Processor Allocation in Multiprogrammed Distributed-Memory Parallel Computer Systems

Vijay K. Naik*
Mail Stop 28–234
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598.
vkn@watson.ibm.com

Sanjeev K. Setia†
Department of Computer Science
George Mason University
Fairfax, VA 22030.
setia@cs.gmu.edu

Mark S. Squillante
Mail Stop H4–B10
IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598.
mss@watson.ibm.com

*Research was partially supported by NASA under the HPCCPT-1 Cooperative Research Agreement NCC2-9000.
†Research was partially supported by the National Science Foundation under grant CCR-9409697.

Abbreviated Title:
Processor Allocation in Distributed-Memory Systems

Corresponding Author:

Dr. Vijay K. Naik
Mail Stop 28–234
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
e-mail: vkn@watson.ibm.com
Tel: 914-945-2181
Fax: 914-945-2141

Abstract: In this paper, we examine three general classes of space-sharing scheduling policies under a workload representative of large-scale scientific computing. These policies differ in the way processors are partitioned among the jobs as well as in the way jobs are prioritized for execution on the partitions. We consider new static, adaptive and dynamic policies that differ from previously proposed policies by exploiting user-supplied information about the resource requirements of submitted jobs. We examine the performance characteristics of these policies from both the system and user perspectives. Our results demonstrate that existing static schemes do not perform well under varying workloads, and that the system scheduling policy for such workloads must distinguish between jobs with large differences in execution times. We show that obtaining good performance under adaptive policies requires some *a priori* knowledge of the job mix in these systems. We further show that a judiciously parameterized dynamic space-sharing policy can outperform adaptive policies from both the system and user perspectives.

List of symbols used in the manuscript

P m l s P_l P_m P_s $avail_l$ $avail_m$ $avail_s$ $avail_{sys}$ min_l
min_m min_s s m l k n_s n_m n_l pe_s pe_m pe_l $\mathtt{Qlen}_k$ t_m
t_l t_s req_{appl} T M L f t_{appl} Z
p_s p_m p_l cv_s c_s c_m c_l J_n S_k x_n U_k D_k D x
$U_{DP,k}$ $U_{MAP,k}$ S^2 $\mathtt{Qlen}_m$ $\mathtt{Qlen}_s$ $\mathtt{Qlen}_l$

$\surd$
$\sum$
$\cdot$
$\rfloor$
$\in$
$*$
λ
μ
$\equiv$
$\geq$
$\lim_{n\to\infty}$
$\tau_{U,m}$
$\leftarrow$
$\lfloor$
$\wedge$
τ_S
$\tau_{U,s}$
$\tau_{U,l}$
$\tau_{S,s}$
$\tau_{S,m}$
$\tau_{S,k}$
$\tau_{S,l}$
τ_U
$\tau_{U,k}$
δ_n
α_n

Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Computer Systems

Abstract: In this paper, we examine three general classes of space-sharing scheduling policies under a workload representative of large-scale scientific computing. These policies differ in the way processors are partitioned among the jobs as well as in the way jobs are prioritized for execution on the partitions. We consider new static, adaptive and dynamic policies that differ from previously proposed policies by exploiting user-supplied information about the resource requirements of submitted jobs. We examine the performance characteristics of these policies from both the system and user perspectives. Our results demonstrate that existing static schemes do not perform well under varying workloads, and that the system scheduling policy for such workloads must distinguish between jobs with large differences in execution times. We show that obtaining good performance under adaptive policies requires some *a priori* knowledge of the job mix in these systems. We further show that a judiciously parameterized dynamic space-sharing policy can outperform adaptive policies from both the system and user perspectives.

1 Introduction

Distributed-memory multiprocessors have matured to deliver excellent performance on many large-scale scientific and engineering applications, with the promise of continued cost-effective growth in performance. These developments in both hardware and software technology have made such high-performance parallel systems an important computing base. To achieve the best overall performance, however, the resources of these systems cannot be used in an inefficient manner, such as dedicating the entire machine to a single application.

Although many computationally-intensive scientific and engineering applications have exhibited extremely good performance on dedicated distributed-memory parallel computers, it is less well understood how to achieve the best performance on a system that is shared by multiple jobs. On one hand, users expect their individual jobs to receive excellent performance. On the other hand, system resources must be judiciously allocated among the various demands of different jobs to achieve the best overall performance. This objective raises a number of fundamental resource management issues for large-scale parallel computing environments.

One such issue, and the focus of our study, is the allocation of processors among the parallel jobs submitted for execution in a manner that maximizes throughput and minimizes response time while maintaining fairness among the jobs. This is complicated by the fact that system workloads for most large-scale scientific computing environments consist of a mixture of jobs with large variations in resource requirements, which are often unpredictable. A better understanding of the corresponding performance tradeoffs is needed to design scheduling policies that will fully realize the potential performance benefits of parallel computing environments.

In the past, distributed-memory parallel systems have provided only rudimentary facilities for partitioning, i.e., *space-sharing*, the processors among arriving jobs. Specifically, the *static partitioning* of the processors into a fixed number of disjoint sets has been most often used to allocate resources to individual jobs. On more recent general-purpose distributed systems, processors can be regrouped and processor partitions can be formed just prior to scheduling a job [1, 5]. On such systems, *adaptive partitioning* policies have been considered, where the number of processors allocated to a job is determined based on the system state at job arrivals and departures. Several batch processing schedulers (e.g., EASY, LoadLeveler, and PBS) have been developed to take advantage of this flexibility in the underlying system (see [12, 34] and references therein). Finally, by modifying the size of the partition allocated to a job during its execution, a *dynamic partitioning* policy can address the potential problems of subsequent workload changes at the expense of increased overhead due to data migration and reconfiguration of the application.

In this paper, we restrict our attention to these three forms of scheduling policies and we extend these approaches to examine the benefits and limitations of exploiting information on the resource requirements of the jobs in the system. We assume that arriving jobs provide some information about their expected execution times on a given set of processors. The jobs are then divided into multiple classes based on this user-supplied information. For the static and adaptive policies, the system processors are partitioned on a per-class basis, and processors are assigned to a job based on its class. The dynamic policy, on the other hand, uses the class information to decide which jobs can be *reconfigured* (i.e., changing the number of allocated processors) while they are executing.

The performance characteristics of these scheduling policies are analyzed based on steady-state performance measures from both the system and user viewpoints. A job mix representative of large-scale computational fluid dynamics applications is used for our simulation experiments. This job mix was chosen primarily to represent a realistic workload from high-performance computing environments, and not because of any special features of this class of applications. Thus, many of the trends observed in this paper are equally applicable to other large-scale scientific and engineering workloads.

Our results demonstrate that significant performance benefits can be obtained by incorporating resource requirements in scheduling decisions, even when this information is coarse. While adaptive policies offer more flexibility than static policies, system performance under both of these approaches depends on the inter-class partitions which need to be determined *a priori* based on knowledge about the job mix. We also show that a judiciously parameterized dynamic partitioning policy can outperform adaptive policies from both the system and user perspectives. Our dynamic policies do not necessarily require prior information about the workload, and they are the preferred space-sharing approach in unpredictable environments.

The remainder of this paper is organized as follows. In the next section, we discuss the general parallel computing environment assumed in this study. The scheduling policies, application workload and performance evaluation methodology used in our study are described in Sections 3, 4 and 5, respectively. Some of the results from our simulation experiments are presented in Section 6, and Section 7 discusses related work. Section 8 concludes the paper with final remarks on the practical implications of our policies and results.

2 Characteristics of the Parallel Environment

We consider a parallel computing environment that is fairly generic and is based on the characteristics of large-scale parallel platforms, such as IBM's RS/6000 SP systems and Cray's T3D and T3E systems [1, 5]. We focus on these systems as they have proven to be commercially viable.

The parallel system assumed in our study consists of a large number of processors each with its own local memory. In our simulation experiments the system has 512 identical processors, each having 512MB of local memory and each sustaining 100 MFLOPS in the computationally intensive sections of an application. The underlying interconnection network is assumed to provide uniform service in moving data from one processing element to another; i.e., with respect to the cost of data movement, all processors are equidistant from one another. This network is assumed to achieve an average message transmission speed of 100 MB/sec. The latency per message, as experienced at the application level, is 5 μsec.

The system supports space sharing of the resources among multiple jobs; i.e., multiple jobs can be executing on different partitions of the system at the same time. To execute a parallel job, partitions of any size (from 1 to the maximum number of processors) can be carved out. Two jobs executing on separate processor partitions are processed independently and are not synchronized in any manner. After a job is scheduled to run on a set of processors, a new processor partition can be created from the remaining set of idle processors to schedule another job. Similarly, when a job departs, its allocated processors become immediately available for other jobs.

The parallel system incorporates an event-driven scheduler that responds to job related events such as arrivals and departures, and to certain internal timers that are specific to the scheduling policies. The function of the scheduler is to allocate processors to new jobs, and to running jobs in the case of dynamic partitioning, according to a particular space-sharing policy. The specific details of the space-sharing policies examined in this study are based on our various assumptions about the system, application and workload characteristics. We describe the specific assumptions in the following sections.

The job mix considered in this paper consists of parallel jobs. Moreover, we assume that some of the parallel jobs are capable of executing on more than one processor partition size; e.g., a job can run on partitions of 8 and 12 processors. We refer to each such processor partition as a *valid processor partition* (*vpp*) for that job. In general, the set of valid processor partitions (or *vpp set*) for a parallel job is dependent on its implementation, and we assume that the system "knows" about the vpp set for each job. In particular, each job arrival initially declares its vpp set and its expected execution time on each partition. A job also indicates a preferred partition size (usually the maximum) from its vpp set. The scheduler then makes use of this information in classifying, prioritizing and scheduling the job.

We also assume that the memory requirements of the jobs submitted to the system are such that the entire address space of each task of a parallel job is resident in the physical memory of the processor where it is executing. These assumptions are justified by the results of several studies [26, 30] that have shown the detrimental effects of demand paging on system and job performance. For these reasons, we consider only those applications that do not exceed the available physical memory on the allocated processors. The vpp set for each application is chosen accordingly.

Although neither the functionality of the scheduler nor the scheduling policies considered here depend on how well a job performs on any of its valid processor partitions, we make the practical assumption that across the range of valid processor partitions for each job, the execution time of the job decreases monotonically (but not necessarily at the same rate) as the partition size increases. Thus, allocating a larger valid processor partition results in a reduction in the execution time of an application. We do not make any assumptions regarding the elements of the vpp set for each job except that all partition sizes are positive and less than or equal to the maximum number of processors in the system. Note also that the vpp sets of two jobs may be completely independent.

In addition to allocating processors to new jobs, the dynamic processor allocation policy considered must decide on dynamic allocation/deallocation of processors to/from running jobs. We describe this policy in more detail in Section 3.3. Here we point out some of the requirements that such a policy imposes on the parallel applications. Under this scheme, a job is always allocated a valid set of processors, but during its lifetime, the job may execute on more than one vpp. Whenever there is a change in the number of processors allocated to a job, it is assumed that the application and the associated runtime system reconfigure the job so it can continue to execute, from that point on, on the new set of processors. (For two approaches toward developing reconfigurable applications, we refer the interested reader to the DRMS [20, 19] and Octopus [9, 11, 10] programming environments.) The exact epoch when a job is reconfigured is left up to the job. However, it is expected that a job responds to the scheduler's reconfiguration request quickly relative to its total execution time. In our analysis we take into account the delay in responding to the scheduler's request, as well as in the actual cost of reconfiguring a job.

In summary, our assumptions on the parallel system architecture, the parallel applications and the workload are fairly general. To focus our study on realistic situations, the characteristics of the applications used in this study (execution time on various processor configurations and memory requirements) are based on a class of real-life parallel applications, the details of which are provided in Section 4. Since the policies we analyze and our methodology are independent of the application domain, we note here that our results are applicable to other class of parallel applications as well.

3 Scheduling Policies

We now define the three space-sharing policies considered in our study, which represent extensions to the standard static, adaptive and dynamic partitioning strategies. In all three cases, arriving jobs are classified based on their resource requirements as being either *small*, *medium* or *large* (defined more precisely in Section 4), and we use the subscripts l, m and s to indicate parameters or measures pertaining to the large, medium and small job classes, respectively. The static and adaptive policies also divide the total number of system processors, P, into three pools from which processors are primarily allocated for jobs from a particular class. We use P_l, P_m and P_s, $P = P_l + P_m + P_s$, to represent the number of processors designated primarily for allocation to large, medium and small jobs, respectively. As jobs move through the system, the available number of processors in each pool vary and we denote by $avail_l$, $avail_m$ and $avail_s$, the number of processors available for allocation from the respective pool; $avail_{sys}$ denotes the total number of processors available for allocation at any given time. For all three policies, a job is always guaranteed a minimum number of processors that is class dependent; we use min_l, min_m and min_s to denote the corresponding minimum number of processors.

