Dynamic resource management on distributed systems using reconfigurable applications

by J. E. Moreira
V. K. Naik

Efficient management of distributed resources, under conditions of unpredictable and varying workload, requires enforcement of dynamic resource management policies. Execution of such policies requires a relatively fine-grain control over the resources allocated to jobs in the system. Although this is a difficult task using conventional job management and program execution models, reconfigurable applications can be used to make it viable. With reconfigurable applications, it is possible to dynamically change, during the course of program execution, the number of concurrently executing tasks of an application as well as the resources allocated. Thus, reconfigurable applications can adapt to internal changes in resource requirements and to external changes affecting available resources. In this paper, we discuss dynamic management of resources on distributed systems with the help of reconfigurable applications. We first characterize reconfigurable parallel applications. We then present a new programming model for reconfigurable applications and the Distributed Resource Management System (DRMS), an integrated environment for the design, development, execution, and resource scheduling of reconfigurable applications. Experiments were conducted to verify the functionality and performance of application reconfiguration under DRMS. A detailed breakdown of the costs in reconfiguration is presented with respect to several different applications. Our results indicate that application reconfiguration is effective under DRMS and can be beneficial in improving individual application performance as well as overall system performance. We observe a significant reduction in average job response time and an improvement in overall system utilization.

1. Introduction

Resource management is a much harder problem on parallel and distributed systems than on conventional

0018-8646/97/$5.00 © 1997 IBM

single-processor systems. The problem becomes even more difficult when the resource requirements of jobs arriving at the system are unpredictable and have high variability. Under such conditions, one obvious solution is dynamic management of resources; i.e., dynamically adjusting the allocation of resource as the demand on the system varies. For such a scheme to be effective, the resource management must adjust quickly in response to any changes in the system workload and/or in demands on the resources. It has been shown in the literature [1–8] that dynamic resource management results in improved job and system performance in comparison to more conventional static resource management strategies, where resources are statically allocated and a job is treated as a single resource scheduling unit during its lifetime.

In this paper we discuss a particular form of dynamic resource management based on reconfigurable applications. We define application reconfiguration as the activity of statically or dynamically changing the degree of parallelism exposed by an application to the external environment. (Other common uses of the term *application reconfiguration* are briefly discussed in Section 10.) The applications we consider express parallelism in the form of concurrent execution of multiple tasks as opposed to, e.g., parallelism in pipelined execution. (Throughout the paper we use the term *task* to mean a thread of execution which, depending on the particular system, could be implemented as a full process or as a lightweight thread.) These multiple tasks and their corresponding data are then mapped onto underlying physical resources, manifested primarily in the form of processors. During an application reconfiguration, the number of concurrently executing tasks of an application may be modified, and the tasks and associated data may be remapped onto the physical resources. This in turn allows the system resource manager to alter the allocation of physical resources to jobs running in the system in a dynamic and adaptive manner. Not all parallel applications are amenable to reconfiguration, and part of this paper is devoted to characterizing those that are.

To make dynamic resource management a reality, three requirements must be met: Applications must be reconfigurable, a run-time mechanism for application reconfiguration must be available, and a suitable run-time infrastructure must be in place. We have developed the Distributed Resource Management System (DRMS), an integrated environment for design, development, execution, and resource scheduling of reconfigurable applications. DRMS aims at delivering better system and job performance through dynamic resource management using reconfigurable applications. DRMS-reconfigurable applications follow an extended SPMD (*single-program multiple-data*) model that we call the SOP (*schedulable and observable point*) model. We explain this model briefly later in this paper (Section 5). We note, however, that the concepts are applicable to many other types of programming models.

Dynamic resource management is beneficial compared to its static counterparts only if the overhead of dynamic changes to the resource pools is not excessively high. In this paper, using extensive performance evaluation of application reconfigurations, we present a detailed breakdown of the costs of reconfiguration. This analysis gives an insight into the strengths and limitations of our approach and, most significantly, directs our work in improving the overall design. We also conduct system-level performance studies which show that job-scheduling policies using dynamic resource management based on reconfigurable applications deliver better system performance than policies that use static resource management.

Summarizing, the goals of this paper are fourfold: 1) to characterize reconfigurable parallel applications; 2) to describe the environment (DRMS) that we have developed for dynamic resource management using reconfigurable applications; 3) to evaluate the performance of application reconfiguration when it is used to support dynamic resource management; and 4) to demonstrate system-level performance improvements as a result of dynamic resource management.

We note here that all performance data presented in this paper are from experiments carried out on the IBM RS/6000* SP system. However, the design of DRMS and the programming model and run-time environment for reconfigurable applications described in this paper are equally applicable to other parallel and distributed systems.

This paper is organized as follows. Section 2 presents the motivation for considering application reconfiguration in the context of this paper. Section 3 characterizes the class of applications that are amenable to dynamic reconfiguration with no algorithmic changes, and it also states the rules for reconfiguration that must be obeyed for consistency and correctness of the application. Several mechanisms that have been proposed to accomplish program reconfiguration are discussed in Section 4. The specific programming model for reconfiguration in DRMS is presented in Section 5. An overview of the DRMS framework is given in Section 6, while Section 7 discusses the development and execution of reconfigurable applications under DRMS. Application reconfiguration performance studies are described in Section 8, and system-wide performance results are presented in Section 9. Section 10 discusses some additional (to Section 4) related work. Finally, Section 11 presents the conclusions of this paper and discusses some future work.

2. Motivation

Applications can take different approaches to exploiting parallelism: multiple but identical tasks operating on multiple data; multiple and distinct tasks operating on the same data; or some combination of the two. The benefits of parallelism are realized by simultaneously carrying out execution of multiple tasks. On the basis of the number of simultaneously executing tasks, we can classify applications into three categories: 1) applications where input data and problem "boundary conditions" determine the possible number (or set of numbers) of executing tasks; 2) applications that are always executed with a fixed number of tasks, independent of the input data or the underlying system configuration; 3) applications where the number of executing tasks is not fixed by the problem or by the input data set, but is determined by some other application- or system-specific parameters (e.g., nondeterministic algorithms that may follow one of many execution paths, randomly).

From the point of view of scalable parallel computing, Category 1 above is of interest. Ideally, each unit of computation could give rise to a task which could follow data, as with the dataflow model of computation. Architectural limitations and other performance considerations require that many computational units be merged and that several distinct data items be agglomerated. In the case of scalable parallel computing, this has given rise to heavyweight computing tasks and large amounts of data agglomeration. Further, performance considerations have resulted in implementations with tight coupling among the executing tasks, data, and execution units. While such applications are quite common in practice, they tend to become monolithic and vertically integrated. The number of tasks is determined almost at the beginning of a program execution, on the basis of the data set and/or the execution units available at the time. Under such a model of monolithic vertical splitting of a program, all tasks must continue to exist throughout the program execution. For perfectly parallel programs, this model works out well. However, most programs dealing with realistic problems are far from perfect, and any monolithic splitting necessarily leads to imbalance among the tasks and/or awkward programming to bring about the necessary balance. As a consequence, programs are unable to use physical system resources in a judicious manner. A further consequence is that complex (and sometimes ugly) resource scheduling strategies are developed when the inflexibilities of the monolithic model are taken for granted.

