RC 20390 (02/28/96)
Computer Science/Mathematics

IBM Research Report

Application-Assisted Dynamic Scheduling on Large-Scale Multi-Computer Systems

Ravi B. Konuru, José E. Moreira, Vijay K. Naik

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Application-Assisted Dynamic Scheduling on Large-Scale Multi-Computer Systems*

Ravi B. Konuru
ravik@watson.ibm.com

José E. Moreira
moreira@watson.ibm.com

Vijay K. Naik
vkn@watson.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

On multi-user large-scale multi-computers, application workload is highly variable and typically unpredicatable. In this paper, we present and analyze the performance of three scheduling policies on a spectrum of application workloads. Two of these policies make their scheduling decisions at job start-up and the resources allocated to a job remain constant until the job completion. The third policy exploits application information in dynamically reconfiguring the resources allocated to the job in order to achieve better system utilization and application throughput. These policies are implemented in the context of DRMS, a distributed resource management system that provides the necessary infrastructure for dynamically varying application resources. Our results show that the reconfigurable policy out-performs the former two policies on a wide spectrum of workloads.

*Submitted for publication at Euro-Par'96, Scheduling and Load Balancing Workshop.

Application-Assisted Dynamic Scheduling on Large-Scale Multi-Computer Systems†

Ravi B. Konuru
ravik@watson.ibm.com

José E. Moreira
moreira@watson.ibm.com

Vijay K. Naik
vkn@watson.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abstract

On multi-user large-scale multi-computers, application workload is highly variable and typically unpredicatable. In this paper, we present and analyze the performance of three scheduling policies on a spectrum of application workloads. Two of these policies make their scheduling decisions at job start-up and the resources allocated to a job remain constant until the job completion. The third policy exploits application information in dynamically reconfiguring the resources allocated to the job in order to achieve better system utilization and application throughput. These policies are implemented in the context of DRMS, a distributed resource management system that provides the necessary infrastructure for dynamically varying application resources. Our results show that the reconfigurable policy out-performs the former two policies on a wide spectrum of workloads.

1 Introduction

The recent years have seen a growth in the use of multi-user, multi-programmed large scale parallel systems with hundreds of processors for the execution of complex scientific codes. Examples of these systems include the IBM SP2 [1] and Cray Research T3D [4]. These parallel systems are usually installed at supercomputing centers and shared by many users with varying demands.

The efficient scheduling of jobs in a manner that maximizes throughput while maintaining fairness among various multi-user jobs has always been a fundamental issue for high-performance computing systems. In the past, large-scale parallel systems employed some form of *static* scheduling characterized by the division of processors into fixed disjoint sets which were the only partition sizes available for user applications. Such static policies have been shown to result in poor system utilization and throughput. Consequently, most current large-scale parallel systems employ scheduling policies that allocate resources at job initiation, thus providing more flexibility. However, once a job is allocated a certain number of processors, these policies do not change the allocation even if there is a change in system load during job execution, which has a negative impact on performance.

Dynamic policies that can modify application partitions during execution, have the potential to perform better than policies that only allocate resources at job initiation. For instance, for the example above, a dynamic policy can initially allocate a small partition to a large job during heavy loads and later enlarge the job's partition size under lightly loaded conditions. The key observation here is that such policies can improve upon prior scheduling decisions and thus have the potential to increase system utilization and throughput. However, dynamic reconfiguration of applications results in run-time overheads such as data redistribution. Thus, to be viable, it is critical for a dynamic scheme to incur low overhead while at the same time requiring minimal modifications to applications. Keeping this in mind, we have implemented a Distributed Resource Management System (DRMS) to efficiently manage system resources and at the same

† Submitted for publication at Euro-Par'96, Scheduling and Load Balancing Workshop.

time cater to the dynamically changing demands from the users. To experiment with different policies, we have designed a pluggable interface to resource management.

In this paper, we describe the DRMS architecture, its programming environment, and the run-time system that form the infrastructure for implementing dynamically reconfigurable scheduling policies. We then present and analyze the performance of three scheduling policies within the DRMS context. Two of these policies make their scheduling decisions at job start-up and the resources allocated to the application remain constant until application completion. The third policy exploits application information in dynamically reconfiguring the resources allocated to applications to achieve better system utilization and application throughput. Our results show that the reconfigurable policy out-performs the former two policies under a wide range of workloads.

The rest of this paper is organized as follows. Section 2 presents the applications used in our study. In section 3, the three policies implemented within the system are described. Section 4 presents relevant aspects of the DRMS infrastructure. Section 5, provides details regarding our performance experiments and analysis of the results. Related research is discussed in section 6 and we finally conclude in section 7.

2 Applications

In this study, we used a job mix consisting of Computational Fluid Dynamics (CFD) applications. To increase the diversity among jobs and to make the jobs more realistic, we chose applications that are based on three class of solvers each of which is used widely in the CFD community. Based on these three solvers, the NAS program at NASA Ames Research Center has developed a set of application benchmarks that are known as NAS Parallel Benchmark (NPB) suite [2]. These have become a de facto standard in evaluating parallel systems for technical computing. The three application benchmarks are known as appsp, appbt, and applu. All three benchmarks compute numerical solution to a synthetic system of five nonlinear partial differential equations (PDE's) that represent some of the key characteristics exhibited by the Navier-Stokes equations. Each application uses an implicit type of numerical solution in solving the system of PDE's and the solution phase consists of several time steps (or iterations). The three benchmarks differ in the solution method used. For complete details on the problem solved and the numerical techniques used by these benchmarks, refer to Chapter 3 of [3].

The computation, communication, and memory requirements of the three benchmarks differ considerably and so do the data dependencies, data structures, available parallelism, and the communication patterns. The NPB suite specifies two data sets ($64 \times 64 \times 64$ and $102 \times 102 \times 102$) for these benchmarks. For our study, we modified these benchmarks to create applications that can handle two additional data sets ($80 \times 80 \times 80$ and $126 \times 126 \times 126$). In addition, to get a larger range of computational requirements, we considered an iteration space ranging from 100 through 40,000. Thus, altogether we consider a pool of 21 different applications such that when computed entirely on a single processor, computational requirements of these applications vary from less than 1,000 seconds to approximately 100,000 seconds. Also, when computed on a single processor, their memory requirements vary from about 50 MB to approximately 1.5 GB. Of course, in our experiments, the jobs requiring long execution times and large amounts of memory were not run on a single processor. The memory and execution time range is typical of supercomputing centers where both application development (small data sets and fewer iterations) and production runs (larger data sets and many iterations) are carried out simultaneously.

For the experiments reported in this paper, we started with SPMD versions of the above described applications that were tuned for the IBM SP2 [19, 21, 20]. We modified these by inserting DRMS annotations (which we describe later in the paper) at the beginning of each iteration so that the applications can be potentially reconfigured to execute on a different set of processors every fifth iteration. In Table I, we tabulate the 21 applications we used in our study and their execution time measured on IBM SP2 with wide nodes. All timing measurements in this table and other tables in this paper are in seconds. As shown in column 1, we group the 21 applications into three categories based on their computational requirements. The

Table I: Applications used in the experiments.

Job category (PE range)	Application<size,iters>	Time on min PEs	Time on max PEs
I (1,2,4)	appsp<64,200>	677	184
	appsp<64,400>	1271	344
	appsp<80,100>	827	203
	appsp<80,200>	1419	361
II (4,8,12,16)	appsp<64,5000>	4034	1327
	appsp<80,2500>	3995	1221
	appsp<102,1250>	3906	1176
	appbt<64,1600>	3554	1045
	appbt<80,800>	3435	979
	applu<64,4000>	7135	2169
	applu<80,2000>	3227	1036
III (8,12,16,20,24,28,32)	appsp<64,40000>	18332	6324
	appsp<80,20000>	18244	5348
	appsp<102,10000>	17248	5195
	appsp<126,5000>	16639	4746
	appbt<64,12000>	13811	4132
	appbt<80,6000>	13399	3961
	appbt<102,3000>	13990	3702
	applu<64,32000>	29800	9538
	applu<80,16000>	13790	5237
	applu<102,8000>	16990	5082

first four applications require less than 1500 seconds when computed on a single processor. Jobs belonging to this category of applications will be referred to as the category I jobs. We configured these applications to run on 1, 2, or 4 processors. The next set of seven applications require between 3,000 to 4,000 seconds (except for applu<64,4000>) when executed on four processors. We refer to jobs belonging to this category as category II jobs. We configured the applications from this category to run on 4, 8, 12, or 16 processors. Finally, the last ten applications form the category III jobs. The execution time of these ten applications on 8 processors is between 13,000 and 20,000 seconds. The category III applications were configured to run on 8, 12, 16, 20, 24, 28, and 32 processors. In column 2 of Table I, for each application we indicate the problem size and the number of iterations required to solve to completion. Listed in columns 3 and 4 are the execution times on the minimum and the maximum number of processors they are configured to run on.