3.1 Multi-class Fixed Partitioning (MFP)

Under MFP, a fixed fraction of the system processors is designated as "belonging to" job class k, $k \in \{s, m, l\}$, which we refer to as the class k processor pool. Within each pool, the processors are divided into a fixed number of disjoint partitions of equal size. Once the system has been partitioned in this manner, the configuration is fixed until the system is explicitly reconfigured. We use the notation MFP($n_s \times pe_s, n_m \times pe_m, n_l \times pe_l$) to denote a system with n_k partitions each of size pe_k, $k \in \{s, m, l\}$, and thus there are three processor pools and $n_s + n_m + n_l$ processor partitions. As a specific example, the MFP($4 \times 32, 1 \times 128, 1 \times 256$) policy configures a 512-processor system into 4 small-job partitions each of size 32, 1 medium-job partition of size 128, and 1 large-job partition of size 256. Although each partition is designated to belong to either the small, medium or large job class, a partition may execute jobs from a different class under certain circumstances, as defined below.

A first-come first-served (FCFS) queue is maintained for each job class. A job in such a queue has nonpreemptive priority over jobs from the other two classes with respect to a processor partition belonging to its class. Moreover, larger job classes cannot acquire partitions belonging to smaller job classes, while small jobs have nonpreemptive priority over medium jobs for large job partitions. In general, this policy favors jobs with smaller execution times in its decisions. This is motivated in part by the fact that the smallest-job-first policy (and its derivatives) minimizes mean job response time on uniprocessor systems [13], as well as by the characteristics of the workloads considered in our study.

Upon the arrival of a large job, the system allocates it a partition from the large job pool if one is available. Otherwise, the arriving job waits in the FCFS queue associated with the large job class. When the execution of a large job completes, the available partition is allocated to the job at the head of the large job queue. If this queue is empty and the system contains waiting small jobs, the partition is allocated to the job at the head of the small job queue. When this queue is also empty, the partition is allocated to the job at the head of the medium job queue, if any.

Similarly, when a medium job arrives to the system and one or more partitions are available in the medium- or large-job pools, the system allocates one of these partitions to the job; if partitions are available in both pools, the job is allocated a partition from the large-job pool. Otherwise, the job waits in a queue associated with the medium job class. Upon the departure of a medium job from a medium job partition, the available partition is allocated to the job at the head of the medium job queue. If this queue is empty and the system contains waiting small jobs, the partition is allocated to the job at the head of the small job queue.

A small job arrival is allocated any of the available partitions in the system, including those belonging to medium- and large-job pools; when there are large or medium job partitions available in addition to small job partitions, the job is allocated one of the large or medium job partitions. If no partition is available, the job waits in a queue associated with the small jobs. When the execution of a small job on a partition belonging to the small-job pool completes, that partition is allocated to the job at the head of the small job queue. If this queue is empty, the partition remains available until a small job arrives.

3.2 Multi-class Adaptive Partitioning (MAP)

This policy resembles the MFP policy in that a pool of processors is created for each job class at system configuration time, and the number of processors in each pool is fixed until the system is reconfigured. However, it differs from the MFP policy in that the processors comprising a pool are adaptively divided into partitions based on system load, as reflected in the per-class queue lengths. We use MAP(min_s–P_s, min_m–P_m, min_l–P_l) to denote a policy where the size of the processor partition allocated to a class k job is within the range min_k to P_k.

An FCFS queue is maintained for waiting jobs of each class. As under MFP, jobs from a particular class have non-preemptive priority over jobs of other classes for partitions belonging to the class such that large jobs cannot acquire partitions belonging to other classes, medium jobs cannot acquire small job partitions, and small jobs can execute on partitions belonging to any class.

The primary goal of this policy is to allocate relatively small partitions to jobs under heavy system loads and relatively large partitions during conditions of light load. In order to achieve this goal, the scheduler uses a "split-and-merge" strategy to adaptively set the size of a partition allocated to an incoming job. This strategy is a multi-class version of the AP4 strategy proposed in [28], under which the scheduler maintains a target partition size for each job class. The target partition size is recomputed upon job arrivals and departures. This enables the scheduler to adapt to changes in system conditions, such as the load and the number of available processors.

A pseudo-code fragment of the MAP policy is shown in Figure 1, which is executed whenever a small job arrives or departs the system. The compute_target procedure shown in Figure 1 describes the algorithm used for computing the target partition size. The variable $\mathtt{t}_k$ denotes the target partition size for class k, and $\mathtt{Qlen}_k$ denotes the number of jobs of class k in the queue, $k \in \{s, m, l\}$. When the length of a queue increases, the target partition size for that class is decreased (i.e., the split strategy) subject to the constraint that the partition must have at least min_k processors. When the queue length decreases and the number of available processors in the corresponding pool becomes greater than twice the current target partition size, the target partition size for that class is increased (i.e., the merge strategy). Intuitively, the split strategy makes the target partition size small enough so that all waiting jobs in the corresponding queue can be allocated a partition. The merge strategy, on the other hand, eliminates one of the unallocated partitions by distributing the processors of that partition evenly among the remaining partitions, and this determines the new target partition size. A merge does not affect the size of previously allocated partitions, but there is at least one unallocated partition whose size increases to the new target size. Since the target partition size determined by the split strategy is bounded below by min_k, the system can prevent arbitrarily small partitions which can result in poor performance for some workloads, as we will show in Section 6. Thus, min_s, min_m and min_l are policy parameters that can be tuned based on a knowledge of the current and expected workload.

We note that, under MAP, medium and large jobs can only be dispatched on a partition "belonging to" their class when the number of available processors of that class is at least as large as the current target size for that class. If there is a medium (large) job waiting in the medium (large) job queue, but $avail_m < t_m$ ($avail_l < t_l$), then the available processors are kept idle. These processors remain idle until either: (i) the target size is recomputed and becomes less than or equal to the number of available processors, or (ii) enough processors "belonging to" that class are released by a departing job so that the number of available processors is at least as large as the target size for that class. In the case of small jobs, however, if there is a job waiting in the small job queue and $avail_s > \max\{min_s, t_s/2\}$, these processors are allocated to that job.

Finally, we note that when there are no large (medium) jobs waiting for processors, MAP will schedule any waiting small job on an available large (medium) partition if the size of the allocated partition is at least as large as $\max\{min_s, t_s/2\}$. Similarly, if there are no large jobs waiting for processors, any waiting medium job can be scheduled on an available large partition if the size of the allocated partition is at least as large as $\max\{min_m, t_m/2\}$. The motivation behind these rules is to keep the system processors as busy as possible.

3.3 Dynamic Partitioning (DP)

DP differs from MAP and MFP in two important ways. First, DP does not divide the system processors *a priori* into partition pools "belonging to" each job class. Second, the partition size of a large or medium job can be dynamically adjusted during its execution by suitably reconfiguring the application. Small jobs are not dynamically reconfigured because of overhead considerations [35].

The operation of the DP policy is controlled via the seven parameters shown in Table I. The system maintains a single FCFS queue for waiting jobs, and a list of running jobs that are eligible for reconfiguration, which we call the *interruptible list*. After a medium or large job reconfigures itself, the job is removed from this list and is not eligible for reconfiguration until a certain time interval, denoted by T, has elapsed. This mechanism is motivated in part by the need to prevent a form of reconfiguration thrashing [35].

When there are no other constraints, the DP policy allocates a valid processor partition to a new job such that the size of this partition is close to the average size of all currently allocated partitions in the system (including this new partition). Partition reconfigurations are also guided by the current average partition size. Thus, DP tries to allocate P/N processors to each job in the system, where N is the number of jobs in the system (including both waiting and executing jobs), while making sure that the allocated partition is in the vpp set for that job. Small jobs are thus scheduled in an adaptive manner, since they are not reconfigured.

Let t_{appl} be the target partition size which is given by $\min\{req_{appl}, \max\{min_k, P/N\}\}$, $k \in \{s,\ m,\ l\}$, where req_{appl} is the preferred partition size as requested by the job. When the number of available processors $avail_{sys}$ found by an arriving job is greater than t_{appl}, then the job is allocated a valid processor partition that is closest to t_{appl}. However, if $avail_{sys} < t_{appl}$, and there are jobs on the interruptible list, the system scheduler notifies one or more of these jobs to shrink down to a valid processor partition that is closer to P/N, provided that: (i) $P/N \geq min_k$, (ii) the sum of $avail_{sys}$ and the number of processors that will be released by the interrupted jobs is at least t_{appl}. To minimize the number of interrupted jobs, the system attempts to release just enough processors so that the total number of available processors is close to t_{appl}.

When a job departs, the target allocation t_{appl} for each waiting job is recomputed by taking into account the processors released by the job. The job queue is examined in a "first fit" fashion for jobs whose t_{appl} is less than the number of available processors. This process is repeated until there are no jobs in the queue or the number of available processors falls below the value of t_{appl} for all waiting jobs.

If the number of available processors exceeds a policy parameter M and the interruptible list contains a job whose current partition size is less than P/N, then the system scheduler notifies that job to increase to a larger partition size from its vpp set that is closest to $\min\{P/N, Z + f \cdot avail_{sys}, req_{appl}\}$, where Z denotes the job's current partition size. Thus, at most a fraction f of the available processors are allocated at a time to one of the large or medium jobs on the interruptible list. Thus, the policy parameters M and T can be used to control the rate at which large and medium jobs are reconfigured.

Finally, the system scheduler must prevent the situation where no small job can be allocated processors because *all* processors in the system are allocated to large or medium jobs and *each* of the running jobs is allocated its minimum allocation. Since DP never reduces the processor allocation of an executing job below its minimum requirement, under these conditions, small jobs will be forced to wait until one of the large or medium jobs finishes, thus leading to poor performance for small jobs. To prevent this type of starvation, the system scheduler uses a policy parameter, denoted by L, to limit the maximum number of large and medium jobs that can be executing concurrently. This policy parameter essentially defines a maximum multiprogramming level for the medium and large job classes. Given the minimum processor requirements of the medium and large job classes, and the number of processors in the system (P), an appropriate value for L can be computed. The system scheduler keeps track of the number of executing large and medium jobs and ensures that this number never exceeds L.

Dynamic Application Reconfiguration: Under the DP policy, large and medium applications have to be able to reconfigure themselves in response to requests from the scheduler. We assume that when a job is notified by the scheduler to modify its partition size, the application initiates a reconfiguration at the next programmer-defined reconfiguration point. At the reconfiguration point, both the distributed data and control structures within the application are reorganized so that the application can continue to execute on its new set of processors. To simplify the analysis, we assume that the reconfiguration takes place in two phases. In the first phase, the entire data set representing the current state is transferred from all processors in the old partition to min_k processors. In the second phase, the data set is then redistributed from this subset to all processors in the new partition. This two-phase scheme can be used for both increasing and decreasing the partition size. Before commencing the second phase, processors are either added or removed from the partition depending on whether the partition size is to be increased or decreased.

4 Application Workload

4.1 Job characteristics

To study job characteristics that are representative of actual large-scale computing environments, we conducted a detailed analysis of several large computational fluid dynamics (CFD) applications. The parallel applications used in our study are some of the NAS parallel benchmarks [2] and parallel versions of ARC3D [27] and INS3D-LU [41], all of which were originally developed at the NASA Ames Research Center. While these applications represent only one class of applications, they sufficiently capture the behavior of job streams executed at typical large-scale computing centers.

We selected a total of 23 different applications with grid sizes ranging from 4K through 100M and execution times ranging from a few seconds to several days. For each application, we determined the execution times for each of the processor configurations in its vpp set. These times were obtained by a detailed measurement-based analysis of each application, which were validated against measurements of an implementation of the applications on several parallel computing platforms at the IBM T. J. Watson Research Center. Some instances of the applications considered in our study could not be executed on these systems due to memory and time constraints, and in these cases the resource requirements were determined by analysis and simulation. For further details on the parallelization techniques and the relevant analysis, we refer the interested reader to [21, 22] and the references therein. The execution times for the 23 applications on 32, 64, 128, 256 and 512 processors are given in our corresponding technical report [24].