To make parallel programs more flexible and less monolithic, it is necessary to provide a model that does away with vertical agglomeration of data as well as the strong coupling between data and tasks. To make such a model practical, the performance costs of such flexibility should be relatively small.

3. Reconfigurable parallel applications

In this section, we explain in more precise terms the reconfigurable application domain that we consider. We give some definitions of reconfigurable applications, list the conditions for reconfigurability, and list the criteria for correctness and consistency of applications across reconfigurations.

- *Definitions*

An application that is associated with only a fixed number of executing tasks, independent of the input dataset or of the external environment, is a *statically configured* parallel application. An application for which the number of executing tasks depends on the input dataset is a *dynamically configurable* application. Similarly, if an application can execute with more than one set of tasks (because of some change in the environment external to the application) for the same input dataset, that application is also said to be dynamically configurable. An application is *reconfigurable* if it executes in two or more stages such that in the first stage it executes with one set of tasks and in each subsequent stage it executes with a different set of tasks. The application is said to be *statically reconfigurable* if this behavior is input- and time-invariant. The same application is said to be *dynamically reconfigurable* if the number of tasks in each stage depends either on the input or on some parameter external to the program. In this paper, we are primarily concerned with dynamically reconfigurable applications. For brevity, in the rest of the paper, we refer to these as reconfigurable applications.

- *Conditions for reconfigurability*

A necessary condition for an application to be reconfigurable is that the application must be capable of exposing multiple degrees of parallelism; that is, it should be possible to execute the application to completion with more than one set of tasks and, in each case, the final outcome should not deviate significantly from the expected outcome (e.g., some round-off differences might be acceptable). The multiple degrees of parallelism should be exhibited with the same input dataset. In addition, reconfigurable parallel applications should possess one or more of the following properties:

- Capability of changing the number of (active) tasks during the course of program execution.
- Capability of changing the mapping of tasks to execution units.
- Capability of changing the affinity between data components and tasks at run time.

- Capability of switching from one type of data agglomeration to another type during the course of a program execution.

Having one of the above properties alone does not qualify an application to be (dynamically) reconfigurable. For example, consider an application that is invariant in the number of tasks for a given input but allows for dynamic scheduling of the tasks on one or more execution units. Such an application is not reconfigurable because it does not possess the necessary condition of being able to expose multiple levels of parallelism.

The above discussion characterizes a reconfigurable application. Note that the programming model is not part of this characterization. Parallel applications using the fork-join model or the workers model, or those belonging to the SPMD model, can all be made reconfigurable. In all cases, reconfiguration must satisfy the correctness and consistency criteria that we describe next.

- *Correctness and consistency issues*

When a program undergoes reconfiguration, there is a change in the state of the program. Such a change in state should not lead to changes in the semantics associated with the program just before the reconfiguration point (RP), nor should it lead to unexpected program behavior after the RP. The former requirement leads to correctness issues and the latter to consistency issues; the two are interrelated. Unexpected program behavior includes any of the following: 1) the program does not produce the same results as it would have if there had been no reconfiguration; 2) the program does not terminate at the same point it would have if there had been no reconfiguration; 3) the program does not interact with the external environment as expected.

In general, reconfigurations must obey the following rules:

1. The application semantics should not change across reconfiguration points (RP).
2. All data that defined the program state before an RP should be preserved and defined across the RP.
3. If any task is terminated at an RP, all unfinished computations associated with that task should be assigned to one or more tasks existing after the RP.
4. If there is algorithmic change following a reconfiguration, the old algorithm(s) should be replaced by "equivalent" new algorithms such that expected program behavior is maintained.
5. Reconfiguration should not cause a deadlock or a livelock as a result of a change in the degree of parallelism.

Reconfigurable applications that obey the above rules are said to be valid reconfigurable applications. Note that the rules specify only the general guidelines, and testing the validity of a reconfigurable application requires detailed knowledge of the application. For this reason, it is important that the abstractions provided by the programming model for developing reconfigurable applications should be simple and easy to test for validity by the program developer.

4. Mechanisms for reconfiguration of parallel applications

Two important issues in successfully designing and developing reconfigurable parallel applications are programming abstractions and efficient run-time mechanisms for reconfigurations. Programming abstractions should be easy to use, motivating users to develop reconfigurable applications. These abstractions should preferably be *additive*, that is, constructs that can be added to an already correct (but not reconfigurable) program. The run-time support for reconfigurations should be such that the disturbance to the application is minimal (in terms of performance) and it should leave the application in a consistent state after a reconfiguration.

We describe some of the mechanism to support reconfigurations that have been proposed in literature for various programming models. The programming models we consider are the following:

- *Workers model:* Work for helper tasks is dynamically carved out.
- *Fork-join model:* Dynamic spawning of tasks and dynamic data agglomeration based on the amount of work or available resources.
- *HPF model:* Parallel execution from a single-threaded specification.
- *AMP model:* Fixed number of executing tasks, which swap data and computations.
- *SOP model:* The model on which DRMS is based, discussed in more detail in the next section.

In the workers model, a (logically shared) global entity defines the tasks that must be executed and the data on which they operate. This global entity can be active (a master task) or passive (a global state pool). Worker tasks are given or fetch tasks from this global entity, execute them, and return the results to the entity. The Piranha system [9] is an example of a mechanism to support reconfigurable applications developed according to the Linda [10] workers model. The tuple space in Linda performs the role of the global state pool. Piranha was developed mainly to harness the idle cycles in networks of workstations. Because usage patterns of such networks are

highly unpredictable, the ability to move computations around dynamically is very important. While [9] reports many successful uses of Piranha, it also reports that some data-parallel programs (specifically, LU decomposition) do not reconfigure efficiently in that environment.

The fork-join model is related to the workers model. This model usually employs a *two-level* scheduling mechanism: First, a number (possibly variable) of kernel-level threads are scheduled for execution on physical processors; these kernel threads are then used as virtual processors for the execution of user-level threads that exploit the parallelism. The user-level threads are created to execute tasks from a shared task queue. This two-level scheduling is used because forking and joining of threads can be done much faster for user-level threads than for kernel-level threads. There are also implementations with a single level of scheduling. Examples of systems that use the fork-join model to support reconfiguration are Cray Multitasking [11], Process Control [12], and Minos [5]. Autoscheduling [13, 14] has shown how an efficient fork-join model can support macro-dataflow execution on time-variant processor partitions. The work on fork-join models mentioned above is all in the context of shared-memory multiprocessors, which eliminates the need for dynamically changing the binding between data spaces and tasks (there is only one, shared, data space). The work described in [7], for private-memory (message-passing) multiprocessors, does not fall in our classification of reconfigurable mechanisms because the number of executing tasks is kept fixed throughout the lifetime of an application.

