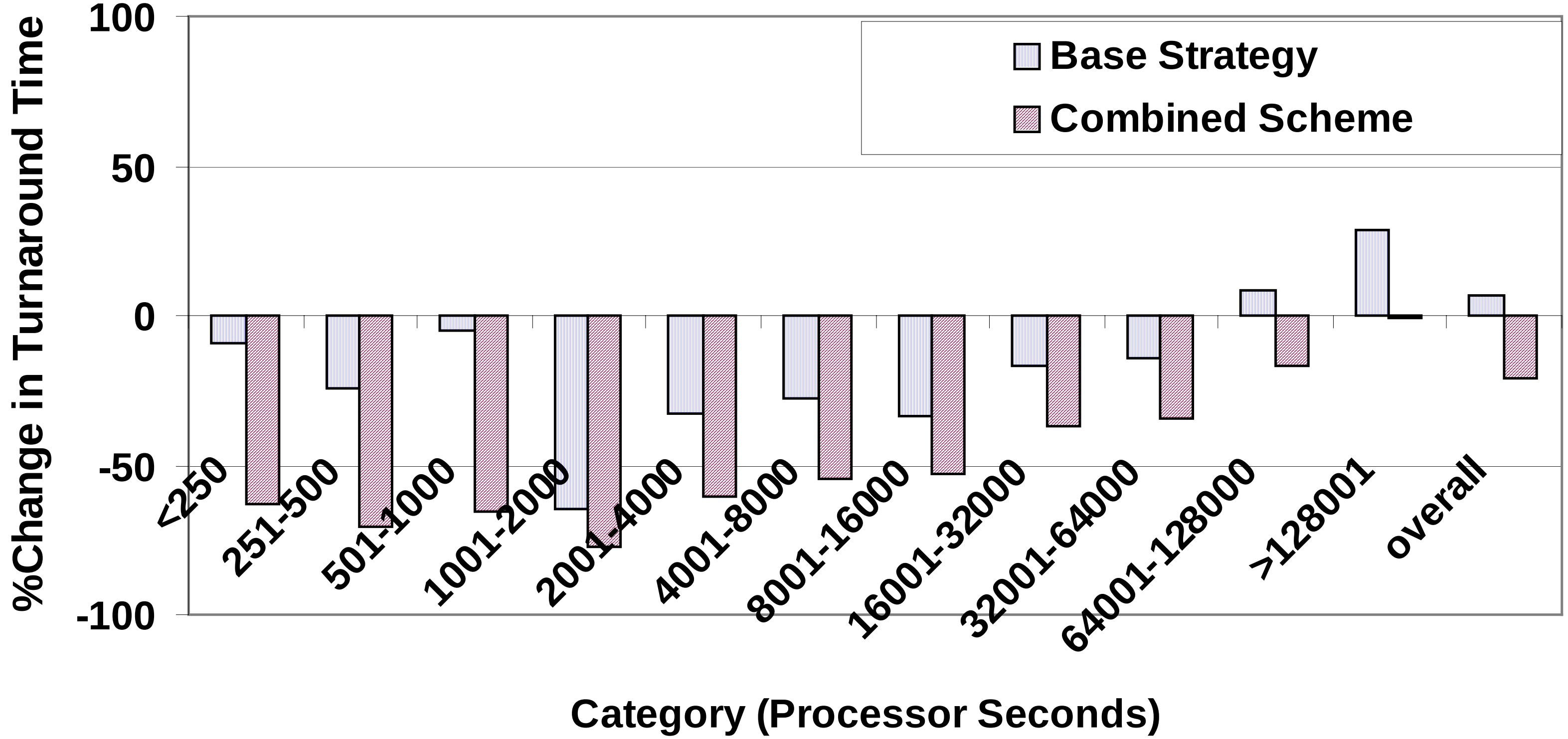
NPB-Based Scalability: Performance



Performance: LF = 100; SDSC log; NPB-based Scalability

- Performance of all job categories improves with new moldable strategy

NPB-Based Scalability: Performance



Performance: LF = 125; SDSC log; NPB-based Scalability

- Performance of all job categories improves with new scheme; better than base scheme for all categories

Conclusions

- Previous moldable scheduling schemes
  - Shown to be effective under some scenarios
  - But detrimental for some job categories especially when scalability is poor
- Modifications proposed to enhance robustness
  - Modified fair-share allocation based on square-root of job weights
  - Category-wise reservations; Xfactor-based reservations
  - Each of these enhancements provided some benefits, but when combined provided significantly greater benefits
  - All job categories improve over rigid scheduling, over wide range of simulation parameters