In the case of DP, an additional factor that needs to be taken into account is the overhead for reconfiguration. This overhead depends on the characteristics of the application being considered as well as the underlying architecture. We computed the overhead for the 8 largest applications in our study assuming that reconfiguration is realized in the two-phase scheme described in Section 3.3. In Table II, we report the corresponding times for a single phase of this scheme. The per-phase reconfiguration cost reported in Table II is the cost of reconfiguring an application from its smallest valid processor partition to its largest valid processor partition.

4.2 Workload characteristics

We consider workloads consisting of a mixture of the 23 applications, which have large variations in resource requirements. Given this large variability in resource demands, the system can coarsely classify jobs as being either *small*, *medium* or *large*. Specifically, we assign the first 15 applications to the small job class, the next 4 applications to the medium job class, and the last 4 applications to the large job class. The workload is then modeled by probabilistically determining the class of a job upon its arrival, i.e., an arrival belongs to the small, medium or large class with probability p_s, p_m, and p_l, respectively, $p_s + p_m + p_l = 1$. We assume that the arrivals to the system are from a Poisson source with mean rate λ. Thus, the arrivals of small, medium and large jobs form three independent Poisson arrival streams with average rates $p_s\lambda$, $p_m\lambda$, and $p_l\lambda$, respectively. The application executed by each class k job arrival is uniformly chosen from among the set of applications in class k. We present in Table III the minimum, maximum and average execution times for each job class when executed on the number of processors indicated in the third column of the table.

We consider four different workloads in our study with the probabilities p_s, p_m and p_l chosen to maintain various ratios of small to medium to large jobs. These workloads are described in Table IV, which includes the coefficient of variation of the workload service times on 32 and 512 processors. Workloads W1 and W2 are typical of those found at large-scale scientific computing centers, where the majority of jobs arriving to the system are small jobs, but most processor cycles are consumed by a few long running applications [7, 38]. Workload W3 corresponds to the extreme situation where the job stream primarily consists of jobs with relatively small service requirements to the extent that the majority of processor cycles are consumed by small jobs. Finally, workload W4 corresponds to the case where the percentage of small jobs submitted to the system is of the same order of magnitude as that for the medium and large jobs combined. The relative processor cycles consumed by the jobs of the three classes are shown in Table IV under the column with heading $c_l : c_m : c_s$, where c_k represents the number of processor cycles consumed by jobs of class k relative to the number of processor cycles consumed by small jobs. Note that this represents a wide spectrum of application workload characteristics which allows us to study the impact of these characteristics on system performance under the scheduling policies of Section 3.

5 Performance Evaluation

5.1 Performance metrics

Our analysis of the scheduling policies in Section 3 is based on metrics that measure performance from two different perspectives, namely from the system and user viewpoints. The first performance metric, $\mathrm{E}[\,S\,]$, is the mean job response time. Let J_n denote the n^{th} job to leave the system, $n \geq 1$, and define α_n and δ_n to be the arrival and departure times of J_n, respectively. We then have

$$\mathrm{E}[\,S\,] \;\equiv\; \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \left(\delta_n - \alpha_n\right), \qquad \delta_n > \alpha_n. \tag{1}$$

The variability of response times is another important performance measure in computer systems. Thus, we also consider the coefficient of variation* of job response times, which is given by

$$\mathrm{CV}[\,S\,] \;\equiv\; \frac{\sqrt{\mathrm{E}[\,S^2\,] \,-\, \mathrm{E}[\,S\,]^2}}{\mathrm{E}[\,S\,]} \tag{2}$$

where

$$\mathrm{E}[\,S^2\,] \;\equiv\; \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \,(\,\delta_n \,-\, \alpha_n\,)^2. \tag{3}$$

We analogously define per-class versions of these response time measures, namely $\mathrm{E}[\,S_k\,]$ and $\mathrm{CV}[\,S_k\,]$, $k \in \{s, m, l\}$. Note that the corresponding throughput measures are directly related to $\mathrm{E}[\,S\,]$ and $\mathrm{E}[\,S_k\,]$ by Little's result [13] and, given the above modeling assumptions, are equal to the job arrival rates λ and $p_k\lambda$, respectively. This set of measures address performance from the *system* perspective.

Our second set of performance metrics is based on a response time to service time ratio. In particular, we define the mean ratio of job response time to job service time, $\mathrm{E}[\,U\,]$, as

$$\mathrm{E}[\,U\,] \;\equiv\; \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \frac{\delta_n \,-\, \alpha_n}{x_n} \tag{4}$$

where x_n denotes the service time of J_n on req_{appl} processors. The corresponding coefficient of variation is given by

$$\mathrm{CV}[\,U\,] \;\equiv\; \frac{\sqrt{\mathrm{E}[\,U^2\,] \,-\, \mathrm{E}[\,U\,]^2}}{\mathrm{E}[\,U\,]} \tag{5}$$

where

$$\mathrm{E}[\,U^2\,] \;\equiv\; \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \frac{(\,\delta_n \,-\, \alpha_n\,)^2}{x_n^2}. \tag{6}$$

Per-class versions of these response time to service time ratio measures are analogously defined, i.e., $\mathrm{E}[\,U_k\,]$ and $\mathrm{CV}[\,U_k\,]$, $k \in \{s, m, l\}$. Note that in each case queueing delays and not being allocated req_{appl} processors both result in differences between the values of $(\delta_n - \alpha_n)$ and x_n. This set of measures reflect the performance perceived by the *user*.

In our experiments we obtain the above system-level and user-level performance measures as functions of the system utilization, which we define here based on the average computational demand of these jobs when executed on 1 processor. Specifically, the average computational demand of a generic job arrival on a processor, denoted by $D(1)$, is equal to $p_s D_s(1) + p_m D_m(1) + p_l D_l(1)$, where $D_k(1)$ denotes the average demand of jobs belonging to class k when executed on 1 processor. The utilization is then equal to $\lambda/(P \cdot D(1))$.

5.2 Simulation Methodology

The performance measures described above are obtained from a stochastic discrete-event simulation written using CSIM, a process-oriented simulation library [29]. In our simulation experiments, the

*The ratio of the standard deviation to the mean.

class of an arriving job is determined based on the input parameters p_s, p_m and p_l (see Section 4), and its execution time is determined based on the application and on the number of processors allocated to it by the scheduler. In the case of the DP policy, the simulator keeps track of the number of iterations executed by each job in the system. Jobs can only be reconfigured at the end of an iteration. On job reconfiguration, the remaining execution time is determined based on the number of remaining iterations and the new partition size of the job.

A large number of simulation experiments were performed under a variety of workload conditions. In each case, the mean response time is obtained using the regenerative method [15] to within 5% of the mean at 95% confidence intervals. The number of job arrivals that had to be simulated to attain these confidence levels ranged from several hundred thousand to several million, depending upon the load on the system.

6 Results

Our main objective in this section is to determine a set of principles for the scheduling of large-scale scientific applications in distributed-memory parallel computing environments. The primary performance measures of interest are the mean job response time (i.e., $\mathrm{E}[\,S\,]$) and the mean ratio of job response time to job service time (i.e., $\mathrm{E}[\,U\,]$), as well as per-class versions of these measures. Our results, a portion of which follow, were obtained as described in Section 5. Unless otherwise noted, the results presented herein are for the parameter values $min_s = 16$, $min_m = 32$ and $min_l = 32$, allocated in units of 8 processors.

6.1 System-Level Performance

MAP vs MFP

We first consider system performance under the MFP and MAP policies. In Figure 2, we plot the average response times under three MFP policies and two MAP policies as a function of the system utilization for workload W1. The three MFP policies are MFP(1×64, 1×224, 1×224), MFP(2×32, 2×112, 2×112) and MFP(2×32, 4×56, 4×56), whereas the two MAP policies are MAP(32-64,56-224,56-224) and MAP(32-64,112-224,112-224). These policies were chosen to illustrate the trends observed in our numerous simulation experiments.

As expected, we observe that MFP(2×32, 2×112, 2×112) yields better system performance than MFP($1 \times 64, 1 \times 224, 1 \times 224$) at high utilizations, but the opposite is true at light to moderate loads. The overall performance of MFP(2×32, 4×56, 4×56) is always worse than that of the other MFP policies, although small job performance under MFP(2×32, 4×56, 4×56) is better than that of MFP($1 \times 64, 1 \times 224, 1 \times 224$) at moderate to heavy loads. These observations suggest that while MFP policies assigning small partitions generally perform better at higher utilizations than policies assigning large partitions, there is a limit to the performance benefits of reducing partition sizes at heavy loads. This limit depends on the speedup and efficiency characteristics of the jobs comprising the workload, and thus exploiting knowledge of the expected workload can be important for formulating a policy that performs well.

We also observe that the two MAP policies perform as well as the best MFP policy at light to moderate loads. At high utilizations, MAP(32-64,112-224,112-224) performs as well as the best MFP policy, but the system performance under MAP(32-64,56-224,56-224) starts to degrade. This is because the latter policy tends to allocate partitions similar to MFP(2×32, 4×56, 4×56) at heavy loads, which was previously shown to provide poor performance. Hence, the MAP policy parameter min_k, which determines the minimum partition size for processors "belonging to" class k, has to be selected with care. We also note that these results are similar to those previously reported for the corresponding single class versions of MAP and MFP [23, 32].

A knowledge of the expected workload – specifically, of the ratios $p_l : p_m : p_s$ and $c_l : c_m : c_s$ (see Table IV) – is also important for determining the best inter-class partitioning of the system processors. To illustrate this, we provide in Table V the processor partitioning that yielded the best system performance for each workload in our numerous simulation experiments with different inter-class partitions. The table clearly shows that the best partitioning varies widely with workload.

In summary, we conclude that a MAP policy offers more flexibility, and is therefore preferable to MFP policies. However, system performance under both MAP and MFP policies is dependent upon some knowledge of the workload characteristics.

MAP vs DP

Figure 2 also illustrates the system performance under the DP policy for workload W1. We observe that DP outperforms the other policies at all loads, and that the performance differences are quite significant. Focusing on the per-class response time curves, we see that DP outperforms the best MAP policy with respect to both small jobs and medium jobs across all loads, as well as for large jobs at light to moderate loads. One reason for this is that the largest number of processors that can be allocated to a job under MAP is determined by the inter-class partitioning, and can never be larger than $\max\{P_s, P_m, P_l\}$, whereas a job can receive as many as P processors under DP. At low utilizations, since there is no contention for processors, most jobs are allocated as many processors as they have requested under DP. Under MAP, however, no job is allocated more than $\max\{P_s, P_m, P_l\}$ processors even at light loads. This is because, once a job is initiated, it runs to completion without being interrupted. Allocating all of the system processors to a long-running job will result in poor performance for any jobs that arrive while the long-running job is being executed. Previous work [23, 4] has shown that for workloads consisting of jobs with widely varying execution times, it is important for adaptive policies to restrict the maximum number of processors allocated to individual jobs to reduce/eliminate such situations. Under DP, however, the execution of large and medium jobs can be interrupted and some of their processors can be reallocated to arriving small jobs. Furthermore, idle processors can be reallocated to any large and medium jobs being executed. The results in Figure 2 show that the overhead of reconfiguring such applications under DP, when done appropriately, can be outweighed by the performance benefits of doing so.

For large jobs, however, system performance under DP at heavy loads is worse than that of the best MAP policy. This is because at high utilizations the MAP policy tends to favor (in a relative sense) large jobs over other job classes; this observation is further supported by the results presented in Figure 4. At heavy loads, under the MAP policies shown in Figure 2, the execution of small jobs is restricted to the 64 processors reserved for this class and the medium jobs are forced to execute on the 224 processors assigned to them, whereas 224 processors are always used to execute large jobs. This results in poor performance for small and medium jobs since the per-class policy allocations are close to the saturation point for these job classes, but this yields acceptable system performance for the large jobs where the policy is not saturated. (Recall that for this workload, the arrival rates of small, medium and large jobs are in the ratio 100:10:1.) On the other hand, the DP policy at high utilization tends to favor (to some extent) small jobs over medium and large jobs (see Figure 4). The number of processors allocated to individual jobs is reduced under DP as the total number of jobs in the system increases, although this can be controlled with the various policy parameters (see Section 6.3). As a result, large jobs are allocated fewer processors under DP than is the case under the MAP policy, which leads to the trends illustrated in Figure 2. We note, however, that even though individual medium jobs are allocated fewer processors at heavy loads under DP than under MAP, the medium job class has better performance under DP. This is because at high utilizations, the medium job class receives a larger fraction of the system processors under DP than it does under MAP.