HPF [15] programs are created by adding data distribution annotations to single-threaded code with array operations. Although the language was not specifically designed for developing reconfigurable applications, it is possible to exert some control on the number of tasks performing computations. HPF allows the specification of virtual processor grids in the declaration section of each scoping unit (function or subroutine) and the distribution of data onto these processor grids. Therefore, it is possible, in principle, to write each scoping unit to execute on a different number of processors. However, note that there are several fundamental and implementation difficulties with this approach. First, `COMMON` variables must have the same distribution specification in all scoping units that use them, and therefore are not amenable to reconfiguration. Also, HPF provides no constructs for direct specification of the number of desired tasks. Therefore, the scheduling environment of the system where an HPF application is running would have to derive such information from the executing program itself, which may be very difficult.

The Adaptive Multiblock PARTI (AMP) library [16] supports reconfiguration of SPMD programs within a fixed number of executing tasks. An application is spawned on the maximum number of tasks on which it can run. During its execution, data redistribution is used to move the active data to different subsets of tasks. Only the tasks belonging to this subset, at each moment, are *active* and perform computation. The other tasks execute the code for the application but, since they have no data associated, they do not perform intensive computations and are called *skeleton* tasks. This approach has some deficiencies: First, it imposes a hard limit on the maximum number of active tasks, namely the number of tasks that were originally spawned; the skeleton tasks can interfere with other applications that have active tasks in the same physical processor, thus creating a performance and scalability problem; finally, it requires that the underlying system be able to support *time-sharing* among many parallel applications on the same processors (one active task from one application, skeleton tasks from the other applications), which is not always supported.

5. SOP programming model

The reconfiguration mechanism of DRMS is based on the SOP programming model that we describe in this section. In this programming model, the execution of a parallel program consists of a sequence of stages we call *schedulable and observable quanta* (SOQs). The number of tasks is fixed during an entire stage; also, the association between data spaces and tasks is fixed and one-to-one. Therefore, each stage behaves like a conventional SPMD program, allowing us to exploit the performance benefits of the monolithic model during the execution of a stage.

Boundaries between stages are defined by *schedulable and observable points* (SOPs). At an SOP, and only at an SOP, the state of the parallel program can be examined and modified. At this point the number of executing tasks and the association between tasks and data spaces can be altered. When this happens, we say that the SOP is a *reconfiguration point*. The stage following the reconfiguration point executes on the new configuration of tasks and data until it reaches a new SOP and a new stage begins. A reconfiguration from one stage to the next can involve a change in the number of tasks, a change in the association of data with tasks, or both.

We define a stage as consisting of four sections: *resource*, *data*, *control*, and *computation*. The *resource section* specifies the number of tasks needed for the execution of the stage. This specification can be in the form of a range of valid numbers of tasks, often dependent on the problem size and other problem-specific parameters. Once a specific number of tasks is selected for execution of the stage, the *data section* specifies an association between the data space and the tasks. We discuss this association in more detail below. The *control section* specifies values for control variables pertinent to the stage. Control variables are used to control the flow of

execution inside a stage, which may vary depending on the number of tasks and data association. Finally, the *computation section* specifies the computations and communications that each task performs for the execution of the stage. These computations and communications are usually steered by the control variables specified in the *control section*.

Let the execution of a parallel job $\mathcal{J}$ consist of $n(\mathcal{J})$ stages. Each stage i is executed with $t(i)$ tasks. Each task $j, j = 1, \cdots, t(i)$, has its own data space. We denote by $d(i, j)$ the data space of task j in stage i. The data space of a task can be divided into two parts: a private data space $d_p(i, j)$ and a shared data space $d_s(i, j)$. The private data space contains temporary data and flow control variables that are pertinent only to this task at this stage. The shared data space contains actual problem data carried from stage to stage. Because of replication of some data, the shared data space of different tasks may actually overlap. The union of the shared data spaces of individual tasks forms the global data space for a given stage:

$$D_{\mathcal{J}}(i) = \bigcup_{j=1}^{t(i)} d_s(i, j). \tag{1}$$

$D_{\mathcal{J}}(i)$ is the collection of *global* data structures of job $\mathcal{J}$ during stage i. If the global data structures of a program are purely static, $D_{\mathcal{J}}$ is an execution-time invariant:

$$D_{\mathcal{J}}(i) = D_{\mathcal{J}}(i-1) = D_{\mathcal{J}}(i+1) = D_{\mathcal{J}} \qquad \forall i. \tag{2}$$

In the general case, dynamically allocated global variables can cause a change in $D_{\mathcal{J}}$ from stage to stage. The global data space $D_{\mathcal{J}}$ is the state that is preserved across a reconfiguration point (additional global variables can be created by the new stage *after* the reconfiguration). The *data section* of a stage specifies how this data space is to be decomposed into the task data spaces $d_s(i, j)$.

6. DRMS framework

DRMS consists of a general framework for the execution of reconfigurable applications and for the dynamic management of resources used by these applications. In the DRMS approach, these two concepts are tightly coupled. Efficient resource management is accomplished because applications can be reconfigured to respond to internal and external (environment) changes. Conversely, applications can reconfigure efficiently because resources are being continuously granted to and reacquired from applications. In this section, we focus on the concepts of DRMS, and in the next section we discuss some specifics of our implementation.

- *Reconfiguration mechanism*

DRMS supports the reconfiguration of applications that follow the SOP programming model. As discussed in Section 5, these applications consist of a sequence of stages (SOQs). Each stage is executed with a constant number of tasks, and there is a fixed association between tasks and data space, just as there is in a regular SPMD program. SOPs mark the boundaries of stages, and at SOPs the number of tasks and/or association of data can change.

When the number of tasks is changed at an SOP, the global data space is preserved across the reconfiguration. (In the next section, we describe the mechanisms used in DRMS for this purpose.) All tasks emerging from the SOP inherit the private data space, but all variables in the private data space of a task are left in an undefined state. These must be reinitialized before being used. Also, the decomposition of the global space into the shared data space of the tasks is undefined. Therefore, a new association between data space and tasks must be specified before the global data are used.

To make the abstractions described above more practical and concise, DRMS offers a set of FORTRAN language extensions for the declaration of SOPs and the specification of stages. These language extensions are used in each stage to declare the desired number of tasks (*resource section*) and the decomposition of the global data space into the shared data space of the tasks (*data section*). The language extensions are processed by a compiler that translates them and links the program with a run-time system, thus creating a reconfigurable application.

The application is closely coupled with a run-time system and the application global data space and execution state are exposed to this run-time system. When the application is at an SOP and when there is a change in the number of tasks, the run-time system suspends the application until the new set of tasks is created, and then continues execution on this new set of tasks. The run-time system also rearranges data as necessary according to the new association between data and tasks. The run-time system, because of its knowledge of the global data space, can provide a set of utility functions that offer convenient abstractions for writing the *control* and *computation sections* of a stage.