All of our applications go through three phases during the course of their execution: (i) *setup* phase, (ii) *solution* phase, and (iii) *summary* phase. The solution phase consists of several iterations, each performing the same number of operations. As seen from Table I, some of the applications we chose differ only in the number of iterations (e.g., appsp<64,200>, appsp<64,400>, appsp<64,5000>, and appsp<64,40000>). The total execution time of these applications differ from one another only in the solution phase and not in the setup or the summary phase.

3 Scheduling Policies

We consider three scheduling policies for this study: (i) the lazy scheduling (LS) policy (ii) the adaptive scheduling (AS) policy, and (iii) the reconfigurable scheduling (RS) policy. In all cases, the scheduler maintains the jobs in the arrival queue prioritized based on the arrival time. Each arriving job defines a list of processors over which it can be scheduled to run. The policy allocates to a job one of the eligible number of processors defined by the job's processor list. We refer to the minimum over the processor list as the *minimum requested number of processors* and the maximum as the *maximum requested number of processors*.

3.1 The LS policy

Under the LS policy, the scheduler continuously scans the job arrival queue and schedules the earliest job that can run on all or on a subset of available processors. In other words, LS is a *first to fit* policy. The number of processors allocated to a job is always equal to one of the eligible number of processors for that job, with an allocation preference towards maximum eligible number of processors. Once a job is scheduled to run on a set of processors, that job runs till completion on those processors. When the load is light, LS policy tends to schedule jobs in the order in which they arrive (lower average queueing delays) and the number of processors allocated to a job is towards the maximum requested by that job (smaller service time). Thus, the expected response time is small for all jobs. When the load is heavy, jobs requesting larger number of processors get lower priority (higher queueing delays) and the number of processors allocated to a job tends to be towards its minimum requested number of processors (longer service time). Thus, the expected response time is high for all jobs and it is the worst for jobs requesting larger number of processors (i.e., the jobs belonging to the large job category). At intermediate load levels, this policy favors jobs requesting smaller number of processors (i.e., small job category) by allocating processors closer to the requested maximum. Thus, the response time is relatively higher for the category III jobs than for the category I jobs.

3.2 The AS policy

Under the AS policy, whenever processors are available to schedule jobs, the scheduler tries to schedule jobs in the order in which they arrived, but instead of scheduling the earliest job on the maximum possible number of processors, it tries to schedule as many of the currently waiting jobs in the arrival queue as possible. In other words, AS is a *maximum to fit with priority* policy; it tries to schedule as many of the jobs as possible, without preemption of schedulable higher priority jobs. The rationale is that this would increase fairness in the processor allocation strategy. As in the case of LS policy, once a job is scheduled to run on a set of processors, that job runs till completion on those processors. When load is light, this policy behaves similar to LS in the sense that each job is allocated closer to its requested maximum number of processors. When load is heavy, the number of processors allocated to each job tends to be closer to its requested minimum, regardless of the job mix. This helps in reducing "fragmentation" of the processor space at moderate to heavy loads and thus improves overall processor utilization. As compared to this, under LS policy, when load is moderate to heavy, the available number of processors at any time tends to be dictated by the average job partition size. Thus, if the processor requirements of arriving jobs are distributed uniformly over the entire range, then, on an average, half of the jobs cannot be scheduled when a job departs. However, if the variance is not large and majority of the jobs can run over a small number of processors, then the difference between LS and AS diminishes and, in some cases, LS might perform better than AS.

3.3 The RS policy

Under the RS policy, when there are processors available to schedule jobs, jobs are scheduled in the same way as in the case of the AS policy. In addition, when not enough free processors are available and there are jobs to be scheduled, it tries to free up processors from jobs that are currently running on more than their minimum requested number of processors. Similarly, when there are no jobs waiting to be scheduled and free processors are available, RS tries to expand one or more of the currently running jobs to run on a larger set of processors. The policy incorporates parameters that can be used to determine (i) the minimum interval that must elapse before a running job can be reconfigured and (ii) the number of processors by which a processor partition of a running job is to be expanded or shrunk. When there are no job reconfigurations, RS defaults to AS. By reconfiguring the jobs that have already been scheduled, RS tries to adjust prior scheduling decisions to meet the current demand on the resources. Thus, the RS policy tries to adapt processor partition sizes of new and existing jobs to the dynamic changes in the load. This policy performs the best when jobs can be reconfigured as frequently as necessary and to any desired number of processors. This flexibility, however, is achieved at a cost and that is the cost of of reconfiguring jobs that are already running. The lower the reconfiguration costs and the more frequently and at a finer granularity a job can be reconfigured, the better is the performance of the RS policy as compared to the AS policy.

We note here that, for all three policies described above, the number of processors to be allocated to a job is not determined a priori (i.e., statically) but is determined dynamically based on the state of the system at the time the job is scheduled. The same job, when submitted at two different times, may be scheduled on two different sets of processors. In that sense, all three policies are variations of dynamic scheduling. The degree of dynamicity varies among the policies. AS is more dynamic than LS since it takes into account the demand on the system, in addition to the available number of processors, before scheduling a job. RS is even more dynamic than AS, since it makes changes to the partitions of already scheduled jobs to adapt the overall processor allocation strategy to the current demand on the system.

4 DRMS Infrastructure for Reconfigurable Scheduling

We have designed and implemented DRMS for supporting not only traditional scheduling policies but also those that can exploit knowledge about dynamic changes in application needs. To reduce the effort in developing reconfigurable code, application writers are provided with a set of program annotations for exposing application global data and dynamic resource requirements. A compiler transforms these annotations into executable code. DRMS also provides an run-time system that interfaces with the rest of the DRMS and shields the programmer from many of the complexities of dynamic reconfiguration. These complexities are handled by a set of components that implement resource management, and a pluggable policy component that implements the specific scheduling policy. This section presents an overview of the DRMS architecture and a brief description of the major components of DRMS relevant to dynamic scheduling.

4.1 DRMS architecture: an overview

Figure 1 shows the main functional components of DRMS and the primary interactions among these components. The functional components that perform the resource scheduling task are the Resource Coordinator (*RC*) and the Job Scheduler and Analyzer (*JSA*). The run-time management and coordination of user applications is accomplished by the User Interface Coordinator (*UIC*), *RC*, and the Task Coordinator and Run-time Monitor (*TC*). A run-time system (RTS), linked to each application, coordinates with the external environment and propagates resource requests from the program to the resource management components of DRMS. It also implements the data distribution operations to support reconfigurable applications. The performance analysis component is handled by run-time performance data gatherer and the associated tools and utilities.