In Table VI we provide reconfiguration statistics for large and medium jobs obtained from our simulations of the DP policy and we quantify the overhead as a fraction of the total response time. Note that the increase in the average response time for medium and large jobs at heavy loads is because of the decrease in the average partition size for these classes, and not due to the reconfiguration overhead. As indicated in the table, the reconfiguration overhead, while not insignificant, ranges from 1–4% of the response time for medium jobs and from 4–11% of the response time for large jobs, and as such, represents a relatively small fraction of the response time for these job classes.

We now compare system performance under the DP and MAP policies for each of the 4 workloads summarized in Table IV to see if the trends observed above hold true across this spectrum of workloads. Let $\mathcal{T}_S \equiv \mathrm{E}[\,S_{DP}\,]/\mathrm{E}[\,S^*_{MAP}\,]$, where $\mathrm{E}[\,S^*_{MAP}\,]$ denotes the smallest mean response time obtained under each of the MAP policies considered; similarly, we define $\mathcal{T}_{S,k} \equiv \mathrm{E}[\,S_{DP,k}\,]/\mathrm{E}\left[\,S^*_{MAP,k}\,\right]$, $k \in \{s, m, l\}$. In Figure 3, we plot $\mathcal{T}_S$ as a function of the system load for each workload, thus illustrating the relative response time of DP with respect to the best MAP policy. Note that when the ratio is less than 1, DP outperforms MAP.

We observe from Figure 3 that when all jobs are considered together, DP outperforms the best MAP policy at all utilizations for all of the workloads. Similarly, DP generally provides lower mean response times for the small and medium job classes than the best MAP policy, with the only real exception appearing for medium jobs under W3 at heavy loads. The DP policy, however, yields worse performance for large jobs than MAP at high utilizations. Thus, the trends described above for workload W1 generally hold true across the wide spectrum of workloads considered in our study.

Figure 3 also quantifies the system performance benefits of the DP policy with respect to MAP. Since we are plotting the *ratio* of the response time of DP to the response time of the MAP policy that provided the best system performance from among all of the MAP policies considered, the plots of $\mathcal{T}_S$ in Figure 3 are influenced by the behavior of both policies. In particular, if the MAP policy has a particularly poor performance characteristic at a certain system load, this results in a small value for $\mathcal{T}_S$.

We observe from Figure 3 that the plots for workload W4 are quite similar to those for workload W1, whereas the plots for workloads W2 and W3 exhibit some differences. We now examine in more detail these results for workloads W2 and W3.

For workload W2, we observe that the value of $\mathcal{T}_S$ taken over all job classes decreases sharply at high utilizations in comparison to the other workloads, i.e., the overall system performance under DP increases relative to the MAP policy being considered – namely, MAP(32-128,64-128,128-256) – as the load increases for workload W2. Moreover, the value of $\mathcal{T}_{S,s}$ at high utilizations is smaller than the corresponding value for the other workloads, whereas the the value of $\mathcal{T}_{S,l}$ at moderate to heavy loads is much larger than the corresponding value for the other workloads. To understand the reasons for these performance characteristics, we need to consider the behavior of the MAP(32-128,64-128,128-256) policy at high utilizations. Essentially, the inter-class partitioning employed by this policy is such that it tends to favor large jobs over jobs of other classes at heavy loads, which results in a large relative response time for large jobs. At the same time, the policy effectively restricts small jobs to 128 processors at high utilizations which leads to saturation and poor performance for small jobs. This results in small values of $\mathcal{T}_{S,s}$ and large values of $\mathcal{T}_{S,l}$ at heavy loads.

For workload W3, we observe that the performance benefits of DP with respect to MAP are smaller than those for the other workloads. In this case, the overwhelming majority of job arrivals belong to the small job class and the MAP policy under consideration – namely, MAP(64-448,0-0,64-64) – reserves most of the processors in the system for these small jobs. We also observe that the values of $\mathcal{T}_{S,m}$ and $\mathcal{T}_{S,l}$ at low utilizations is significantly smaller than the corresponding value for other workloads. This is once again an artifact of the MAP policy under consideration, wherein a medium or large job will be allocated at most 64 processors even at light loads; conversely, the DP policy will tend to allocate partitions that are much closer to 512 processors at low utilizations. On the other hand, the value of $\mathcal{T}_{S,m}$ at heavy loads is significantly larger than the corresponding value for other workloads. This is due in part to the particular parameter setting for the DP policy being considered, namely $L = 1$. (Recall that parameter L determines the maximum number of large and medium jobs that can be executing in the system at any given time.) At high utilizations, the FCFS queue discipline leads to long queueing delays and poor performance for medium jobs.

In summary, the results in this section show that DP can yield significant system performance benefits over MAP and MFP. They also illustrate how a particular inter-class partitioning for MAP may not perform well over all system loads, and that this is even worse under the MFP policy. One of the advantages of DP is that it is not necessary to partition the processors *a priori* among the various job classes. However, the various DP policy parameters, such as L, can have a significant impact on the relative system performance of the different job classes. We examine the impact of varying these parameters in Section 6.3.

6.2 User-Level Performance

We now examine the performance of MAP and DP from the user viewpoint. Recall that $\mathrm{E}[U]$ is defined as the average ratio of the response time to the execution time on req_{appl} processors (see Eq. (4)), and as such quantifies the application's performance as perceived by a user. Note that both queueing delays experienced by the application and any increase in execution time because of being allocated fewer than req_{appl} processors are reflected in $\mathrm{E}[U] > 1$. Furthermore, since different classes of jobs will have different expectations of performance, we consider the average response ratio for each class separately.

In Figure 4, we plot $\mathrm{E}[U]$ for each job class as a function of system utilization under MAP(32-64,112-224,112-224) and DP for workload W1. We observe that both policies are relatively "fair" at light to moderate loads; i.e., the $\mathrm{E}[U]$ values are relatively close to 1. At high utilizations, however, DP tends to favor small jobs over large and medium jobs, but only to a relatively small extent. The MAP policy, on the other hand, favors large jobs at the expense of the performance of small jobs, and the performance differences are quite significant. Since the overwhelming majority of the jobs being submitted to the system are small jobs, this is clearly undesirable. Figure 4 also shows that DP has better performance than MAP from the user viewpoint under workload W1, with the only exception being in the case of large jobs at heavy loads.

In Figure 5, we plot the corresponding coefficient of variation $\mathrm{CV}[U]$ (see Eq. (5)) for the different job classes under MAP and DP for the workload W1. As we are considering performance from the user viewpoint, a low coefficient of variation is desirable. We observe that the value of $\mathrm{CV}[U_s]$ under both MAP and DP is much higher than that for the other classes of jobs. This is as expected since the coefficient of variation of the execution times for small jobs (see Table IV) is much larger than the coefficient of variation of the execution times for the other job classes. The value of $\mathrm{CV}[U_s]$ under MAP is much larger than its value under DP at low to moderate utilizations. This is because there is a wide variation in the number of processors allocated to small jobs under MAP, where a small job may be allocated a partition designated for the small, medium or large job classes. On the other hand, the number of processors allocated to a job under DP depends only on the number of jobs currently running and waiting in the system, since DP tries to allocate an equal number of processors to each running job. As the load increases, the MAP policy increasingly constrains small jobs to execute only on partitions "belonging to" small jobs, which leads to a reduction in the value of $\mathrm{CV}[U_s]$ with increasing system load. Under DP, however, the value of $\mathrm{CV}[U_s]$ increases sharply at high utilizations, which occurs because DP reduces the average number of processors allocated to small jobs as the system load increases. Table III shows that within the class of small jobs, req_{appl} varies widely from 8 to 512. Thus, while decreasing the partition size for small jobs with small req_{appl} does not have much impact on the response ratio for those jobs, it has a significant impact on the response ratio for small jobs with large req_{appl}. This tends to increase the value of $\mathrm{CV}[\,U_s\,]$.

Let $\mathcal{T}_U \equiv \mathrm{E}[\,U_{DP}\,]/\mathrm{E}[\,U^*_{MAP}\,]$, where $\mathrm{E}[\,U^*_{MAP}\,]$ denotes the smallest mean ratio of response time to service time obtained under each of the MAP policies considered; similarly, we define $\mathcal{T}_{U,k} \equiv \mathrm{E}[\,U_{DP,k}\,]/\mathrm{E}\left[\,U^*_{MAP,k}\,\right]$, $k \in \{s, m, l\}$. In Figure 6, we plot $\mathcal{T}_U$ for each job class under all 4 workloads. We observe that the trends observed in the case of workload W1 generally hold true for the other workloads. That is, DP yields lower $\mathrm{E}[\,U_s\,]$ and $\mathrm{E}[\,U_m\,]$ values than MAP at all loads, but DP provides a larger $\mathrm{E}[\,U_l\,]$ than MAP at high utilizations. The exceptions to these trends observed in Figure 6 can be explained by taking into account the specific MAP and DP policies under consideration. For example, the large response ratio at heavy loads for medium jobs under workload W3 is due in part to the particular parameter setting for the DP policy where $L = 1$.

It is interesting to compare the $\mathcal{T}_U$ curves in Figure 6 with the $\mathcal{T}_S$ curves in Figure 3. We observe that the value of $\mathcal{T}_{U,s}$ is closer to 1 than the corresponding value of $\mathcal{T}_{S,s}$ at low utilizations, whereas the opposite is true at heavy loads. This implies that at low utilizations, the performance of MAP from the user viewpoint is comparable to that of DP, even though the absolute performance of DP is better. In fact, for workload W2, MAP yields somewhat better user-level performance than DP at light loads, even though the absolute mean response time is lower under DP. Conversely, at high utilizations, the performance benefit of using DP instead of MAP is much higher from the user viewpoint (55% – 90%) than from the system perspective (5% – 80%). In the case of medium jobs, the performance benefit of DP over MAP is essentially the same from both system and user perspectives at light loads, but it is larger from the user viewpoint at high utilizations. In the case of large jobs, the $\mathcal{T}_{U,l}$ and $\mathcal{T}_{S,l}$ curves tend to be similar.

6.3 Performance Impact of DP Parameters

The results presented in the previous two sections have shown that DP yields better performance than MAP from both the user and system perspectives. We now analyze the performance characteristics of DP in more detail. In particular, several simulation experiments were conducted to examine the impact of varying the policy parameters described in Table I. Here we briefly describe the salient points of these experiments to illustrate the observed trends. In Figure 7, we plot $\mathrm{E}[\,S\,]$ under DP with different values of the parameters L, M and T for workload W1.

The parameters T and M can be used to control the rate at which medium and large jobs are reconfigured. Increasing T makes jobs "uninterruptible" for larger intervals of time, whereas increasing M reduces the rate at which for large and medium jobs can acquire free processors. Figure 7 shows that for workload W1, increasing M to 128, while keeping the other parameters fixed, degrades performance at moderate to high utilizations. This is caused by the reduced probability of having more than 128 available processors as the system load increases, which decreases the ability of large and medium jobs to increase their partition size when there are idle processors in the system.

Although not shown in Figure 7, we also examined the impact of varying the policy parameter

f, which corresponds to the fraction of idle processors that can be acquired by an executing large or medium job. We found that $f = 1$ generally yielded the best performance, implying that reallocating all idle processors to an executing large job results in better performance than allocating a fraction of these processors to the large job, while "saving" the remaining fraction for future arrivals. This result, of course, depends quite heavily upon the job arrival process.

The impact of varying T, shown in Figure 7, suggests that $T = 0$ results in the best performance at light loads, but as the utilization increases $T = 3$, and then $T = 10$ result in the best performance. This is because as the load increases, there is a higher probability that a large job will be reconfigured more frequently, which results in a larger system overhead. It is therefore important to dampen the rate of large and medium job reconfigurations by increasing T as the system load rises to improve performance under DP.

We also observe in Figure 7 that the parameter L can have a significant impact on the performance of DP at high system utilizations. Recall that L is used to control the maximum number of large and medium jobs that can be executing simultaneously in order to ensure that there is never a situation in which all of the processors in the system are allocated to large or medium jobs, thus starving the small job class for a significant amount of time. Hence, an appropriate value for L has to be selected based on the minimum processor requirements for medium and large jobs, i.e., min_m and min_l, respectively. In the plots varying L, the DP policy has $min_m = min_l = 64$. This implies that if there are 8 or more large and medium jobs executing on a 512 processor system, then all arriving jobs will be forced to wait until one of the executing jobs completes. To prevent this from happening, L should be less than 8. As illustrated in Figure 7, we see that DP with $L = 16$ has poor performance at high utilizations for exactly this reason. On the other hand, reducing L to a small value can also degrade performance because it results in increased queueing delays for large and medium jobs. This is shown in Figure 7 where we see that DP with $L = 6$ also has poor performance at heavy loads. We observe that setting $L = 7$ balances the goals of improving small job performance (by preventing large and medium jobs from acquiring all of the processors in the system) and reducing queueing delays for large and medium jobs.