Reconfigurable applications run under an environment provided by DRMS. This environment consists of a scheduling module that has direct control over the physical processors of the parallel system. For each parallel job, this scheduling module creates a partition of processors that executes the job. This partition can be resized at the SOPs of that application, but it is fixed for the duration of an application stage. The scheduler also decides on the mapping of application tasks to processors

in the partition. In our current implementation of DRMS for the RS/6000 SP platform, for performance reasons, only one task is mapped onto a processor. This maintains a one-to-one correspondence between application tasks and execution units. Thus, changes in the number of tasks of an executing application cause a corresponding change in the size of the processor partition executing the application. Although not available in our current implementation, in principle the processor partitions of multiple jobs may overlap with one another. When that happens, a physical processor time-multiplexes the execution of the tasks belonging to multiple applications.

• *Creating reconfigurable applications*

There are many alternatives for generating a reconfigurable application according to the SOP model. In this paper we focus on three approaches that are important in the context of scientific and technical applications.

The first approach starts from a conventional SPMD program that executes with a set of tasks that is fixed at the beginning of the application execution. That is, the number of executing tasks can be set only once, at job start time. To make such a program reconfigurable, the programmer manually identifies the segments of the program that are to be transformed into stages. DRMS language extensions can be used to specify the SOPs and declare, for each stage, the task requirements and the association between data and tasks. The programmer adds code to reinitialize local variables and compute control variables that steer the computation.

The second approach also starts from a conventional SPMD program, but the transformation into a reconfigurable program is performed automatically by a compiler. In the general case, the user may still have to specify the decomposition of the global data space, since that can be arbitrarily complex and very difficult for the compiler to determine. The user can also specify the number of tasks (perhaps a range of numbers) necessary for the execution of the program. The compiler can use control-flow and dataflow analysis to identify points in the program where an SOP can be inserted. The basic requirement is that all tasks must be able to synchronize at the point. The compiler then identifies those variables that must be preserved across the reconfiguration and makes them part of the global data space. Using the language extensions provided by DRMS, the specifications for number of tasks and data decomposition are automatically inserted at the SOP by the compiler, according to the original specifications given by the user. The compiler must also identify which variables are control variables that depend on the number of tasks and/or data association (for example, loop bounds) and create the proper initialization code for them.

Finally, the third approach consists of creating reconfigurable applications from HPF source programs. HPF programs already contain the specification for data decomposition, so the user does not have to add any information. HPF compilers usually translate HPF source code into SPMD code. We can then perform the same analysis as in the second alternative but helped by the extra semantic information available from the HPF source. Also, because the SPMD code is generated automatically from a single-threaded specification, its behavior is more constrained than that of an SPMD code written directly by the user. We note that the ability to continue execution on a different set of tasks and to redistribute the data to accommodate this new set is not provided by the `REDISTRIBUTE` construct available in the standard HPF model.

• *DRMS architectural overview*

Figure 1 shows the main functional components of DRMS and the primary interactions among these components. The DRMS compiler translates programs with DRMS annotations, linking them with a run-time system (RTS) to create reconfigurable executables. The functional components that perform the resource coordination and task scheduling are the resource coordinator (RC) and the job scheduler and analyzer (JSA). Run-time management and coordination of user applications are accomplished by the user interface coordinator (UIC), the RC, and the task coordinator and run-time monitor (TC). The performance analysis component is handled by a run-time performance data gatherer and the associated tools and utilities.

The system-level allocation and scheduling decisions are made by the JSA on the basis of the implemented scheduling policies. These decisions may take into account information such as application-supplied resource requests, job priorities, and individual processor utilization, as well as system-level information such as current and expected workload. A particular implementation of the 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 interface directly with the user or user job, but rather it communicates its decisions to the RC, which interacts with the UIC and the TC. There is only one logical RC and one logical JSA for the entire DRMS.

Each user application has an associated TC which consists of multiple agents, one per application task (therefore, this set of TC agents can vary during application execution). One of the agents acts as a master for coordination with the external world, including the RC and the UIC. We call this agent the *master* TC. The run-time interactions between the user application and the rest

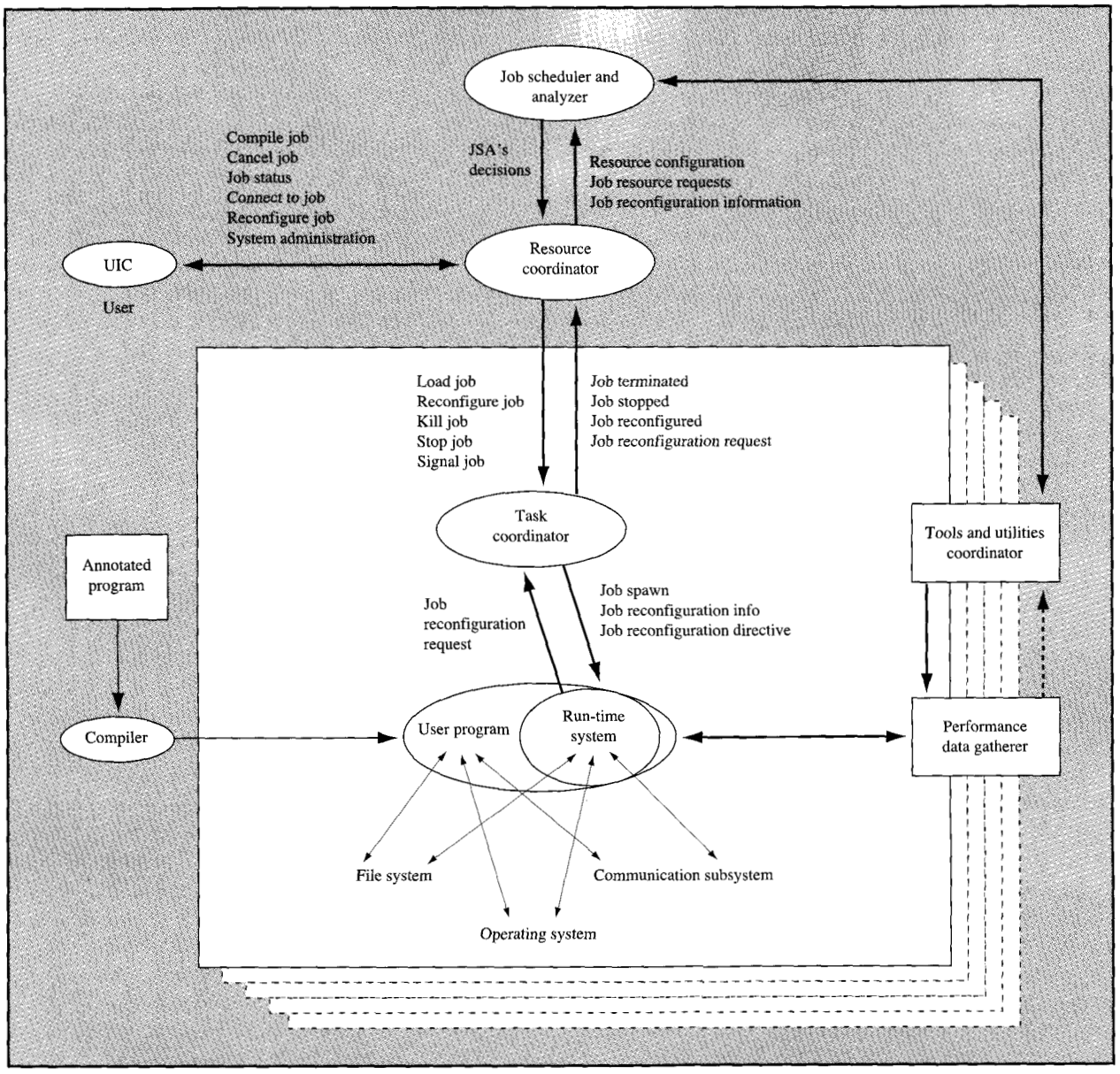

Figure 1

DRMS architecture.

of the system, including other applications, user, and RC, are managed by various subcomponents of the TC. The main functions carried out by the TC are acquiring/releasing processors from/to the RC and starting/restarting application task execution on the allocated processors.