The system-level allocation and scheduling decisions are made by *JSA* based on the implemented/enforced scheduling policies. These decisions may take into account information such as application supplied resource requests, job priorities, individual processor utilization, as well as system-level information such as current and expected workload. A particular implementation of *JSA* can use the information gleaned from the performance analysis tools in addition to its own knowledge of the applications for its decision making. Policies for making such decisions can be supplied and modified by system administrators. The *JSA* does not

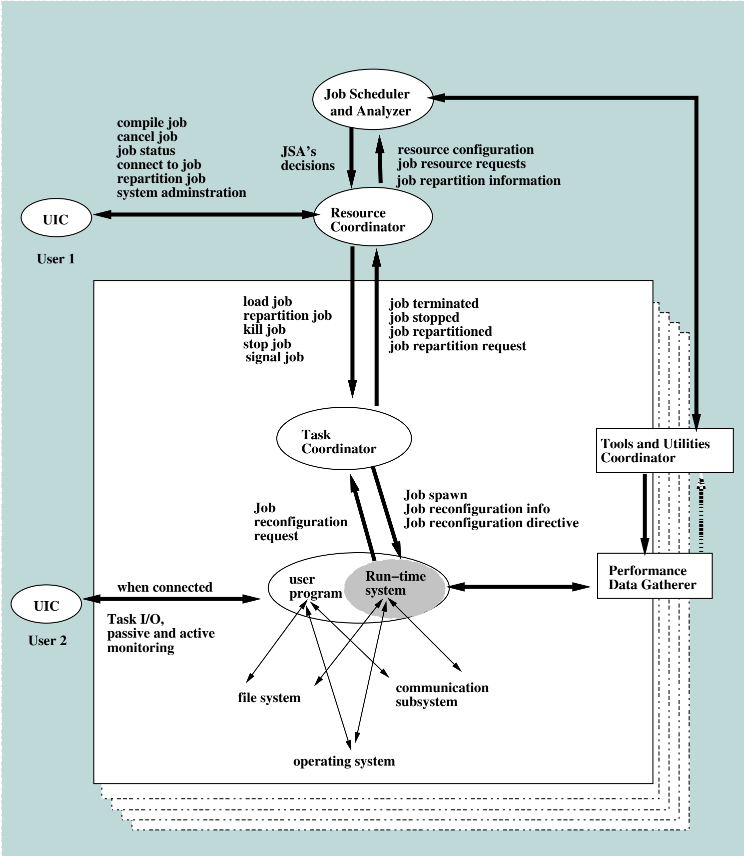


Figure 1: DRMS Architecture

interface directly with the user or user program, but rather it communicates its decisions to *RC*. *RC* interacts with *UIC* and *TC*. There is only one logical *RC* and *JSA* for the entire DRMS.

Each user application has an associated *TC*, which consists of multiple agents, one per processor on which the user application is scheduled for execution. One of the agents acts as a master for coordination with the external world including *RC* and *UIC*. The run-time interactions between the user application and the rest of the system, including other applications, the user, and *RC*, are managed by various subcomponents of *TC*. The main functions carried out by *TC* are: acquiring/releasing processors from/to *RC* and starting user and restarting program execution on the allocated processors.

The user submits jobs and interacts with the system throughout the course of the program execution via the *UIC*. The primary function of *UIC* is to provide convenient user interface. The performance gathering component is designed to assist users and the system administrators to understand some of the characteristics of user programs [6].

A job interacts with the resource management system only at user-specified points in the application program execution called Schedulable and Observable Points (SOPs)[5]. However, the *TC* coordinates external interactions throughout the course of job execution. When the *JSA* decides to reconfigure a job (expand or shrink) during the course of its execution, this decision is conveyed to *RC*. *RC* relays this decision to the *TC* of that job which in turn delivers this message to the RTS asynchronously. At the next SOP, the RTS interprets the reconfiguration message, causes synchronization among the tasks of the application, and communicates with the *JSA*. After the initialization of the application's new processor partition, the RTS is informed of the new partition information. Then the RTS, in conjunction with the *TC*, rearranges the application data so that the program can run on the new set of processors. Thus, the reconfiguration cost as seen by the application is the time elapsed between the initiation of resource request(or resource change) to the time at which the application resumes execution on the new set of allocated processors.

An application can also voluntarily initiate an expansion or shrinkage of its allocated set of processors. At a program-initiated expansion, the *TC* acquires the additional processors from *RC* and reschedules the computations on the larger processor partition while coordinating with the RTS in redistributing application data. In this paper, we consider only *JSA* directed application reconfiguration.

4.2 Run-Time System

The run-time system consists of two components: the application run-time system and the DRMS run-time system. The application run-time system is responsible for supporting the distributed data structures used by the application. It builds and maintains descriptors for virtual processor grids and distributed arrays on each processor executing an application. It also performs all the necessary data movement when a data redistribution is necessary, such as during a reconfiguration of allocated processors. Because it has knowledge of the distributed data, it can perform a series of utility functions on that data. Some of these functions will be described in the next section.

The DRMS run-time system forms the interface between the application and the local *TC*. Using this interface, the DRMS application run-time system inquires and conveys information about changes in allocated resources and dynamically reconfiguring application data using the information gleaned from the annotated application and the current execution state. The RTS supports blocking and non-blocking requests for additional resources and propagates information to the local *TC* regarding future decrease in resource usage by the application. Further, it also handles asynchronous notification of reconfiguration events from the local *TC*.

4.3 Language support for reconfigurable applications

For implementing policies such as RS, and also to allow applications to modify their resources dynamically, applications have to be *resizable*, that is, they must be able to accommodate changes in the size of their processor partition while they are executing. DRMS provides a set of *language extensions* and *library functions* for SPMD Fortran programs that facilitate the task of designing resizable applications. These extensions implement the DRMS programming model, a modified SPMD model. In the classical SPMD model, a group of processors all execute the same code, with each processor applying the code to a different section of the data set. In the DRMS model, a parallel program execution consists of the consecutive execution of a number of SPMD *stages*, each on a (potentially) independent set of processors.

We define a stage as a unit with its own resource requirements, data distribution, and execution code. In our model, a stage consists of four sections: *resource section*, *data section*, *control section*, and *computation section*. Each section is, in general, parameterized on the actual resource availability and other problem specific factors, such as data size, numerical conditioning, etc. The *resource section* specifies the type and quantity of each resource necessary for the execution of a stage. The *data section* specifies how the global data structures should be distributed among the processors executing the stage. Each processor has its own independent data space. The local data space of a processor is used to store local variables that are only used by the corresponding processor. It is also used to store the *local partitions* of global data. The local partitions of a processor contain problem data that, in principle, is pertinent to all processors, and therefore can be accessed, directly (remote memory operation) or indirectly (message passing), by any other processor. The *control section* specifies values for variables that control the execution of code inside a stage. Usually, the flow of this execution depends on resource allocation and data partitioning. The *computation section* specifies the computations to be performed in each processor, as well as the communication that occurs between processors.

```
        DO i = 1, m
c$DRMS$    RESIZE 1:16:*2
           CALL drms_get_new_numproc(num_pes)
           npx = sqrt(num_pes)
           npy = num_pes/npx
c$DRMS$    PROCESSORS, DIMENSION(npx,npy) :: P
c$DRMS$    DIMENSION(n,n), DISTRIBUTE(BLOCK,BLOCK) ONTO P,
c$DRMS$    BORDERS((1,1),(1,1)) :: u0, u1
           xsize = drms_local_extent(u0,1)
           ysize = drma_local_extent(u0,2)
           IF (MOD(i,2) .eq. 0) THEN
              CALL drms_update_borders(u1)
              DO j = 2, ysize-1
                 DO i = 2, ysize-1
                    u0(i,j) = h + 0.25*(u1(i-1,j)+u1(i+1,j)+u1(i,j-1)+u1(i,j+1))
                 END DO
              END DO
           ELSE
              CALL drms_update_borders(u0)
              DO j = 2, ysize-1
                 DO i = 2, ysize-1
                    u1(i,j) = h + 0.25*(u0(i-1,j)+u0(i+1,j)+u0(i,j-1)+u0(i,j+1))
                 END DO
              END DO
           END IF
        END DO
```

Figure 2: Example of Jacobi relaxation using DRMS.

DRMS language extensions are in the form of *annotations*, source level commands that are ignored by a regular Fortran compiler. The DRMS compiler translates these annotations into executable Fortran code in a preprocessing phase. The output of the preprocessor is then compiled by a regular Fortran compiler. For the sake of brevity, rather than going through an exhaustive description of all annotations and library functions that support partition reconfiguration, we give a flavor of our language support by means of an example. We refer the interested reader to [17] for a more complete description of language and library issues in DRMS. Figure 2 shows the main loop of a Poisson solver using Jacobi relaxation. Each iteration of the loop performs one step of the relaxation and constitutes one stage of the execution of the program. The body of the loop uses several DRMS annotations and functions that we now describe.