The results in the previous section have shown that DP yields better performance than MAP for the workloads and system model considered. A natural question that arises is: how sensitive is DP performance to the assumptions about the overhead for dynamic application reconfiguration? In other words, would DP still outperform MAP if the overhead for dynamic reconfiguration were higher? Conversely, if the overhead was lower, would MAP still outperform DP at high utilizations for large jobs?

To address these questions, we conducted simulation experiments for workload W1 assuming that the overhead to reconfigure each large and medium application (shown in Table II) is increased or decreased by a constant factor z. In Figure 8, we plot the ratio of the response time of DP assuming various values of this overhead factor (denoted by $DP_{(z)}$) to that of DP assuming the overhead given in Table II (denoted by $DP_{(1)}$). We also plot the ratio of the response time of MAP to that of $DP_{(1)}$.

We observe that increasing the overhead has the greatest impact on the relative response time at moderate loads, while reducing the overhead has the greatest impact at high utilizations. However, increasing the overhead by a factor of 4 results in a 40% increase in response time, while reducing the overhead by a factor of 4 results in a 20% decrease in response time. We also observe that increasing the overhead has a greater impact on the performance of small jobs than on jobs belonging to the other classes. It is interesting to compare the relative performance of DP with larger reconfiguration overhead values to that of MAP. Figure 8 shows that $DP_{(2)}$ outperforms MAP at all loads, while $DP_{(4)}$ and $DP_{(8)}$ only outperform MAP at low utilizations. In the case of small jobs, $DP_{(4)}$ outperforms MAP at almost all loads, while $DP_{(8)}$ does so at high utilizations. In the case of medium and large jobs, increasing the overhead results in a decrease in the cut-off utilization below which $DP_{(z)}$ outperforms MAP. Conversely, decreasing the overhead results in an increase in the cut-off utilization below which DP(z) outperforms MAP. In fact, in the case of $DP_{(0.25)}$, the DP policy outperforms MAP for all classes of jobs at almost all utilizations.

These results show that, for our workload and system models, the overhead of dynamic reconfiguration has to increase by a factor of 4 above those considered in previous sections before DP exhibits worse performance than MAP. This reinforces the conclusion that any performance degradation resulting from the overhead of dynamically reconfiguring long-running applications under DP is outweighed by the benefits of doing so, provided that the proper policy parameters are employed.

6.4 Improving DP Performance

We now consider a few ways in which DP performance can be further improved. First, we examine the performance benefits of damping the rate at which large and medium jobs can be reconfigured if the overhead of doing so is high. From Figure 8, we observe that the MAP policy under consideration outperforms $DP_{(4)}$ and $DP_{(8)}$. In Figure 9, we plot the ratio of response time under two damped versions of DP to that of MAP for the same workload and system models. Here the rate at which large and medium jobs can be reconfigured is reduced by increasing the policy parameter T from 3 to 6 and the parameter M from 32 to 64. We observe that damping hurts performance at light loads, while resulting in a small but noticeable benefit at moderate to high utilizations. This suggests that if applications in the system workload have high reconfiguration costs, DP should adjust its rate of reconfiguration according to the load on the system.

Another simple modification to DP that will improve performance is to maintain two separate overload queues for medium and large jobs, respectively. Recall that the DP scheduler maintains a single queue in which large and medium jobs are held if the total number of executing jobs from these classes exceeds the policy parameter L. For workload W3, at high utilizations, medium jobs performed poorly because a single queue was used. Keeping two queues and using a policy that gives priority to medium jobs but ensures (e.g., by aging) that large jobs are not starved, can avoid this problem.

Finally, we examine how the parameter min_k, which determines the minimum number of processors allocated to a job of class $k \in \{s, m, l\}$, can be used to improve the performance for class k. As a specific example, we consider here the relative performance of DP and MAP under workload W2. In Figure 10, we plot the ratio of the response time of DP to that of MAP for two different parameter settings. We observe that when $min_m = min_l = 32$, the performance of DP for large jobs relative to that under MAP is quite poor, even though the overall performance of DP is much better. However, increasing min_m and min_l to 64 results in a significant improvement in large and medium job performance, while resulting in a relatively small degradation in overall performance. Note that it is important to reduce L to 7, if min_m and min_l are increased. This example demonstrates how the min_k parameters can be used to achieve the desired per-class performance objectives, in addition to the parameters that control the rate of reconfigurations.

7 Related Work

A large number of research studies have examined multiprogrammed scheduling policies for parallel computers. Our focus in this paper has been on the class of space-sharing scheduling strategies. In contrast to space-sharing, the class of *time-sharing* policies share the processors by rotating them among a set of jobs in time. Another important class of scheduling strategies, called *gang scheduling*, consists of various combinations of sharing in both time and space. Several approaches have been considered in each of these scheduling classes, and we refer the interested reader to the Octopus system [11, 10] and to [37, 40] and the references therein for details on related performance tradeoffs and design considerations. We note, however, that our results are relevant for the space-sharing component of any gang scheduling strategy.

Within the class of space-sharing strategies, several static partitioning policies have been proposed and analyzed; see [33, 8, 23, 32, 4, 28] and the references therein. This approach, however, can lead to relatively low system throughputs and resource utilizations under nonuniform workloads, as these and other studies show. Some research studies have shown how different *a priori* information about program characteristics can be exploited to make scheduling decisions (e.g., [33, 8, 23, 4]), while others have considered different forms of adaptive partitioning that do not rely on such information (e.g., [23, 32, 4, 28, 31, 36]). These adaptive strategies tend to outperform static partitioning by tailoring partition sizes to the current load, but their performance benefits can be limited due to their inability to adjust scheduling decisions in response to subsequent workload changes. Dynamic partitioning tends to alleviate these problems by modifying the processor partitions allocated to programs throughout their execution, but at the expense of increased overhead which depends upon the parallel architecture and application workload; see [39, 16, 23, 17, 35, 9] and the references therein.

Much of this previous work has been primarily concerned with uniform-access, shared-memory parallel computer systems. In these environments, the runtime costs of a dynamic partitioning policy tend to be relatively small and thus the benefits of dynamic partitioning outweigh its associated overheads. This has been the conclusion of a number of studies (e.g., [39, 16, 35]), which have shown that dynamic partitioning outperforms other space-sharing strategies in uniform-access, shared-memory environments. Our study differs from this body of research in that we focus on distributed-memory architectures, where the costs associated with dynamic partitioning can be significant due to factors such as data and task migration, processor preemption and coordination, and reconfiguration of the application. Moreover, we extend previous research on static, adaptive and dynamic partitioning policies by effectively exploiting coarse information on the resource requirements of each job in a manner that does not impose burdensome demands on the users or the system administrator. We compare the performance characteristics of these general classes of scheduling policies from both the system and user perspectives within the context of distributed-memory parallel computers executing a workload of computationally-intensive scientific and engineering applications.

A number of studies have also considered the problem of processor allocation in parallel architectures with a hypercube system structure; see [3, 25] and the references therein. This body of work has primarily focused on constructing and allocating subcubes, often in an attempt to minimize the mean makespan of jobs, and includes nonpreemptive, preemptive and parallel scheduling algorithms. In contrast, our study considers systems where the processors can be partitioned arbitrarily with a general interconnection network that supports equidistant pair-wise processor communications (at the application level), and we focus on minimizing system- and user-level response time metrics.

The MAP and DP policies considered in this paper are respectively similar to the Adaptive Scheduling and the Reconfigurable Scheduling policies considered in [14]. The development of the policies described in that work have been influenced by the results presented in this paper. There are, however, some differences in the policy details. Despite these differences, many of the performance results from the DRMS [14, 19, 20] and Octopus [9, 11, 10] projects validate the conclusions of the work presented here. In addition, the DRMS and Octopus programming environments and runtime systems suggest that reconfigurable applications can be developed and supported on existing parallel systems.

8 Concluding Remarks

Current and near-future trends for high-performance computing centers suggest environments in which jobs with very different resource requirements arrive at various intervals to systems consisting of large numbers of processors. In this paper, we have investigated the impact of these trends on a general class of scheduling policies that space-share the system resources. Our main objective was to determine a set of principles for space-sharing large-scale scientific and engineering applications in distributed-memory environments.

Our analysis demonstrates that the system scheduling policy should be able to dynamically reduce the number of processors allocated to each job with increasing load, up to a minimum number of processors, in order to provide good performance for the entire workload. These processor allocation decisions should distinguish between jobs with large differences in execution times, since increasing the performance of a particular job class (from both the system and user viewpoints) tends to be obtained at the expense of decreasing the performance of the other classes. Our analysis further demonstrates the importance of having a scheduling policy that adapts appropriately to periods of low and high arrival activity, a common scenario in large-scale computing environments.

The DP policy considered in this paper is one approach that is capable of realizing these key scheduling objectives. Our results show that DP can yield the best overall performance from both the system and user perspectives. Furthermore, by appropriately choosing the minimum partition sizes on a per-class basis, as well as by adjusting the parameters that control the rate of reconfiguration, the desired performance differences among the various job classes can be achieved without imposing burdensome demands on the users or the system administrator.

We conclude with a few remarks on the practical implications of the policies and results presented in this paper.

Programming models: The parallel system considered in this study is modeled after MPP systems with applications that use message-passing for interprocessor communication and coordination. Such environments tend to be scalable in the sense that large systems can be built using the same architecture and the applications can be developed to run on multiple processor configurations. In fact, many applications for scalable MPP systems are written to run on more than one processor configuration. Thus, a policy such as MAP can be readily applied to current multiprocessor environments using the existing application base. A policy such as DP, on the other hand, requires that an application be capable of handling changes in the processor partition at runtime. On some systems, this may also require reconfiguring the underlying message passing subsystem in an efficient manner. Developing such applications, although non-trivial, can be accomplished using emerging programming models that support an abstraction for runtime processor reconfiguration. For example, the MPI-2 standard allows for dynamic spawning of new processes [18]. The Adaptive Multiblock PARTI (AMP) library supports an abstraction for reconfiguration that is useful for a class of SPMD programs [6]; under the AMP model the number of executable threads per application is fixed, but the number of "active" threads participating in the computations may vary during runtime. DRMS [19] and Octopus [10] environments provide abstractions for developing reconfigurable applications, as well as the necessary runtime support for application reconfiguration. We believe that as these programming environments become more mature, the application base and runtime support necessary for implementing DP-like policies will become available on production systems.

Memory requirements: We have used the min_k policy parameters to indirectly address the impact of memory requirements on the performance of adaptive and dynamic scheduling strategies. This is motivated by recent studies showing that the memory requirements of parallel jobs can have a significant effect on policy performance [26, 17, 30]. A primary conclusion of these studies is that memory requirements place a minimum processor constraint that has to be satisfied by adaptive and dynamic policies. These studies also propose time-sharing versions of adaptive and dynamic partitioning for memory-constrained workloads. As the memory requirements of applications increase, block paging and prefetching mechanisms [40] can be used together with the scheduling policies considered in this paper (as well as time-sharing versions) to directly address job memory requirements. We plan to further examine these issues in our future work.

Job resource requirements: One of the characteristics of the scheduling policies considered in this paper is that arriving jobs provide their expected resource requirements. We note that the coarseness of this information does not impose a burdensome imposition on users. Moreover, it is often in each users best interest to accurately specify this information for all job submissions (e.g., under DP, small jobs receive the best performance in the small job class, and larger jobs receive the best performance in the long run as part of the larger classes by acquiring more processors when the system load drops). The system scheduling policy, however, has to be able to deal with misinformation (intentional or otherwise) about the resource requirements of jobs. To do so, it should monitor the resource usage of each running job, and then terminate or reduce the priority of each job that exceeds the limits for its class. The latter approach requires the ability to checkpoint and archive an application (as provided by DRMS). This approach can also be used to accommodate applications for which it is not possible to determine an approximate limit on the required resources until runtime.

Acknowledgements

We thank the anonymous referees for their comments which helped to improve the style and presentation of this paper.

References

[1] Agerwala, T., Martin, J. L., Mirza, J. H., Sadler, D. C., Dias, D. M., and Snir, M. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5:63–73, 1991.

[3] Y. Chang and L. N. Bhuyan. Parallel algorithms for hypercube allocation. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 105–112, 1993.

[4] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 33–44, May 1994.