The user submits jobs and interacts with the system throughout the course of the job execution via the UIC. The primary function of the UIC is to provide a convenient user interface. The performance-gathering component is designed to assist users and the system administrators to understand some of the characteristics of user jobs [17]. It can also provide performance feedback that the JSA can use in making more intelligent scheduling decisions.

A reconfigurable job interacts with the resource management system only at its SOPs. 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 the RC. The 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 job can run with a new set of tasks on a new set of processors. This asynchronous approach to application reconfiguration allows an application to execute an SOP at a very low cost, since only a local test, as opposed to a whole communication protocol with the JSA, has to be performed at each SOP. It relies on the fact that the JSA is constantly aware of the resource requirements of the applications. Since these requirements can change during the course of a computation, the JSA must be kept updated. An application can also voluntarily initiate an expansion or shrinkage of its allocated set of processors. This happens whenever the current number of tasks an application is executing does not belong to the valid set of tasks specified for a stage. In this case, the application must wait, possibly in a suspended state, until the JSA can allocate the necessary number of processors.

7. Application reconfiguration under DRMS

In this section, we discuss some specifics of our current implementation of DRMS. We illustrate the use of DRMS language extensions to FORTRAN that allow the development of reconfigurable applications. We also describe the functions performed by the compiler and the run-time system in order to produce a reconfigurable application. We describe the detailed steps involved in a specific reconfiguration operation, and we illustrate the inner workings of the system using a hypothetical reconfiguration scenario.

• *Writing reconfigurable FORTRAN applications*

DRMS language extensions to FORTRAN are in the form of *annotations*, source-level comments that are ignored by a regular FORTRAN compiler. They always start with the `!$DRMS$` sequence. The DRMS compiler translates these annotations into executable FORTRAN code in a preprocessing phase. The output of the preprocessors is then compiled by a regular FORTRAN compiler. We illustrate our language support by means of an example. **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. At each iteration k, a new value $u_k(i, j)$ for grid point (i, j) is computed as

$$u_k(i,j) = h + \tfrac{1}{4}[u_{k-1}(i-1,j) + u_{k-1}(i+1,j) + u_{k-1}(i,j-1) + u_{k-1}(i,j+1)]. \quad (3)$$

Because of the nearest-neighbor communication, the natural approach is to partition the grid into contiguous sections and let each task operate in one section. Because of the boundaries between sections, it is also convenient to let the section belonging to one task overlap with the sections from its neighbors.

The application in Figure 2 corresponds to the code executed by each task participating in the computation. The declaration of `u0` and `u1` at the top of the program represents the declaration of the *local sections* of those variables (i.e., local to a task). The distributed arrays `u0` and `u1` are declared as having global shape `n` × `n` through the DRMS `DIMENSION` annotation. The local shape for the section in each task is automatically computed by DRMS. DRMS also allocates the appropriate local storage and associates it with `u0` and `u1`.

The first instruction in the body of the iteration (`RESIZE`) marks an SOP. The `RESIZE` annotation specifies the number of tasks for the execution of the stage. It implements the *resource section* of the stage. In this example, it is specifying that the set of valid numbers of tasks for execution is {4, 8, 16, 32}. DRMS supports the declaration of a variety of regular ranges of tasks, as well as declaration of any irregular range in the form of a list of valid values. After this instruction, execution continues on one of the valid numbers of tasks, possibly different from the previous iteration.

The next annotation implements the *data section* of the stage. It defines how the global, `n` × `n`, arrays `u0` and `u1` should be decomposed into local sections for each task. A two-dimensional, HPF-like `BLOCK` distribution is used, and overlap among the sections is specified with the `BORDERS` declaration. DRMS supports all forms of data distribution from HPF: block, cyclic, block-cyclic, and collapsed. It also supports a different form of block distribution, that we call `BLOCKD`, and two forms of irregular distribution: block-list and arbitrary (specially useful for manipulating sparse matrices). We refer the reader to [18] for a more complete reference on the various annotations and data distributions used in DRMS, and for a description of our data redistribution algorithms.

Two calls to the DRMS utility function `drms_local_extent( )` are made to initialize the control variables `xsize` and `ysize` with the extents of the local sections of each processor. This implements the *control section* of the stage.