The first annotation, `RESIZE`, is an examples of a *resource specification* annotation. It specifies how many processors are necessary for the execution of the stage. In this particular case, the code specifies a range of partition sizes from 1 to 16 in multiplicative steps of 2. Therefore, valid partition sizes for the execution of a stage (loop iteration) are (1,2,4,8,16). DRMS also allows the specification of ranges with additive step, ranges of powers of integers, single partition size, all possible sizes, and irregular ranges specified in the form of lists. Note that each iteration step can be executed on a different partition size.

Once resource requirements are specified and a valid set of resources obtained from the system, the code declares how to distribute data onto the available processors. In this example, a DRMS library subroutine, `drms_get_new_numproc()`, is used to find out how many processors were actually allocated. That number is factored into two components (`npx`,`npy`) in order to create a grid `P(npx,npy)` of virtual processors. Such grids are created with a *processors declaration* annotation. The DRMS `PROCESSORS` declaration is similar to the HPF `PROCESSORS` directive [12], but, although not shown here, it also allows explicit control on the mapping of virtual to physical processors. Both one-to-one and many-to-one mappings are supported.

After the processor grid is created, the global data arrays `u0` and `u1` are declared, using a *data declaration* annotation. These arrays are specified as having a global shape $n \times n$ and `BLOCK` distribution along both axes. The shapes of the local partitions for each processor are automatically computed by DRMS. If there is a change in the number of processors from iteration i to iteration $i+1$, data is automatically moved across processors to conform to the new specification. DRMS supports all forms of data distribution of standard HPF: block, cyclic, block-cyclic, and collapsed. It also supports an additional form of regular block distribution, called `BLOCKD`, that achieves better load balancing, and two forms of irregular distributions: block-list (similar to Fortran D [8]) and arbitrary, which is specially useful in manipulating sparse matrices. As an additional difference from HPF, DRMS allows the declaration of *borders*. Borders are regions that extend the local partition of a processor beyond the *base partition* implied by the distribution. In the Jacobi example, all partitions are extended by 1 in both directions. The borders on the edge of the processor grid represent problem boundaries, while the other borders overlap with the base regions of other processors.

Once data is distributed, another DRMS library function (`drms_local_extent()`) is used to find the extents of the local partitions of each processor. Those extents are used to determine loop bounds for the computation of the relaxation. Before performing one step of the relaxation, the border regions have to be updated with data from the actual owners of the overlapped base partition. This is accomplished with a call to `drms_update_borders()`, yet another DRMS library subroutine.

Note that this was an example of a code specifically designed to run only under DRMS, since it makes use of DRMS functions even in the computation section. It is also possible to just use DRMS annotations for declaring resources and data distribution, and then use the communication functions available in the particular machine (remote memory operations, MPI, PVM) to perform the update of borders. The code could then also be compiled by a regular compiler, that would ignore the DRMS annotations, and executed on a fixed partition.

5 Performance Analysis

The objective of this study is to evaluate the performance of the three scheduling policies as implemented in the context of DRMS. To actually measure the performance of DRMS under various steady state job submission conditions, it would require long experiments on large number of processors. Such experiments are not practically possible at this developmental stage of DRMS. As a compromise, we measured several performance parameters of DRMS and used them in a DRMS system-level simulator. In this section, we first present the performance results of DRMS. This is followed by the description of the simulator. Finally using the results of our simulations, we analyze the performance of the three scheduling policies.

5.1 Performance results from DRMS

This section presents preliminary performance results of the DRMS system running on a 32-node IBM SP2 with wide nodes[1]. We used the SP2 system at NASA Ames Research Center for these experiments.

As discussed in section 4.1, the total reconfiguration cost consists of four components: (i) the synchronization among the tasks of the application at an SOP and the communication with the *JSA*, (ii) partition initialization, (iii) spawning the application on the reconfigured processor partition and (iv) redistributing application data. In our experiments, we measured the total reconfiguration cost, the partition initialization cost and the application spawn cost. From these three costs, we derived the cost of task synchronization and communication with the *JSA*.

Table II summarizes the reconfiguration and redistribute costs measured for the applications considered. During execution, jobs from category II and III were reconfigured several times, with the allocated processors ranging from 4 to 32. The table shows the minimum and maximum of each of the resize and redistribute costs for each application. Intuitively, as the problem size of an application grows, the time to resize and redistribute should increase and this is observed in Table II. Note that resize and redistribute costs of some of

Table II: Measured reconfiguration and redistribute costs for applications under DRMS.

Application	Reconfiguration time		Redistribute time	
	min (s)	max (s)	min (s)	max (s)
appsp<64>	4.51	7.17	1.15	1.94
appsp<80>	5.35	5.35	1.69	1.69
appsp<102>	6.42	20.40	3.52	6.66
appbt<64>	4.07	12.30	1.31	2.74
appbt<80>	6.07	7.21	2.37	4.90
appbt<102>	7.80	9.31	4.08	5.94
applu<102>	5.35	11.60	1.42	3.44

the applications (appsp64, appsp102, applu102) show a larger variation than for other applications because these were long-running and executed on a larger range of processors and were subjected to multiple resizes.

The difference between the redistribute cost and resize cost is accounted by the costs of task synchronization and communication with the *JSA*, partition creation and spawning the application on the new partition (costs (i), (ii) and (iii) listed above). We measured the average partition initialization cost to vary from about 1.3s for 1 processor to about 2.0 seconds on 32 processors. The average cost of spawning an application varied from about 0.3s for one processor to about 2.5 seconds on 32 processors.

5.2 The DRMS System Simulator

Applications are modeled in the simulator at a very high level. Each application type shown in Table I is characterized by its setup time, its summary time, its iteration time, its number of iterations, and its reconfiguration and redistribute costs. The setup, summary, and iteration times are dependent on the type of the application and number of processors on which it is executed. The execution of an application consists of the execution of its setup phase, the specified number of iterations, and its summary phase. When an application is reconfigured, its execution time incurs a *reconfiguration cost*. We used the reconfiguration and redistribute costs obtained from measurements performed with DRMS for our application suite, as shown in Table II. The actually measured costs vary with the source and target partition sizes, and the values we used are upper bounds from these measured costs. Therefore we expect our system to perform even better than simulated. To simplify simulation, we made the reconfiguration and redistribute costs dependent only on problem sizes. For problem sizes of 64, 80, 102, and 126 the reconfiguration costs are 8s, 11s, 17s, and 29s, respectively. The redistribute costs are 3s, 6s, 12s, and 24s respectively.

The central point of the simulator is the job scheduler, which mimics the actions of the JSA in DRMS. The simulator is event-driven, with 6 different types of events that trigger the job scheduler: (i) *Job arrival:* A job of a given type arrives in the system. This job is put in the job scheduling queue. (ii) *Job start:* A job begins executing on a given number of processors. The processors become busy. (iii) *Job finish:* A job finishes execution and leaves the system. The processors on which it was executing become free to be used by other jobs. (iv) *Job eligible:* A job becomes eligible for resizing. This occurs when a minimum reconfiguration interval has elapsed since its last reconfiguration or since the job started. (v) *Job reconfiguration:* A job acknowledges a reconfiguration request. It queries the JSA for the actual change in its number of processors. If such a change occurs, the job reconfigures. Its new finish time is recomputed and a new *job eligible* event is created. (vi) *Job shrink:* A job that was asked to shrink completes its redistribution. The released processors become free and can be scheduled with new jobs by the JSA. At each of these six events, the job scheduler is invoked in order to schedule jobs for execution and decide which jobs need to be reconfigured.