[5] Cray Research Inc, Eagan, MN. *Cray T3D System Architecture Overview Manual*, 1994.

[6] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data parallel programming in an adaptive environment. In *Proceedings of 9th International Parallel Processing Symposium*, April 1995.

[7] D. G. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *Job Scheduling Strategies for Parallel Processing,* D. G.

Feitelson and L. Rudolph (eds.), pages 337–360. Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[8] D. Ghosal, G. Serazzi, and S. K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[9] N. Islam, A. Prodromidis, and M. S. Squillante. Dynamic partitioning in different distributed-memory environments. In *Job Scheduling Strategies for Parallel Processing,* D. G. Feitelson and L. Rudolph (eds.), pages 244–270. Springer-Verlag, 1996. Lecture Notes in Computer Science Vol. 1162.

[10] N. Islam, A. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. Extensible resource mangement for cluster computing. In *Proceedings of the International Conference on Distributed Computing Systems*, May 1997.

[11] N. Islam, A. Prodromidis, M. S. Squillante, A. S. Gopal, and L. L. Fong. Extensible resource scheduling for parallel scientific applications. In *Proceedings Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[12] J. Jones. NAS Requirements Checklist for Job Queuing/Scheduling Software. NAS Technical Report NAS-96-003, NASA Ames Research Center, 1996.

[13] L. Kleinrock. *Queueing Systems Volume II: Computer Applications.* John Wiley and Sons, 1976.

[14] R. B. Konuru, J. E. Moreira, and V. K. Naik. Application-assisted dynamic scheduling on large-scale multi-computer systems. In *Proceedings of Second International Euro-Par Conference (Euro-Par'96)*, pages II:621–630. Springer-Verlag, August 1996. Lecture Notes in Computer Science Vol. 1124.

[15] S. S. Lavenberg. *Computer Performance Modeling Handbook.* Academic Press, 1983.

[16] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[17] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, May 1994.

[18] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, November 1996. DRAFT.

[19] J. E. Moreira, V. K. Naik, and R. B. Konuru. Designing reconfigurable data-parallel applications for scalable parallel computing platforms. Research Report RC 20455, IBM Research Division, May 1996. *Submitted for publication.*

[20] J. E. Moreira, V. K. Naik, and R. B. Konuru. A programming environment for dynamic resource allocation and data distribution. In *Proceedings of LCPC96, Ninth International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[21] V. K. Naik. Performance of NAS parallel benchmark LU on IBM SP systems. In A. Ecer et al., editors, *Proceedings of Parallel CFD'94*, 1995.

[22] V. K. Naik. A scalable implementation of NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2), 1995.

[23] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of Supercomputing '93*, pages 824–833, November 1993.

[24] V. K. Naik, S. K. Setia, and M. S. Squillante. Processor allocation in multiprogrammed, distributed-memory parallel computer systems. Research Report RC 20239, IBM Research Division, October 1995.

[25] B. Narahari and R. Krishnamurti. Scheduling independent tasks on partitionable hypercube multiprocessors. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 118–122, 1993.

[26] V. G. Peris, M. S. Squillante, and V. K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, May 1994.

[27] T. H. Pulliam. Efficient solution methods for navier-stokes equations. *Lecture Notes for the von Karman Institute for Fluid Dynamics Lecture Series: Numerical Techniques for Viscous Flow Computation in Turbomachinery Bladings*, 1986.

[28] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning policies of multiprocessor systems. *Performance Evaluation*, 19:141–165, 1994.

[29] H. Schwetman. *CSIM Reference Manual.* MCC Corporation, Austin, Texas, 1992.

[30] S. K. Setia. The interaction between memory allocation and adaptive partitioning in message-passing multicomputers. In *Job Scheduling Strategies for Parallel Processing,* D. G. Feitelson and L. Rudolph (eds.), pages 146–164. Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[31] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[32] S. K. Setia and S. K. Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, January 1993.

[33] K. C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1989.

[34] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY-LoadLeveler API project. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, April 1996.

[35] M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Job Scheduling Strategies for Parallel Processing,* D. G. Feitelson and L. Rudolph (eds.), pages 219–238. Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[36] M. S. Squillante and K. P. Tsoukatos. Optimal scheduling of coarse-grained parallel applications. In *Proceedings Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[37] M. S. Squillante, F. Wang, and M. Papaefthymiou. Stochastic analysis of gang scheduling in parallel and distributed systems. *Performance Evaluation*, 27&28:273–296, 1996.

[38] J. Subhlok, T. Gross, and T. Suzuoka. Impact of job mix on optimizations for space sharing schedulers. In *Proceedings of Supercomputing '96*, 1996.

[39] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[40] F. Wang, M. Papaefthymiou, and M. S. Squillante. Performance evaluation of gang scheduling for parallel and distributed multiprogramming. In *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, April 1997.

[41] S. Yoon, D. Kwak, and L. Chang. LU-SGS implicit algorithm for three-dimensional incompressible navier-stokes equations with source term. Technical Report 89-1964-CP, AIAA, 1989.

```
if (Qlen_s > 0)
  if (Qlen_l = 0) ∧ (avail_l > 0)
      t_l ← compute_target(Qlen_l, P_l, min_l, t_l, avail_l)
      while (Qlen_s > 0) ∧ (avail_l ≥ t_s/2) do
          Set current partition_size to min(avail_l, t_l)
          Qlen_s --
          avail_l ← avail_l − partition_size
          Schedule job on partition_size processors
      od
  fi
  if (Qlen_m = 0) ∧ (avail_m > 0)
      t_m ← compute_target(Qlen_m, P_m, min_m, t_m, avail_m)
      while (Qlen_s > 0) ∧ (avail_m ≥ t_s/2) do
          Set current partition_size to min(avail_m, t_m)
          Qlen_s --
          avail_m ← avail_m − partition_size
          Schedule job on partition_size processors
      od
  fi
  if (Qlen_s = 0) ∧ (avail_s > 0)
      t_s ← compute_target(Qlen_s, P_s, min_s, t_s, avail_s)
      while (Qlen_s > 0) ∧ (avail_s ≥ t_s/2) do
          Set current partition_size to min(avail_s, t_s)
          Qlen_s --;
          avail_s ← avail_s − partition_size
          Schedule job on partition_size processors
      od
  fi
fi


procedure compute_target(Qlen_k, P_k, min_k, t_k, avail_k)
  if (P_k ≤ t_k * Qlen_k) { use split strategy }
      Set t_k to max(min_k, ⌊P_k / Qlen_k + 0.5⌋)
  else
      if (avail_k ≥ 2 * t_k) { use merge strategy }
          Set n_k ← ⌊P_k / t_k⌋ − 1 and t_k ← ⌊P_k / n_k⌋
      fi
  fi
  Return updated target size t_k
```

Figure 1: The scheduling algorithm MAP($\mathtt{min}_s$–$\mathtt{P}_s$, $\mathtt{min}_m$–$\mathtt{P}_m$, $\mathtt{min}_l$–$\mathtt{P}_l$) invoked on a *small job* arrival or departure. The variables $\mathtt{t}_l$, $\mathtt{t}_m$ and $\mathtt{t}_s$ are initialized to $\mathtt{P}_l$, $\mathtt{P}_m$ and $\mathtt{P}_s$, respectively. $\mathtt{Qlen}_l$, $\mathtt{Qlen}_m$ and $\mathtt{Qlen}_s$ denote the lengths of the large, medium and small job queues, respectively.

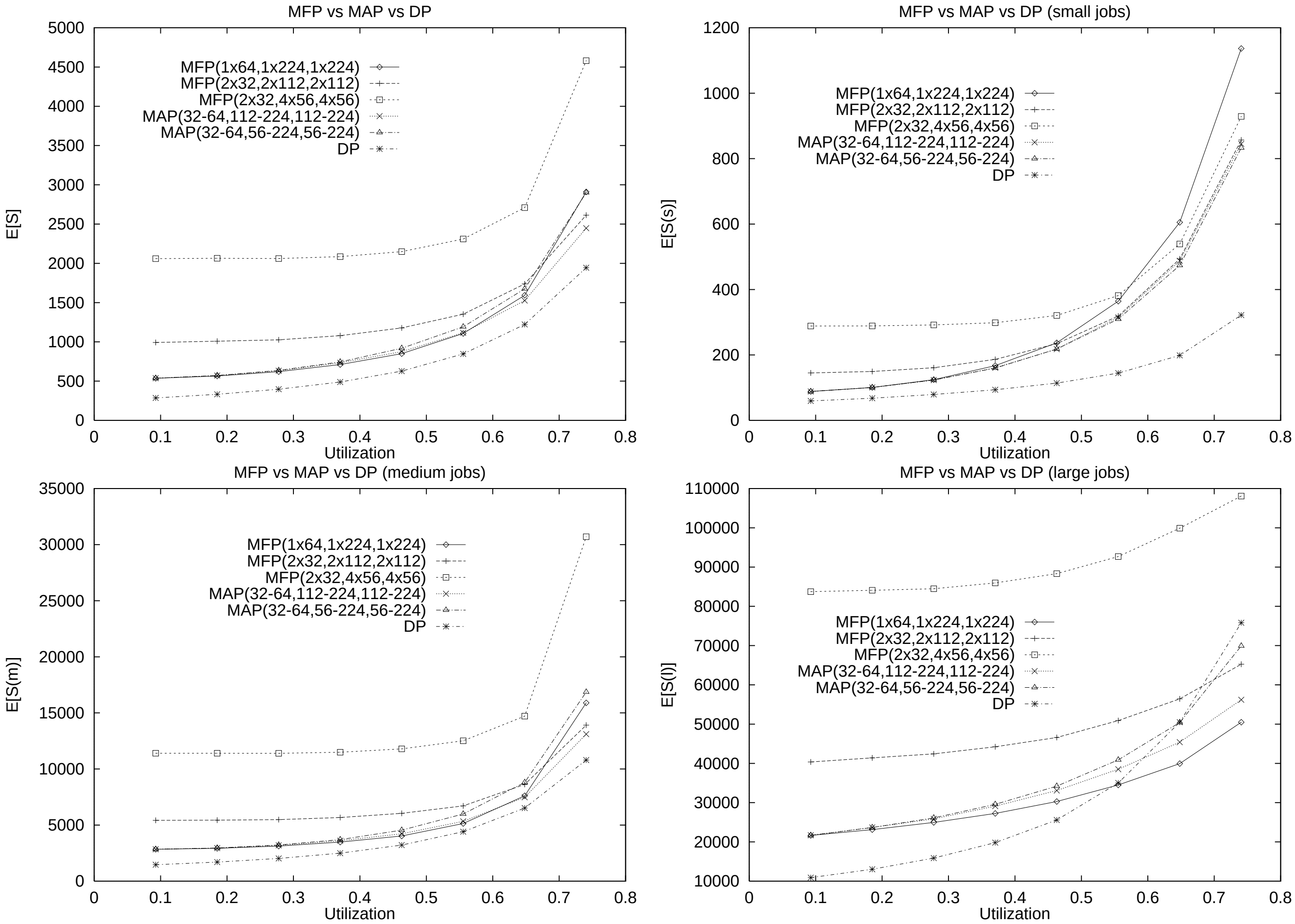

Figure 2: Average response times ($E[S]$) of MAP, MFP and DP for workload W1. The specific DP policy considered has $T = 3, f = 1, M = 32, L = 13$. For all policies, $min_s = 16, min_m = 32, min_l = 32$.