The remainder of the iteration body implements the *computation section* of the stage. Control variables `xsize` and `ysize` are used as bounds in the loops that operate on the local sections of arrays `u0` and `u1`. Because

```
      REAL u0, u1
!$DRMS$ ARRAY :: u0(:,:), u1(:,:)
!$DRMS$ DIMENSION(n,n) :: u0, u1
      DO i = 1, m
!$DRMS$   RESIZE 4:32:*2
!$DRMS$   DISTRIBUTE(BLOCK,BLOCK), BORDERS((1,1),(1,1)) :: u0, u1
         xsize = drms_local_extent(u0,1)
         ysize = drms_local_extent(u0,2)
         IF (MOD(i,2) .eq. 0) THEN
            CALL drms_update_borders(u1)
            DO j = 2, ysize-1
               DO i = 2, xsize-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, xsize-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 reconfigurable Poisson solver.

overlap regions must be updated with data from the actual owners, a call to the DRMS utility function `drms_update_borders( )`, which automatically updates these regions, is used before the computation proper.

• *DRMS compiler and run-time system*

The DRMS compiler translates FORTRAN programs augmented with DRMS annotations into executable, reconfigurable applications. The first step in compiling is a preprocessing of the annotated program by the DRMS annotations preprocessor (DAP). The DAP is a source-to-source translator that generates output in FORTRAN. The output from the DAP can then be compiled by a native FORTRAN compiler and linked with the DRMS run-time system (RTS) to generate an executable. The DAP performs two primary functions: It translates the DRMS annotations into calls to the RTS that performs the specified action, and it creates "handles" for the local sections of the distributed arrays to allow the RTS to manipulate and reshape these sections.

A `RESIZE` annotation is translated into a call to the RTS communicating the specified range of tasks. The RTS then interfaces with the TC to obtain the appropriate number of tasks and to continue application execution on the new set of tasks. Data distribution annotations are translated into calls that build descriptors of the declared distributed arrays. The RTS uses these descriptors to compare old and new distributions, redistributing data accordingly. Using the descriptors for the distributed data, the RTS can automatically compute the local sections of each task. DRMS-created local sections always use *dense storage*; i.e., there is no gap between consecutive array columns. The descriptors also allow the RTS to provide a series of utility functions, as seen in the example of Figure 2, that facilitate parallel programming in general and the development of reconfigurable applications in particular.

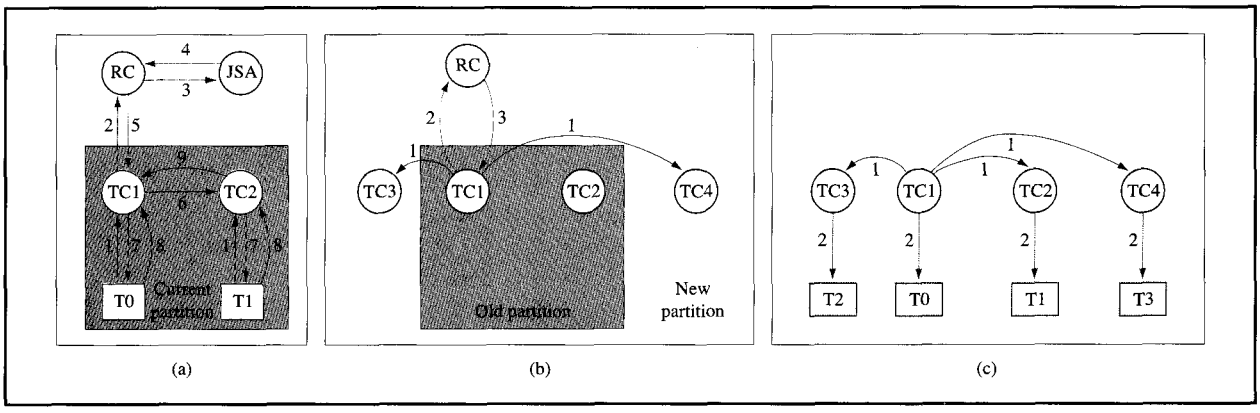

Figure 3

Job-initiated expansion: (a) Resource allocation; (b) partition reconfiguration; (c) job restart on new partition.

• *Implementation of application reconfiguration under DRMS*

In this subsection, we describe the steps involved in achieving the different types of processor reconfiguration in the current implementation of DRMS on the IBM RS/6000 SP. We focus on reconfiguration initiated by internal events in the application. We then discuss the differences with respect to externally (system) initiated reconfiguration.

Figure 3 illustrates the internally initiated expansion of an application from two processors to a four-processor partition. The edges represent messages between the various components of DRMS. The numbers on the edges order the events in time. When two edges have the same number, message communication along these edges can occur concurrently.

The application starts executing with two tasks: T0 and T1. When the application reaches an SOP that requires a reconfiguration to four tasks, all of the tasks of the application send an expansion request to their local TCs via the run-time system [Figure 3(a), step 1]. This expansion request provides the range of tasks on which the application can continue to execute. The application request is forwarded by the master TC of the application (TC1) to the RC, which in turn forwards it to the JSA (steps 2 and 3). Assuming that the JSA policy allocated to this job two additional processors to start two additional tasks (with IDs 3 and 4), the JSA sends its response to the RC with the details of the processors allocated (step 4). The RC in turn forwards the response to the master TC of the application, which percolates the information to the other TCs (in this case TC2) of the partition (steps 5 and 6). The TCs forward the response to the RTS within the application (step 7). Because the expansion requires a reinitialization of the processor partition, all of the tasks on the current partition exit. The master TC is informed of remote task exits by other TCs of the partition (steps 8 and 9).

At this point, the job is ready to be restarted on the new partition of four processors. The master TC includes the new TCs into its partition [Figure 3(b), step 1]. It then requests the RC for configuring the new partition (step 2). After receiving an acknowledgment from the RC (step 3), the master TC is ready to restart the job on the new partition. Figure 3(c) shows the job restart. The TCs coordinate and restart the application with four tasks on the new partition (steps 1 and 2). At this point, the control is handed over to the DRMS RTS which, on the basis of the information received from the TCs, performs data redistribution across the new set of tasks.

Executing an interactive job on a dynamically changing number of processors implies that the standard I/O from the tasks must be properly directed to the user interface (UIC) that submitted the interactive job. To achieve this redirection, the tasks dynamically attach to the UIC during the task restart phase, in order to send their output and receive user input. Thus, in a reconfiguration, the current set of tasks detach from the UIC before termination, and the newly spawned tasks attach to UIC before commencing the application execution.

Job-initiated shrinkage is achieved nearly identically to job-initiated expansion. Steps 1 through 9 of Figure 3(a) are identical, after which the master TC informs the RC about the free processors, along with its reinitialization request for the smaller new partition. After receiving the reinitialization acknowledgment from the RC, the master TC then restarts the application on the new (smaller) partition.

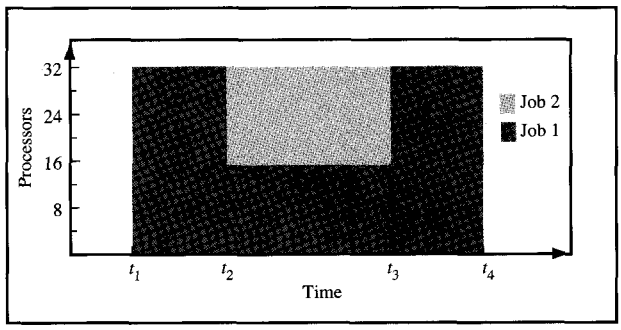

Figure 4

Reconfiguring a job (Job 1) to accommodate another job (Job 2).

Also, when there is a change in the system resource requirements, the JSA can expand or shrink a reconfigurable job so as to better utilize the system resources. This reconfiguration is initiated by sending a signal asynchronously from the JSA to the application tasks via the RC, the TCs, and the RTS. When the application reaches an SOP, the reconfiguration flag is examined, and the same steps as those for a job-initiated reconfiguration are performed.

- *Example of a reconfigurable job execution*

We now discuss how the execution of a reconfigurable job would proceed under DRMS in a hypothetical job submission scenario. Let there be two job arrivals, J_1 and J_2, one at time t_1 and the other at time t_2, $t_1 < t_2$. Let both jobs be Jacobi computations as in the example of Figure 2, with job J_2 smaller in terms of total amount of computations performed. Also, let the system have 32 processors, which are all free at time t_1. The behavior of processor allocation to each job is illustrated in **Figure 4**.

When job J_1 arrives at time t_1, it can start executing on the largest possible number of processors, 32, since the system is empty (assuming that the scheduling policy allows such an allocation). When execution encounters the `RESIZE` directive (collectively by all tasks) in the first iteration, the application informs the JSA of the valid range of tasks (4, 8, 16, 32). Since the application is already running on a valid partition size (32) and there are no competing jobs, there is no need to reconfigure the application, and the execution simply continues on 32 tasks. The JSA and the RTS both *remember* the range specified, anticipating that the application may execute a `RESIZE` instruction again. During the next iterations there is no longer any interaction between the application and the JSA, since the application is already executing on a valid number of tasks and the JSA is aware of the valid partition sizes for this application. Also, the data distribution is the same as long as the number of tasks is fixed. Therefore, there is no need for redistribution.