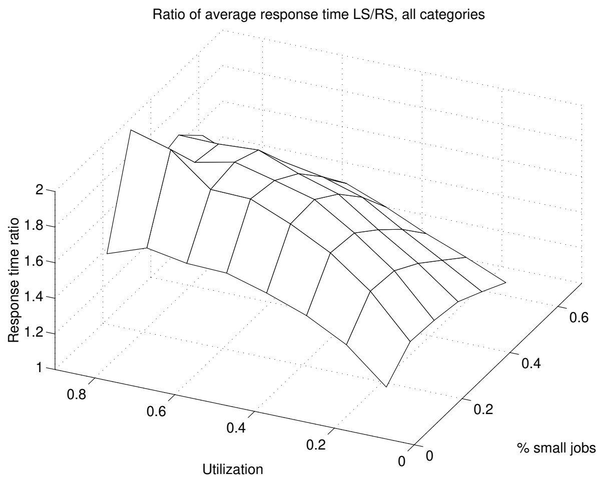
Job arrival times are generated using an exponential distribution, with different mean inter-arrival times for each job type. The mean inter-arrival time for a job type is computed in order to deliver the desired system load. Statistics are gathered during the simulation. Measured performance parameters are the response times and service times for each job type and for the job mix as a whole. For these parameters we measure both mean (μ) and standard deviation (σ) of n samples. To reduce the correlation between successive samples, we use the batch means method as described in [11]. The number of job reconfigurations is also counted for each class. After the system achieves steady state, the simulation runs until at least a minimum number of jobs of each type completes execution. It then keeps running until the standard error ($\delta = \sigma/\sqrt{n}$) of the response time is less than $\xi\mu$ for each job type and also for the whole job mix. In our experiments, we use $\xi = 0.01$, which results in a standard error of less than 1% of the mean for each of the response times. Consequently, the 95% confidence interval for each of the mean response times is $[\mu \pm 0.02\mu]$. The simulator was validated by comparing its steady-state performance results to analytical results for systems with closed-form solutions (more specifically, $M/M/m$ queues [11]), and also by comparing its results for particular job submission traces to observed results in the real DRMS system.

5.3 Performance comparison of the three policies

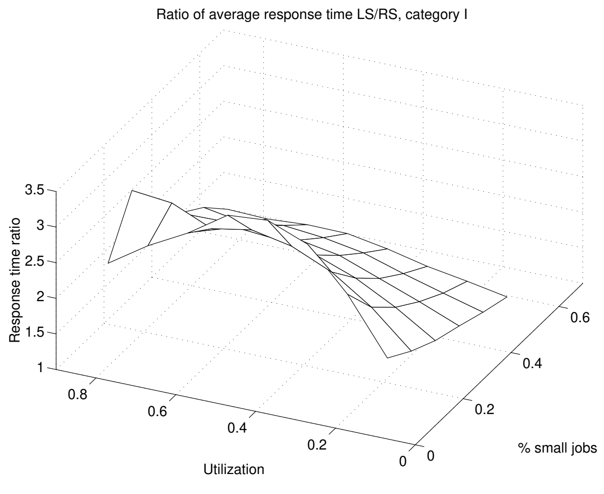
Table III: Workloads used in the experiments.

Workload	Time on min PEs		Time on max PEs	
	$\overline{T}$	C_T	$\overline{T}$	C_T
05:25:70	6753	1.04	1961	1.12
15:25:60	4208	1.39	1167	1.52
25:25:50	3056	1.58	830	1.73
35:25:40	2400	1.65	645	1.82
45:25:30	1975	1.62	527	1.79
55:25:20	1678	1.50	445	1.66

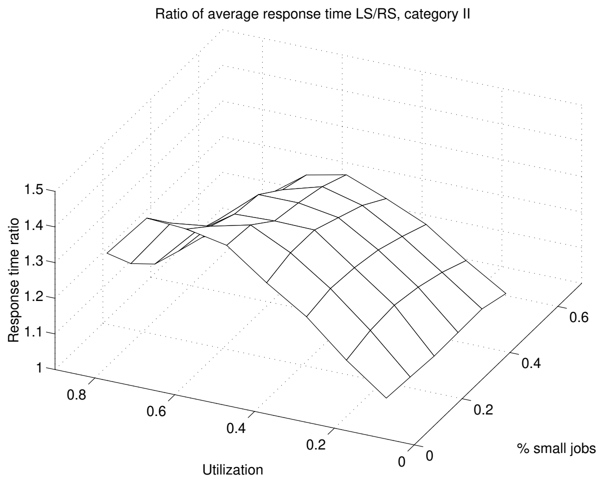
We first present simulation results comparing the average response times under the three scheduling policies. Let E[$R_{(p)}$] denote mean response time under policy p, $p \in \{\mathtt{AS}, \mathtt{LS}, \mathtt{RS}\}$, when all jobs are considered together. Similarly, let E[$R_{(p)}$, k] denote the mean response time under policy p, for jobs belonging to category k, $k \in$ {small, medium, large}. For the ease of comparison, we consider the ratios E[$R_{(\mathtt{AS})}$]/E[$R_{(\mathtt{RS})}$] and E[$R_{(\mathtt{LS})}$]/E[$R_{(\mathtt{RS})}$] over a range of system utilization values produced by a workload range. Thus, when the ratio E[$R_{(\mathtt{AS})}$]/E[$R_{(\mathtt{RS})}$] (or E[$R_{(\mathtt{LS})}$]/E[$R_{(\mathtt{RS})}$]) is greater than 1, RS policy outperforms AS (or LS) policy and vice versa when the ratio is less than 1. The utilization is defined as the job arrival rate times the ideal service time of the average job when executed on all processors. The ideal service time of a job when executed on all processors is determined as the execution time on a single processor divided by the total number of processors. Note that, because of the inefficiencies introduced by parallelism, the actual service times of the jobs we consider are higher. In our discussion, we refer to system utilization also as the load on the system or simply as system load. The workload we use for our experiments is defined by the proportion of the load produced by the jobs from small, medium, and large categories as listed in Table I. The jobs from each category are those listed in Table I. For our experiments, we used six workloads as shown in Table III. For each workload shown in that table, we list the mean service time and the coefficient of variation in the service time when jobs are run on the minimum and the maximum requested number of processors. A workload of 05:25:70 means that 5%, 25%, and 70% of the load are from category I, II, and III jobs, respectively. Within a category, all jobs arrive with the same rate.



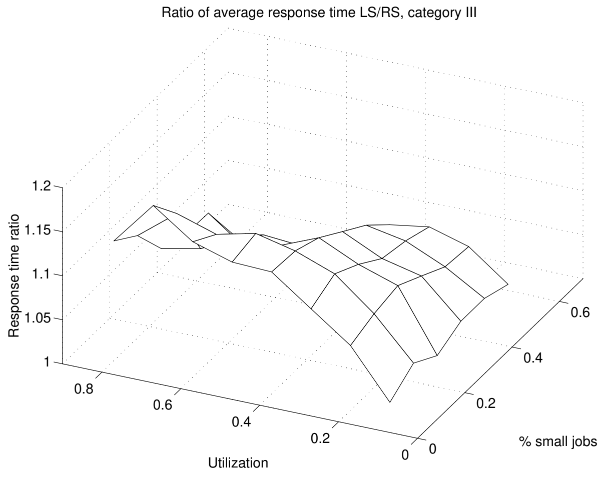
(a)



(b)



(c)



(d)

Figure 3: A comparison of response times under LS and RS policies under similar system utilization and workload conditions.

Shown in Figures 3(a) and 4(a) are the changes in the response time ratios E[$R_{(\mathtt{LS})}$]/E[$R_{(\mathtt{RS})}$] and E[$R_{(\mathtt{AS})}$]/E[$R_{(\mathtt{RS})}$], respectively, as the system load and the fraction of the system load contributed by small jobs are varied. As seen from the two figures, the system response time with RS policy is always better than that the other two policies, and between the AS and LS policies, the former performers relatively better. Notice that, for a given workload, the performance of RS policy first gets better with increasing load and then the advantage diminishes at high system loads. In the case of LS policy, the bulge is more towards the maximum system load we report (i.e., 0.8) whereas in the case of AS policy, the maximum advantage is somewhere in the middle of the system load range we consider. Similarly, in both cases, as the proportion of small jobs in the workload is increased, the advantage of RS policy initially increases and then it decreases. The response time ratios per category are shown in Figures 3(b) through 3(d) and in Figures 4(b) through 4(d). We observe from these plots that, even on per category basis, RS policy always outperforms the other two policies. Category I jobs benefit proportionately higher with RS policy than the jobs from other two categories. This is so because, under RS policy, any excess over a minimum number of processors allocated to medium and large jobs (i.e., category II and III jobs) can be taken away to schedule new jobs. This benefits the small jobs the most since they can be accommodated more often, thus reducing their queueing delays. Also, in our experiments, large and medium jobs could be expanded or shrunk in increments of 4 processors. Thus, when the system load is not towards the peak, small jobs can almost always get their maximum requested number of processors and thus, reduce their service time at the expense of an increase in service time of the jobs from other two categories.