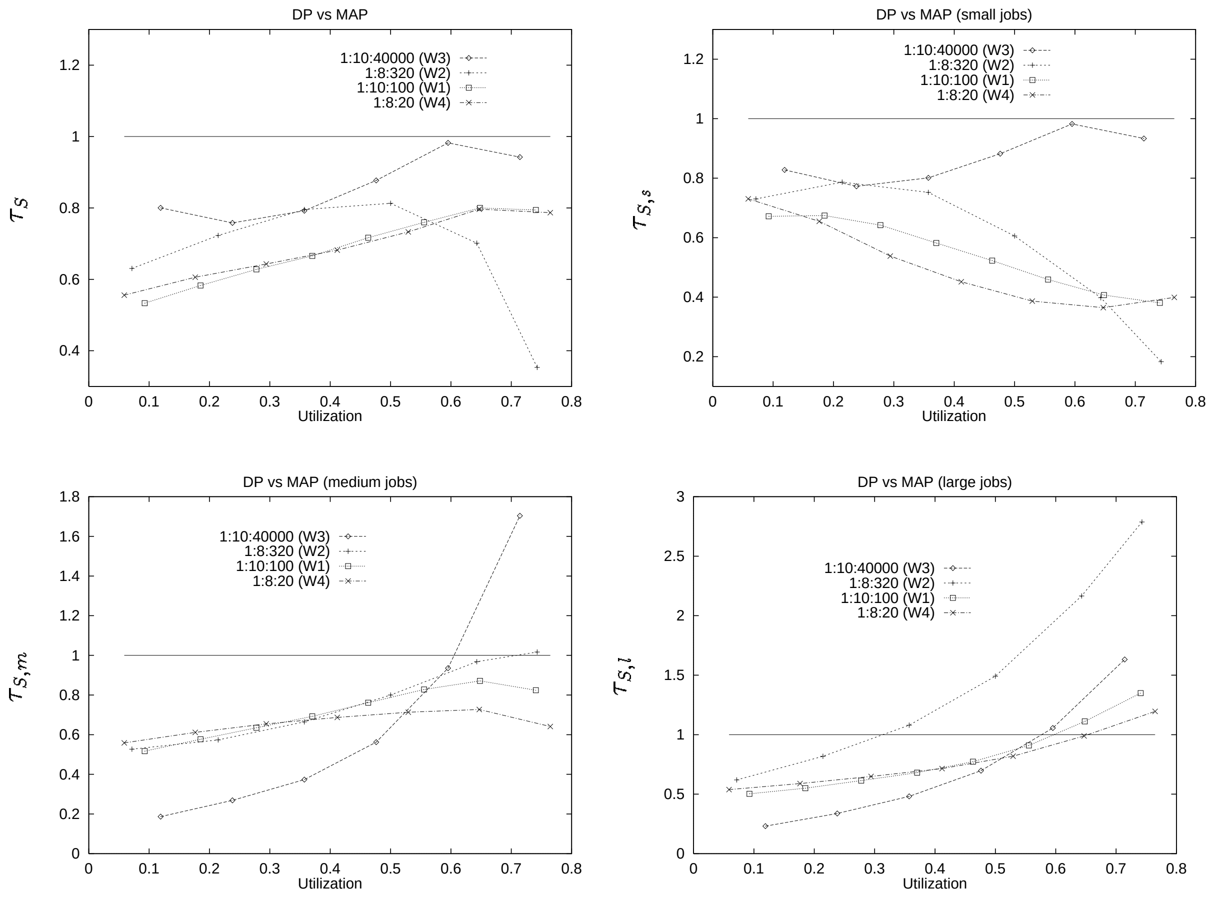

Figure 3: Relative Response time (τ_S) - DP vs MAP

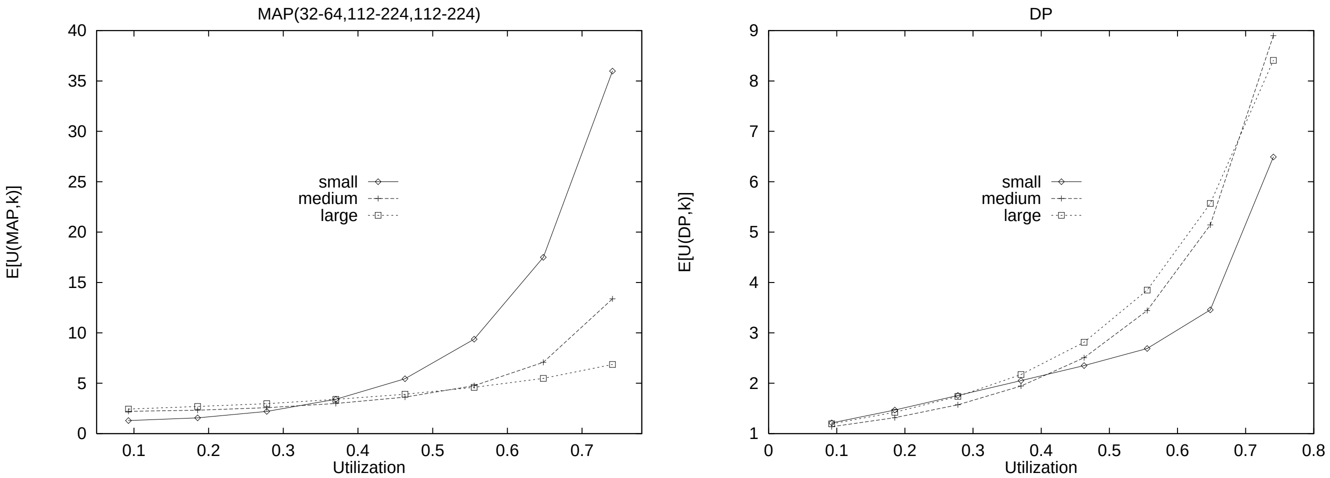

Figure 4: Average response ratio (E[U]) for small, medium and large jobs under MAP(32-64, 112-224, 112-224) and DP for W1 workload

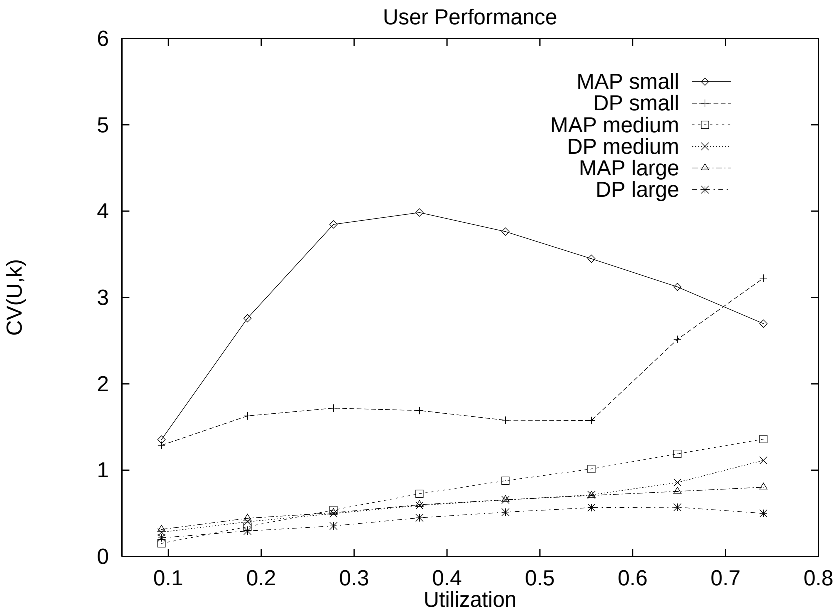

Figure 5: Coefficient of variation of the response time ratio for the different job classes under MAP(32-64, 112-224, 112-224) and DP for workload W1

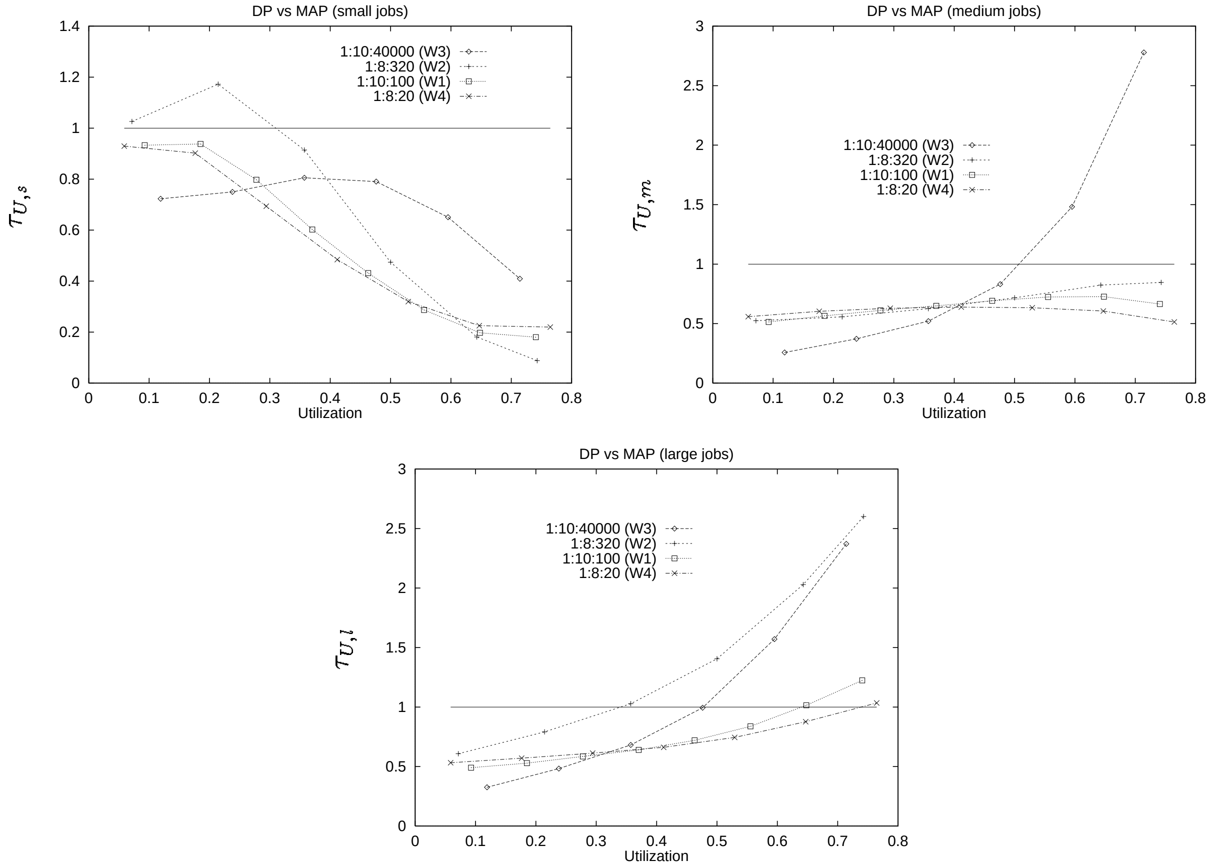

Figure 6: Relative response ratio (τ_U) - DP vs MAP

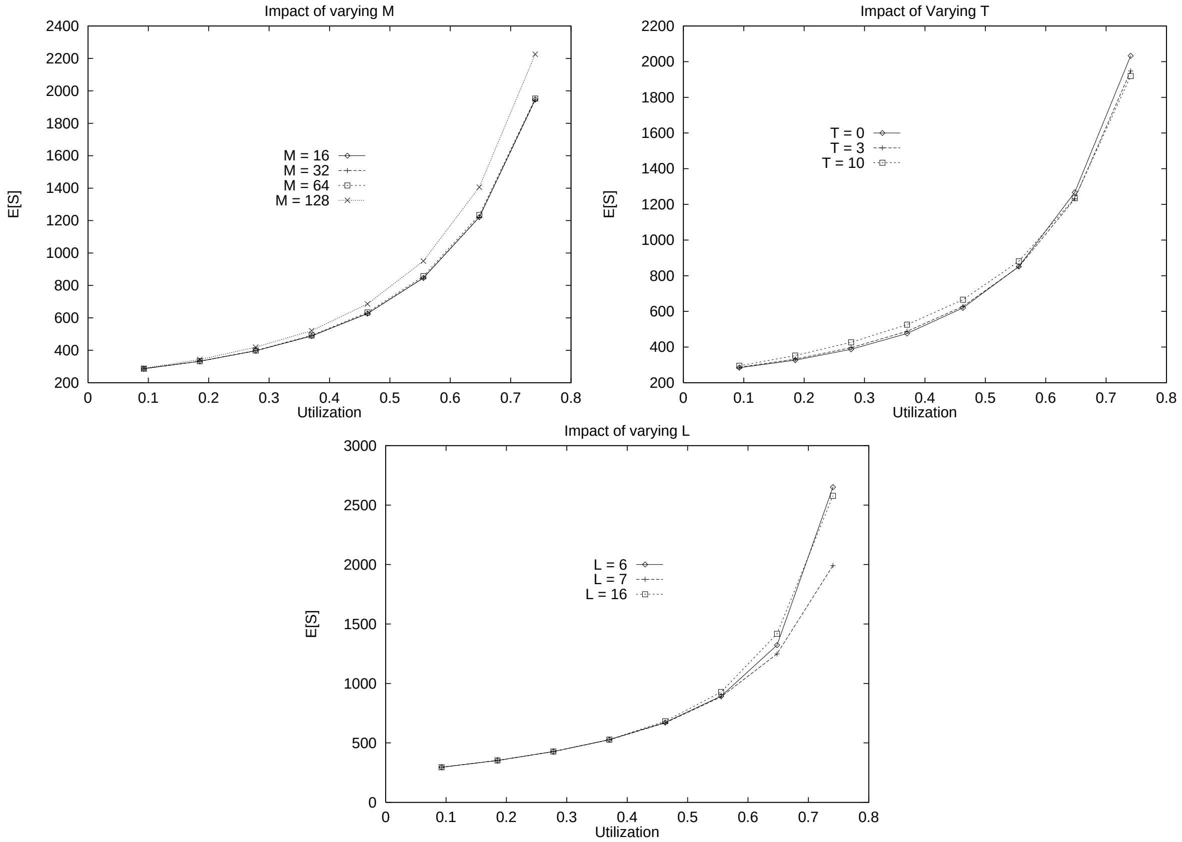

Figure 7: Impact of varying L, T and M on average response time under DP.