At time t_2, job J_2 arrives. All processors are busy with job J_1, but the JSA knows that J_1 is a reconfigurable application. Therefore, it sends a reconfiguration request to the application, through the TCs and the DRMS runtime system. This request is used to set a flag internal to the RTS that indicates that the JSA would like to reconfigure this application. When the application executes the next `RESIZE` instruction, it contacts the JSA, which reallocates resources and assigns a new set of tasks for each job. For example, 16 tasks might be given to each application. The execution of job J_1 then continues, after the resize, on 16 tasks. Because of this change in the number of tasks, the data distribution is no longer the same as in the previous iteration. The RTS performs all necessary data movement to accomplish the new distribution while preserving the semantics of the global data structures. Array descriptors and array handles are also updated to reflect the new distribution.

Both jobs then continue execution on 16 processors until J_2, which is smaller, finishes at time t_3. At that point the JSA reacquires those 16 processors and sends another reconfiguration request to J_1, because it knows that that job can use the processors. Again, when J_1 encounters `RESIZE` instruction the next time, it contacts the JSA, which reallocates all 32 processors to J_1. Job J_1 continues execution on 32 tasks and again redistributes its data. The job keeps executing on 32 tasks, without further contacting the JSA, until it finishes at time t_4.

8. Performance studies of application reconfiguration

We performed a series of experiments to verify the functionality and measure the performance of DRMS in reconfiguring parallel applications. Our performance evaluation goals in this section are to quantify the cost of a reconfiguration as seen by an individual application. The positive impact on the overall system performance of efficient application reconfiguration has been demonstrated elsewhere [4, 7, 8]. In particular, it was analyzed for the DRMS environment in [19], and some results of that analysis are presented and discussed in Section 9. We note here that the implementation of DRMS we used for the performance study is not yet fine-tuned for performance, but it is continuously evolving and is being tested for functionality. As such, the performance results presented in this section should not be interpreted in the absolute sense, but should be used as relative measures showing performance trends.

From an application point of view, there are two components of the cost for *reconfigurability*: the computation overhead of the reconfigurable DRMS program as compared to the original SPMD version, and

the actual cost of performing a reconfiguration. The computation overhead is introduced by our program transformations that add the necessary code to create a reconfigurable application. Note that this computation overhead is incurred whenever the application executes, regardless of the number of reconfigurations it goes through. We measure this computation overhead by comparing the *steady-state* (i.e., after cold cache misses and page-fault transients) execution times of the DRMS and SPMD versions when executed on the same number of processors. We measure the reconfiguration time as the elapsed wall-clock time from the moment an application reaches an SOP that changes its pool of resources (tasks) to the moment it is ready to continue its execution on the new pool. For our current implementation of DRMS, there is a one-to-one correspondence between tasks and processors. We refer to processors in our performance studies because they are the physical devices that are allocated to an application. We also use the term PE (processing element) to refer to a processor.

In our implementation on the RS/6000 SP, we identify five components of the reconfiguration time: "switch" time, "exit" time, "spawn" time, "redistribution" time, and "other" components. Switch time is the time it takes to reconfigure the partition data structures that control message routing through the interconnection network (high-performance switch). The partition data structures must be updated to support communication across the new set of processors. Our approach to application reconfiguration actually involves terminating the application on the old set of processors and restarting it on the new set, from the point at which it was stopped. We call the times for these operations the exit and spawn times, respectively. In general, a change in the processor set executing an application involves a change in the distribution of data across processors. This component is the redistribution time. Finally, we group all other costs under the other components, including the time the application takes to communicate its resource requirements to the RC and the time it takes the JSA to allocate the new set of processors. Also included is any application-specific reinitialization necessary to proceed with execution on the new set of processors.

The redistribution time itself can also be subdivided into four components: 1) "Compute" time is the time it takes for computation of the slices of data that must be exchanged among processors. 2) "Buffer" time is the time it takes to copy data from/to their actual locations to/from intermediate buffers used for data exchange. 3) "Message" time is the time for actual data exchange among processors using message-passing. 4) "Sync" time is the average time processors wait for the slowest processor in the redistribution operation before synchronizing at the end of it. Because of load imbalances and the nature of the message-passing operations, some processors take longer than others in their redistribution operation.

Table 1 Characteristics of benchmarks used to evaluate DRMS performance.

Application	*Distributed arrays*	*Problem space* (elements)	*Problem space* (MB)	*Iterations*
JACOBI	2	2×4098^2	256	200
APPBT	6	42×102^3	340	200
APPLU	5	17×102^3	138	250
APPSP	8	24×102^3	194	400
CHOLESKY	1	5,308,247	41	35,588

- *Experimental environment*

We conducted our experiments on a 32-processor partition of an IBM RS/6000 SP with wide nodes at the NASA Ames Research Center. Each processing element on the RS/6000 SP considered in this study is an IBM RS/6000 Model 590 processor (POWER2 Architecture*), with 256 KB of data cache and 66.5 MHz clock speed. On the RS/6000 SP, the processors are interconnected via a high-performance switch. Other important performance parameters of this machine, for our experiments, are its memory-to-memory transfer bandwidth within a node (B_m = 290 MB/s) and its unidirectional node-to-node communication bandwidth through the network (B_n = 36 MB/s). We refer the interested reader to [20] for further details on the RS/6000 SP. We measured the elapsed times for the execution of operations using a real-time clock with effective resolution better than 1 μs. All of the elapsed times measured were larger than 50 ms.

- *Benchmarks*

We used five different applications as benchmarks in our study: JACOBI, APPBT, APPLU, APPSP, and CHOLESKY. All of these applications operate on large distributed arrays; they are organized as a main loop, with a problem-size-dependent number of iterations, that performs almost the entire computation. For each of the applications, we started with SPMD versions optimized for the RS/6000 SP. We then manually added DRMS annotations to define an SOP at the beginning of each iteration. Thus, in the DRMS version, each iteration is a stage that can be executed on a potentially independent set of processors. JACOBI uses MPI [21] for message-passing; all other applications use MPL [22].