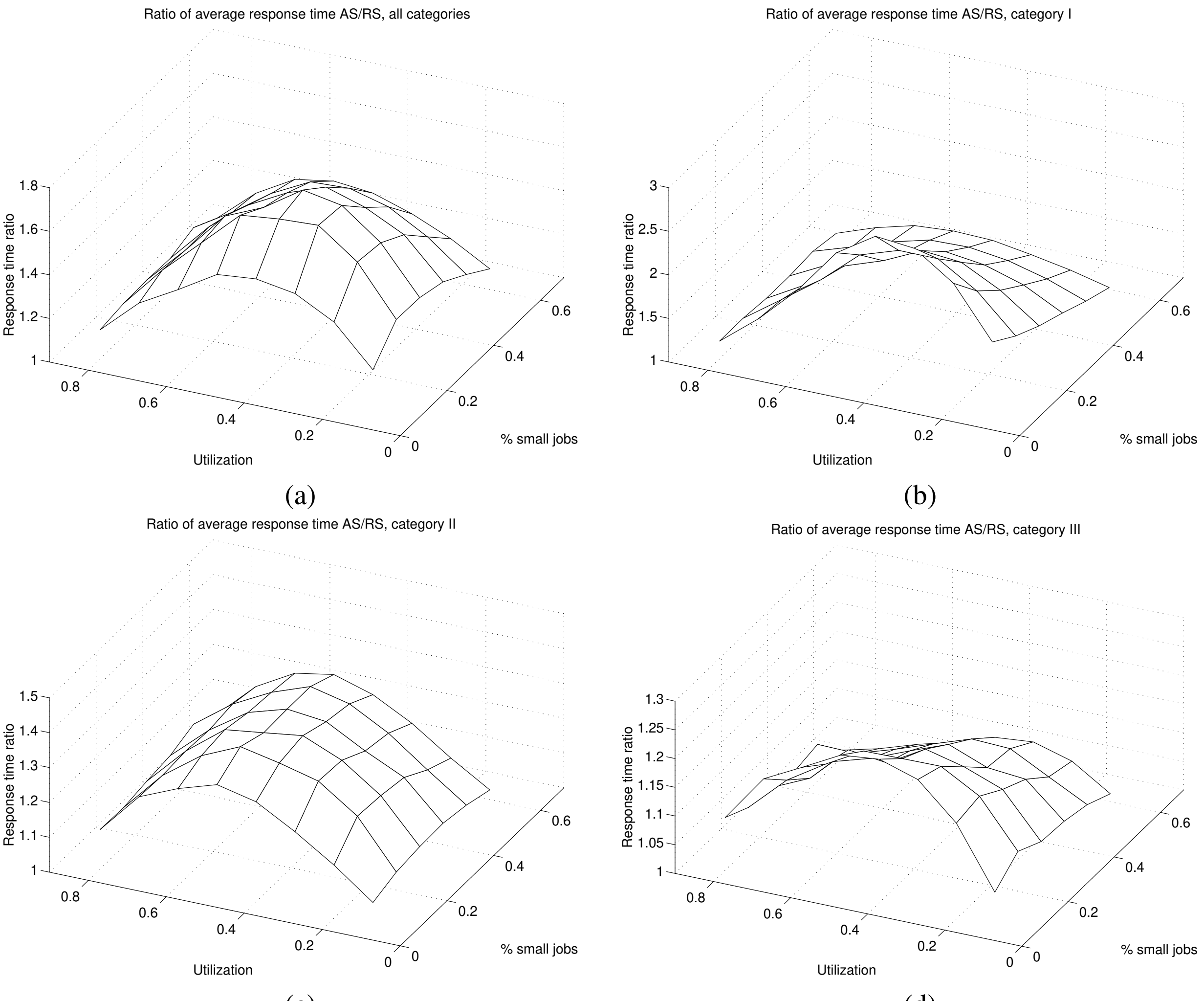


Figure 4: A comparison of response times under AS and RS policies under similar system utilization and workload conditions.

In Figures 3 and 4 we have presented response time ratios under various workload conditions. To get a better understanding of three policies in terms of the actual magnitudes of the response time, we plot the individual average response times for the three policies in Figures 5(a)–(d). For these experiments, we have used the workload of 25:25:50. We observe from these figures that jobs from all categories benefit with RS policy, but category I jobs benefit the most and the category III jobs the least. At high arrival rates, category I jobs benefit at the expense of category III jobs which experience an increased service time since they are either allocated a smaller partition size or are reconfigured to run on a smaller partition size whenever there are other jobs waiting to the scheduled. Notice also that AS policy performs significantly better than the LS policy when the performance of category I jobs alone is considered. For the other two categories, for the workload of 25:25:50, performance of AS is not significantly different than LS policy. Thus, much of the gains obtained with RS policy are because of its ability reconfigure running jobs.

To get a better understanding of the RS policy in terms of its sensitivity to various parameters we conducted additional experiments. Specifically, we consider the effects of reconfiguration costs, reconfiguration