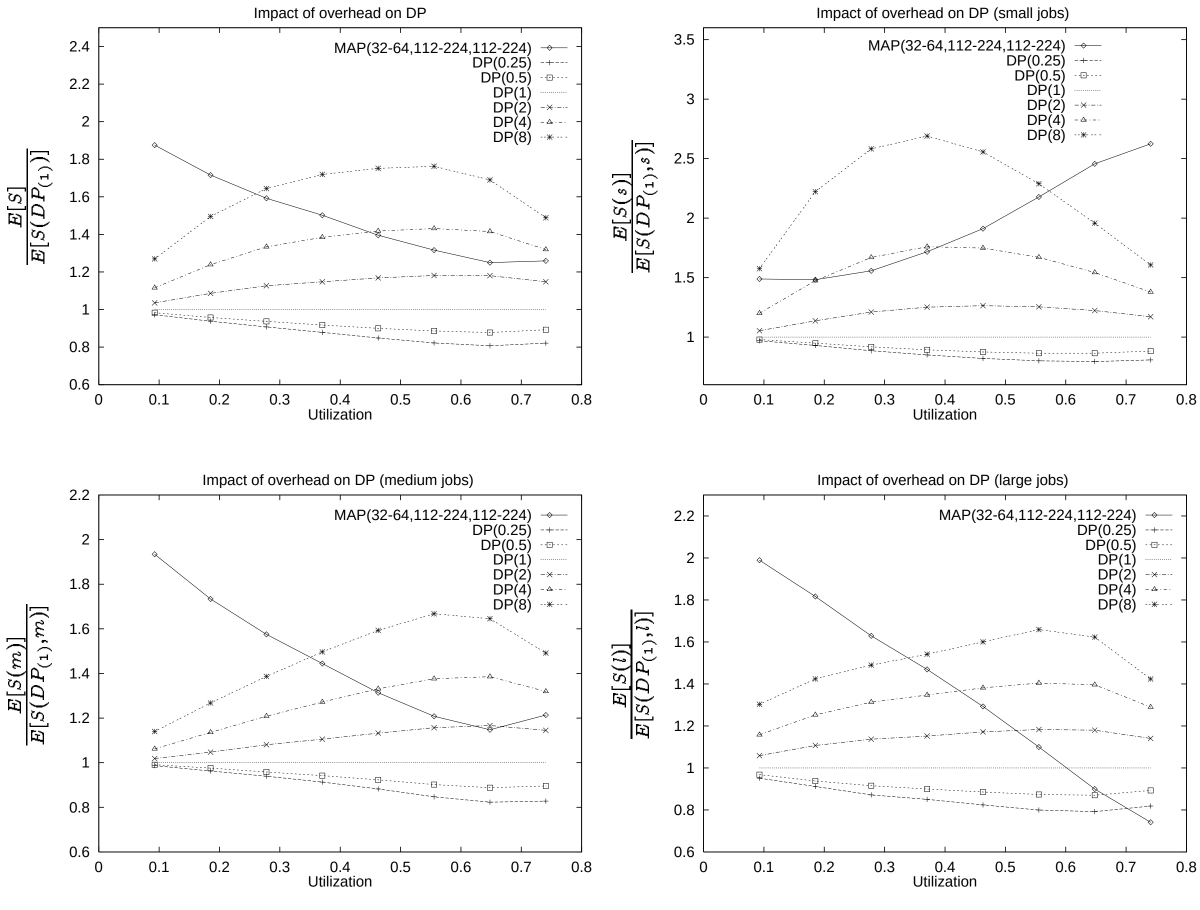

Figure 8: Relative response times with respect to $DP_{(1)}$ under workload W1 for various overhead factors

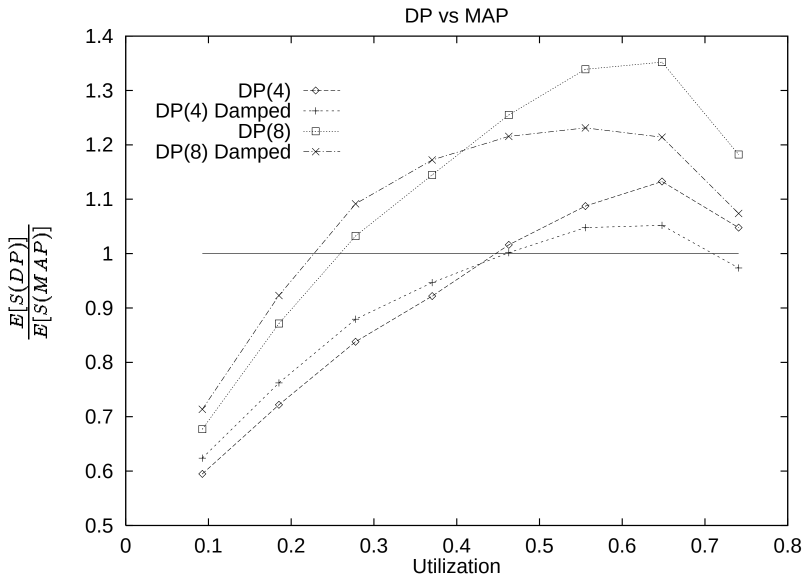

Figure 9: Impact of damping on DP performance

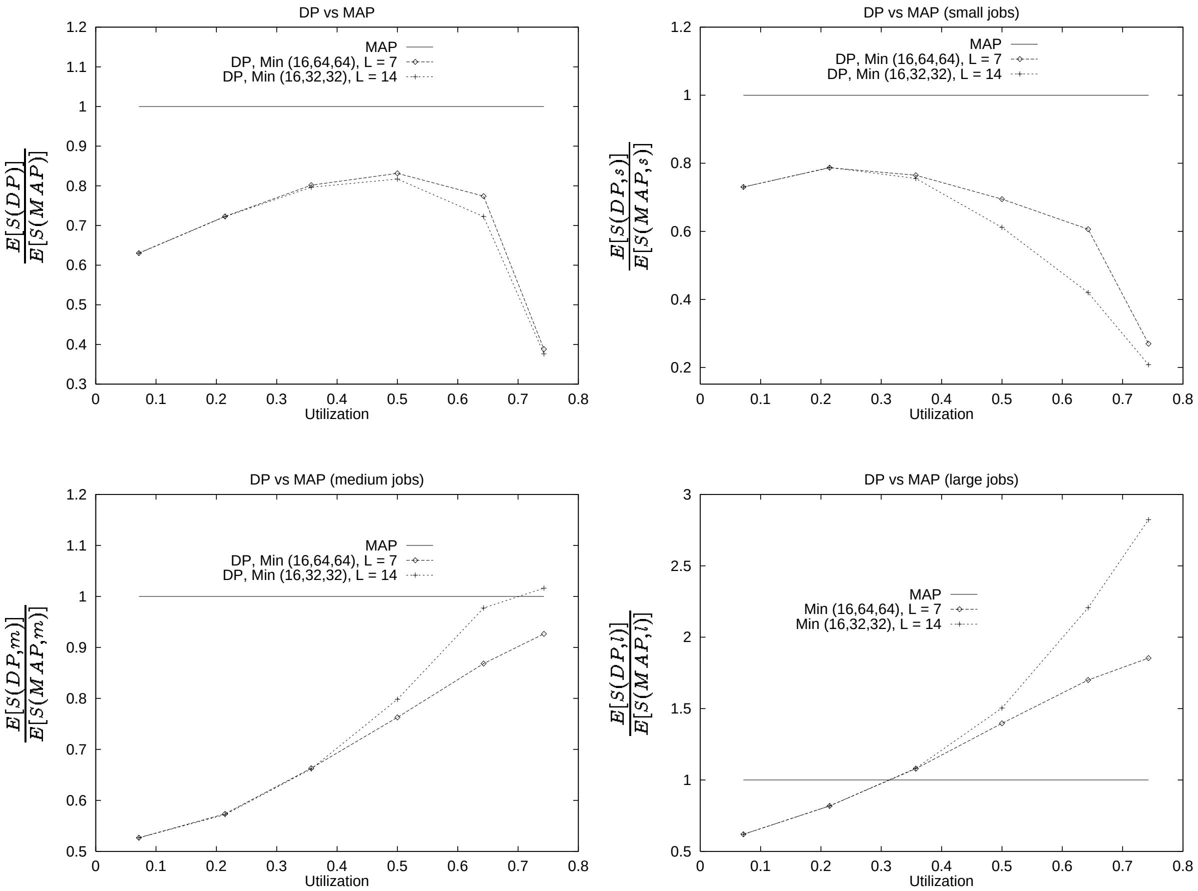

Figure 10: Improving DP performance by varying min_k and L. This figure plots the relative response time of DP and MAP for the W2 workload.

Table I: A glossary of the notation used in the description of the MFP, MAP, and DP policies

Policy	Variable	Description
MFP	s, m, l	Number of small-, medium-, and large-job partitions
	pe_s, pe_m, pe_l	Number of processors in small-, medium, and large-job partitions
MAP	P_s, P_m, P_l	Number of processors in the pool assigned to each class
	t_s, t_m, t_l	Target size under MAP for each class
	$avail_s$,$avail_m$,$avail_l$	Available processors in the pool for each class
	min_s, min_m, min_l	Minimum partition size for each job class
DP	min_s, min_m, min_l	Minimum partition size for each job class
	req_{appl}	Number of processors requested by an application
	t_{appl}	Target partition size for an application under DP
	Z	Current partition size for a job
	T	Time interval following a reconfiguration that must elapse for a job to be eligible for the next reconfiguration
	M	Minimum number of processors that must become available before initiating any expansion for an executing medium or large job
	f	Fraction of available processors that an executing large or medium job is allowed to acquire during an expansion
	L	Maximum number of large and medium jobs that can be executing concurrently

Table II: Reconfiguration costs for the 8 largest applications considered in our study.

Application #	Overhead (sec)
16	2.4489
17	4.8211
18	4.8211
19	9.4936
20	9.4936
21	18.8386
22	18.8386
23	37.3849

Table III: A summary of the job characteristics.

Class	# of Jobs	PEs	Avg. *sec*	Min *sec*	Max *sec*
Small	15	32	412	4	2532
Medium	4	128	4545	1589	7698
Large	4	128	33771	18468	48688

Table IV: The four workloads considered in our study.

Workload	$p_l : p_m : p_s$	$c_l : c_m : c_s$	cv_s(32 PEs)	cv_s (512 PEs)
W1	1:10:100	3.19:4.18:1	4.67	4.09
W2	1:8:320	0.99:1.04:1	6.91	5.25
W3	1:10:40000	0.008:0.01:1	2.48	1.47
W4	1:8:20	15.96:16.74:1	2.81	2.63

Table V: Best inter-class division of the system processors under MAP and MFP for each workload

Workload	P_s	P_m	P_l
W1	64	224	224
W2	128	128	256
W3	448	0	64
W4	32	224	256

Table VI: Reconfiguration overhead statistics for medium and large jobs under DP.

Utilization	Avg. number of contractions		Avg. number of expansions		Reconfiguration overhead (*sec*)		Fraction of Response time	
	Medium	Large	Medium	Large	Medium	Large	Medium	Large
0.1	0.65	4.77	0.65	4.73	17.4	468.9	0.01	0.043
0.2	1.39	9.73	1.37	9.62	37.0	950.9	0.021	0.073
0.29	2.22	15.45	2.21	15.23	59.4	1480.5	0.029	0.093
0.39	3.19	21.86	3.19	21.72	85.1	2065.1	0.034	0.10
0.49	4.36	29.28	4.38	29.37	116.2	2746.1	0.036	0.107
0.59	5.72	38.03	5.78	38.59	152.4	3573.9	0.034	0.102
0.69	6.85	45.63	6.97	46.81	182.8	4300.5	0.028	0.085
0.78	6.78	45.41	6.98	47.27	182.0	4303.7	0.016	0.056