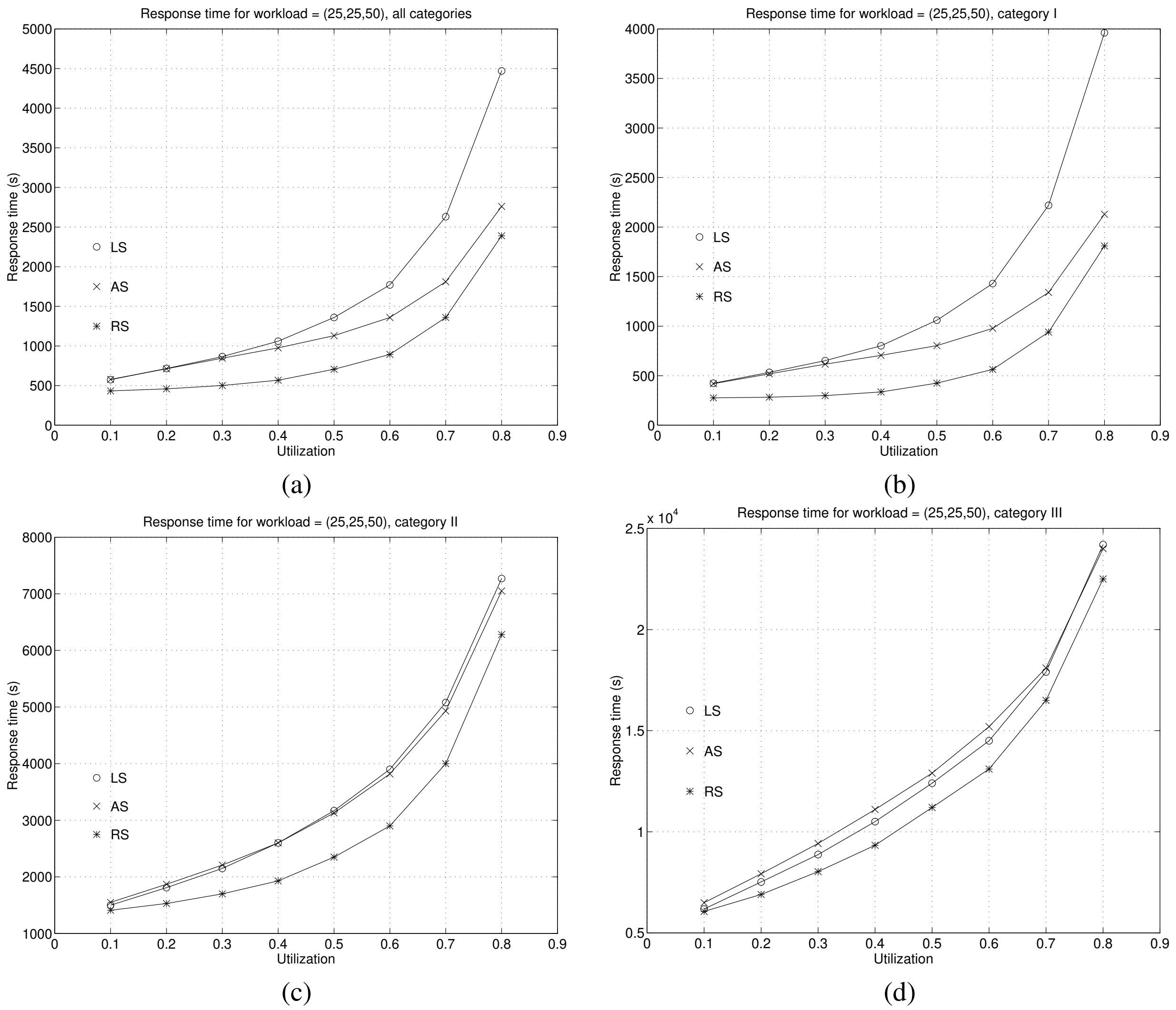
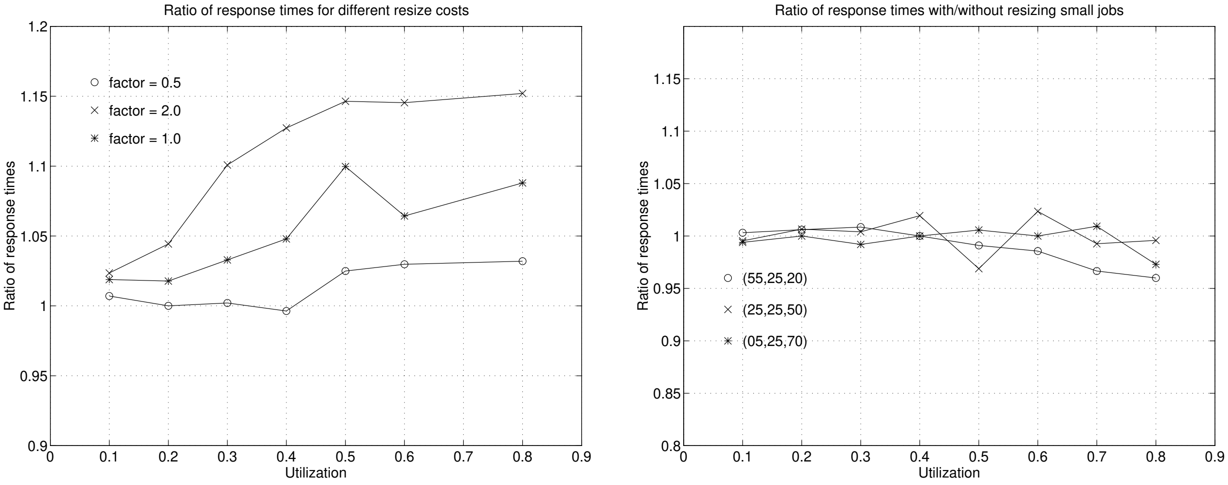


Figure 5: Absolute average response times with AS, LS, and RS policies under similar system utilization and for 25:25:50 workload

granularity, reconfiguring small jobs, and the frequency with which jobs can be reconfigured.

First we show the effect of reconfiguration costs on the average response time. These are shown in Figure 6(a), where we plot relative average response times under three different reconfiguration cost factors for the same workload. In that figure, the case with factor of 1 corresponds to the nominal reconfiguration costs described in Section 5.1; factor of 0.5 means that all reconfiguration costs are half of the nominal costs and a factor of 2 means double the nominal costs. We also determined the average response time under ideal case where all reconfiguration costs are zero (i.e., factor = 0). We normalize the average response times for the first three cases with respect to the ideal case and show the resulting response time ratios in Figure 6(a). In all cases, we used the workload comprising 25% of work from category I jobs, 25% of work from category II jobs, and 50% of work from category III jobs. Clearly, as the reconfiguration cost increases, the average response time also increases and it is higher at higher arrival rates. However, it should be noted that the response time is not too sensitive to the changes in the reconfiguration costs. With the nominal reconfiguration costs (factor = 1), the average response time is 10% worse than the ideal case and with double the nominal costs, the average response time gets at most 15% worse. In fact, if the costs are reduced by half, the effect of reconfiguration is almost negligible over a significant range of arrival rates.

In the experiments we have presented so far, small jobs are not subjected to any reconfiguration. We



(a) (b)

Figure 6: Effects of reconfiguration costs and of reconfiguring small jobs on the average response time under the `RS` policy.

now show that reconfiguring small jobs has negligible impact on the overall response time. For this we conducted experiments where small jobs are allowed to be reconfigured (within their processor range) in the same manner as the medium and large jobs and compared the average response time with that when small jobs are not subject to reconfigurations. We considered three workloads (5:25:70, 25:25:50, and 55:25:20) and arrival rates corresponding to 0.1 through 0.8 system utilization. We plot the corresponding response time ratios in Figure 6(b). Note that in all three cases the ratios are close to one. In fact, most of the variation is within the confidence interval of our experiments. Thus, reconfiguring small jobs has no impact on the performance of `RS`. On the other hand, not reconfiguring small jobs has the benefit of allowing non-resizable jobs (even sequential jobs) into the job stream.

Another parameter that affects the `RS` policy performance is the degree of flexibility offered by jobs in determining the number of processors to be allocated, initially as well as during a reconfiguration. We refer to this as the *allocation granularity*. Finer and more continuous the allocation granularity, the better control the scheduler has in tuning the system behavior to the current load. In the results we have presented so far, we used the allocation granularity for various job categories as shown in Table I (i.e., { 1, 2, 4 } for category I jobs, { 4, 8, 12, 16 } for category II jobs, and { 8, 12, 16, 20, 24, 28, 32 } for category III jobs). To study the effect of the allocation granularity, we experimented with jobs that have a coarser granularity of acceptable processor partition sizes. Specifically, we restricted category I jobs to { 1, 4 } processor range, category II jobs to { 4, 8, 16} processor range, and category III jobs to { 8, 16, 32 } processor range. We considered three workloads: 5:25:70, 25:25:50, and 55:25:20. All our results indicated negligible effect on the response time. We conducted another set of experiments where category III jobs had a processor range of { 16, 32 } instead of { 8, 16, 32 }. For the other two job categories, we maintained the same coarse granularity as above (i.e., { 1, 4 } and { 4, 8, 16 }, respectively). We used the same three workload as before (5:25:70, 25:25:50, and 55:25:10). From these set of experiments, we found that there was a significant impact on the average response time especially at higher arrival rates. The effect is not as significant if majority of the work is from category I jobs (i.e., small jobs). However, the performance deterioration can be prohibitively high at high loads consisting of larger proportions of work from category III jobs (e.g., workload of 5:25:70 at 0.7 to 0.8 system utilization).

A fourth parameter that can affect the performance of `RS` policy is the minimum time interval between two successive reconfigurations of the same job. We refer to this interval as the *resize interval factor*. In the results presented so far, after a reconfiguration, a job becomes eligible for another one only after a time interval equal to ten times its reconfiguration cost has elapsed; i.e., we use resize interval factor of ten. (Correspondingly, our applications are instrumented so that they check for reconfiguration requests every 5th iteration.) We note here that smaller the resize interval factor, quicker the scheduler is in adapting the processor allocation policy to the current demand on system. However, if a job is subjected to frequent reconfigurations, the overhead incurred increases its service time as well as the response time. Thus, there is a potential tradeoff. To quantify the effects of the reconfiguration costs on average response time, we conducted experiments where the resize interval factor was varied from 2 through 20 for the workload of 25:25:50. Results from our experiments indicate that that the average response time is not too sensitive to the changes in the resize interval factor. When the resize interval factor is varied from 2 to 20, the response time is affected at most by 5%. We note however that resize interval factor has an effect on the response time. This can be seen by considering the `AS` policy as the extreme case of `RS` policy where the resize interval factor is so large that no job is ever eligible for a reconfiguration. From our earlier results, we have seen the benefits of `RS` policy over the `AS` policy.

6 Related Work

Scheduling policies can be coarsely grouped into two large classes: *time-sharing* and *space-sharing*. In time-sharing policies, the machine is divided along the time axis and each job is given a set of time slices for its execution on all processors. In space-sharing policies, the machine is partitioned into groups of processors, and each partition is assigned exclusively to the execution of one job. In this paper, and in the first phase of our research, we have focused on space-sharing policies, such as those currently implemented by DRMS on the IBM SP2. Some systems, such as the SGI Challenge and Power Challenge multiprocessors [9], support a mix of time and space partitioning. It is our intention to extend our work to those kinds of systems. We refer the reader to [7] for a comprehensive survey in this area.

Many studies have shown that dynamic partitioning policies can alleviate the problem of adapting to workload changes, at the expense of additional reconfiguration overhead [23, 25, 13, 10, 15, 22, 16, 24]. In particular, [25, 13, 10, 15] analyze the benefits of dynamic partitioning on uniform-access, shared-memory systems and show that dynamic reconfiguration policies outperform all other space-sharing policies. In the realm of private-memory (message-passing) systems, [22] have also demonstrated that dynamic reconfiguration policies outperform the other policies. A discussion of the effects of different processor scheduling policies, and reconfiguration overhead, in dynamically reconfigurable systems can be found in [16, 14].

Our work differs from the previously mentioned research in that we have implemented a working environment for a commercial message-passing system (IBM SP2) that supports dynamic reconfiguration of processor partitions. We provide the language extensions and run-time services that allow user to easily port their existing SPMD applications to execute on reconfigurable partitions. We also provide all the resource control and scheduling mechanism to coordinate the execution of these jobs.

Extensive work has also been done on language support for data distribution in multiprocessors. Many of the data distributions in DRMS come for Fortran D [8] and HPF [12], but we have also added an arbitrary form of distribution to support sparse matrix operations. A similar extension to HPF is described in [18]. Different from HPF, our support for distributed data is in the context of dynamically reconfigurable, explicit SPMD parallel applications. Our language extensions incorporate mechanisms for implementing application oriented job scheduling. We also expose more of the details of computations and communications, with the goal of achieving better performance. We intend to, in the future, support the execution of HPF programs on reconfigurable partitions.

Simulation is a commonly used technique to evaluate the performance of complex systems, such as multiprocessors with dynamically reconfigurable partitions. It does, however, incur in high computation costs. Sometimes hours of simulation were necessary to generate a single one of our plots. This high computational cost of simulations, among other factors, has encouraged authors to develop analytical performance models of dynamically partitioned multiprocessors [14, 16, 24]. The available models, however, were not appropriate for evaluating the DRMS system at the level of detail we wanted.

7 Conclusions

We have described a comprehensive framework for application-assisted dynamic scheduling on multicomputer systems. The framework includes language extensions, a compiler, run-time system, resource coordinator and resource scheduler, and a network of job manipulators. This framework has been implemented in the form of the Distributed Resource Management System (DRMS) for the IBM SP2.

DRMS language extensions and library functions facilitate the task of designing a reconfigurable parallel applications. This is accomplished by organizing the application as a sequence of SPMD stages, each with its own resource and data distribution specifications. DRMS automatically takes care of all data rearrangements when the application moves from one stage to the next. The DRMS resource management and scheduling subsystem is capable of taking full advantage of reconfigurable applications by dynamically reconfiguring their processor partitions as the system conditions change. We have measured the performance of DRMS using reconfigurable parallel CFD applications and have determined the reconfiguration overhead experienced by an application to be in the range of 5–20s, even for the larger problem sizes.

We have used the performance parameters of DRMS and our application suite to perform a simulation study of reconfigurable *vs.* non-reconfigurable scheduling policies under typical scientific an engineering workloads. The results from our study indicate that the reconfigurable scheduling policy out performs the other policies as long as the job mix is such that there are sufficient job reconfiguration opportunities. However, not all jobs need to be reconfigurable. Thus, the `RS` policy is ideally suited for multiprocessor based supercomputing centers where the job mix has large variations with significant fraction of the cycles being consumed by large and long running jobs. We have shown that jobs from all categories benefit from a reconfigurable scheduling policy, even when only a fraction of all jobs are reconfigurable. Our experiments, in several instances, the average response time across all categories was reduced by a factor of two.

Our future work includes installing DRMS in a production environment where multiple users will submit and control their jobs through DRMS. We will also work on minimizing the overhead for application resizing. Our simulation studies show that we can improve the average response time of applications by an additional 10% by cutting down this overhead. Finally, we will conduct more performance evaluation studies of DRMS, including measurements of the system in production mode.

Acknowledgements: This work is partially supported by NASA under the HPCCPT-1 Cooperative Research Agreement No. NCC2-9000.

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] Bailey, D., Barszcz, E., Dagum, L., and Simon, H. NAS parallel benchmark results. *IEEE Parallel & Distributed Technology*, 1:43–51, 1993.

[3] Bailey, D., Barton, J., Lasinski, T., and Simon, H. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, 1991.

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

[5] Ekanadham, K., Moreira, J. E., and Naik, V. K. Application oriented resource management on large scale parallel systems. In Dharma P. Agrawal, editor, *Proceedings of the 1995 ICPP Workshop on Challenges for Parallel Processing*, pages 56–63, August 14 1995.

[6] Eknadham, K., Naik, V. K., and Squillante, M. S. PET: Parallel performance estimation tool. In *Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 826–831, 1995.

[7] Feitelson, D. G. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790, IBM Research Division, October 1994.

[8] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., and Wu, M. Fortran D language specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, December 1990.

[9] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Technical report, Silicon Graphics Computer Systems, 1994. Available from `http://www.sgi.com`.

[10] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[11] Jain, R. *The Art of Computer Systems Performance Evaluation*. John Wiley & Sons, New York, 1991.

[12] Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele Jr., G. L., and Zosel, M. E. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[13] Leutenneger, S. T and Vernon, M. K. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, May 1990.

[14] Madhukar, M., Leuze, M., and Dowdy, L. Petri net model of a dynamically partitioned multiprocessors system. In *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, pages 73–82, October 3-6 1995.

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

[16] McCann, C. and Zahorjan, J. 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.

[17] Moreira, J. E., Naik, V. K., and Konuru, R. B. A system for dynamic resource allocation and data distribution. Technical Report RC 20257, IBM Research Division, October 1995. Submitted to Journal of Parallel and Distributed Computing.

[18] Müller, A. and Rühl, R. Extending high performance fortran for the support of unstructured computations. In *Proceedings of the 1995 International Conference on Supercomputing*, pages 127–136, July 3-7 1995.

[19] Naik, V. K. Performance of NAS parallel application-benchmarks on IBM SP1. In *Proceedings of Scalable High Performance Computing Conference*, pages 121–128, 1994.

[20] Naik, V. K. Performance of NAS parallel benchmark LU on IBM SP systems. In N. Satofuka, J. Periaux, and A. Ecer, editors, *Parallel Computational Fluid Dynamics: New Algorithms and Applications*, pages 53–62. Elsevier, 1995.

[21] Naik, V. K. A scalable implementation of NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2):273–291, 1995.

[22] Naik, V. K., Setia, S. K., and Squillante, M. S. Processor allocation in multiprogrammed, distributed-memory parallel computer systems. Technical Report RC 20239, IBM Research Division, October 1995. Submitted to Journal of Parallel and Distributed Computing.

[23] Park, K.-H. and Dowdy, L. W. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18(2):91–120, 1989.

[24] Squillante, M. S. *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, chapter On the benefits and limitations of dynamic partitioning in parallel computer systems, pages 219–238. Springer-Verlag, 1995.

[25] Tucker, A. and Gupta, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.