Some intrinsic characteristics of the applications are shown in **Table 1**. For each application we list the number of distributed arrays; the volume of problem data that must be distributed, both in number of elements and MB (one element is a double-precision floating-point value);

Table 2 Results for steady-state performance without any reconfiguration.

Application	*PEs*	*drms* $\mu \pm \sigma$ (ms)	*spmd* $\mu \pm \sigma$ (ms)	*%*
JACOBI	32	63 ± 5	58 ± 1	8
	16	123 ± 11	112 ± 1	9
APPBT	32	1198 ± 8	1166 ± 6	3
	16	2335 ± 17	2255 ± 9	4
APPLU	32	534 ± 3	542 ± 10	−1
	16	968 ± 11	980 ± 11	−1
APPSP	32	467 ± 10	440 ± 4	6
	16	874 ± 8	836 ± 5	5
CHOLESKY	32	362 ± 1	361 ± 0	0
	16	592 ± 1	591 ± 1	0

and the number of iterations of the main loop that are executed. Because of some overlap of data caused by borders, the actual volume of distributed data is, in general, dependent on the number of processors and larger than that of the problem data. We describe the key features of each of these applications that are relevant to this study. For further information, we point to pertinent references.

JACOBI

This benchmark solves the well-known Poisson equation on a square grid using the point Jacobi relaxation method [23]. The grid is discretized using a second-order central differencing scheme which results in a *five-point stencil*. The numerical scheme consists of updating the values at each grid point with the average of the values (from the previous iteration) at the four neighboring grid points. This phase is referred to as the *relaxation* phase. The solution is iteratively improved until a desired convergence criterion is met. For this, a convergence check (by computing the *error norm*) is performed at the end of each iteration. In our study, we performed a fixed number of iterations (200), in which each iteration consisted of a relaxation and a computation of the error norm. The distributed data structures consist of two bidimensional arrays (each with one element per grid point), error-norm components (one per grid partition), and two scalar values defining the problem (grid size and number of iterations) which are replicated on each task. A two-dimensional `BLOCK` distribution, with borders, was used. The grid size we selected was 4098 × 4098, including the boundary condition frame (this results in an actual relaxation grid of 4096 × 4096). For purposes of measuring reconfiguration times, we forced a reconfiguration every ten iterations.

APPBT, APPLU, and APPSP

These applications are part of the NAS parallel benchmark (NPB) suite released by the NAS program at the NASA Ames Research Center for benchmarking highly parallel supercomputers [24, 25]. They resemble closely the state-of-the-art Computational Fluid Dynamics (CFD) application codes and are representative of the computations commonly encountered in aerophysics applications [26]. The depth of these benchmarks, and the fact that they capture the essence of typical large-scale CFD applications, have made them popular not only for the purpose of evaluating parallel supercomputing systems, but also in demonstrating the viability of novel software and architectural concepts. Since these three benchmarks have some structural similarities, we discuss them together. For complete details on the problem solved and the numerical techniques used by these benchmarks, refer to Chapter 3 of [24]. The benchmarks compute a numerical solution to a synthetic system of five nonlinear partial differential equations (PDEs) that represent some of the key characteristics exhibited by the Navier–Stokes equations. An implicit type of numerical solution is used in solving this system of PDEs. The solution phase consists of several time steps, each involving 1) assembly and solution of the linear system (three substeps); 2) computation of the right-hand sides of the equations; and 3) updating the solution for the next time step. The benchmarks differ in the manner in which the linear systems are assembled and solved. For APPBT and APPSP, three-dimensional `BLOCKD` distributions with borders were used. APPBT runs for 200 iterations, and APPSP for 400. For APPLU, which runs for 250 iterations, a two-dimensional `BLOCKD` distribution was used. The class B benchmarks (larger problem size) were utilized. As with JACOBI, a reconfiguration was forced every ten iterations for each of the applications.

CHOLESKY

This benchmark computes the Cholesky factor L of a symmetric positive definite matrix A such that $A = LL^T$ [27]. Matrices A and L are sparse and are stored in column-compressed form in this benchmark. Each column has a different number of elements, and the matrices are stored in a single one-dimensional vector. A recursive partitioning algorithm [28] is used to assign individual columns to processors. In DRMS, a one-dimensional arbitrary distribution was used to declare the data configuration. As input to the factorization, we used the STK31 matrix from the Harwell–Boeing collection [29]. We forced a reconfiguration every 500 iterations. A more thorough discussion of our implementation of reconfigurable CHOLESKY, including detailed performance measurements with several test matrices, can be found in [30].

Table 3 Reconfiguration results—I.

Application	*Operation*	*Reconfigure* $\mu \pm \sigma$ (s)	*Redistribute* $\mu \pm \sigma$ (s)	*Rate* (MB/s/PE)	*First* $\mu \pm \sigma$ (ms)
JACOBI	16 → 32	12.99 ± 2.24	0.96 ± 0.02	16.85	59 ± 1
	32 → 16	12.82 ± 2.35	1.14 ± 0.00	14.14	114 ± 1
APPBT	16 → 32	16.16 ± 2.96	4.15 ± 0.01	6.66	1215 ± 8
	32 → 16	14.51 ± 2.27	3.75 ± 0.01	6.85	2351 ± 16
APPLU	16 → 32	12.35 ± 3.68	2.03 ± 0.01	6.03	1352 ± 11
	32 → 16	13.08 ± 1.79	1.90 ± 0.01	5.66	2476 ± 24
APPSP	16 → 32	9.94 ± 3.21	2.74 ± 0.01	5.75	487 ± 6
	32 → 16	12.79 ± 0.53	2.68 ± 0.01	5.46	910 ± 6
CHOLESKY	16 → 32	15.49 ± 2.82	0.31 ± 0.00	8.27	—
	32 → 16	14.33 ± 1.73	0.30 ± 0.00	8.43	—

• *Results*

In our first set of experiments we reconfigured each application from 16 to 32 and from 32 to 16 processors. Each reconfiguration was performed at least 100 times (multiple runs of each application were necessary), and we discarded the 10% smallest and largest samples. We performed our analysis on the remaining (filtered) samples. We summarize the observations for this set of experiments in **Tables 2** and **3**. For each application and reconfiguration we list various results, in the form of mean and standard deviation ($\mu \pm \sigma$) for the filtered samples. The notation $P_1 \rightarrow P_2$ denotes reconfiguration from P_1 (source) to P_2 (target) processors.

Table 2 compares the steady-state time per iteration for the DRMS and SPMD versions of the application when executing on the same number of processors. The column *drms* lists the time for the DRMS version (t_{DRMS}), and the *spmd* column lists the time for the SPMD version (t_{SPMD}). The time for CHOLESKY is for 1000 iterations; it was obtained by dividing the total factorization time by the number of iterations. For the other applications, the iteration times were obtained by direct measurement. The % column is a measure of the increased length of the DRMS times compared to the corresponding SPMD times. It is computed as $100 \times (t_{DRMS} - t_{SPMD})/t_{SPMD}$. Note that negative values indicate that the DRMS version is faster.

In Table 3, the *Reconfigure* column presents the total reconfiguration time. The *Redistribute* column presents the redistribution component of the total reconfiguration time. The *Rate* column lists the *redistribution rate* for the operations. The rate is computed as the actual amount of distributed data in the target partition, divided by the redistribution time, divided by the number of processors in the smaller partition (the smaller partition, in general, limits the rate). Therefore, the redistribution rate is in units of MB *per* second *per* PE.

The column labeled *First* lists the execution time of the first iteration after a reconfiguration, on the new set of processors. Because of page and cache misses, we expect this first iteration to execute more slowly than when the application is in steady state on a fixed number of processors (compare to Table 2). This penalty to reachieve steady-state performance is also a form of reconfiguration cost, paid by the application. It is not included in the *Reconfigure* column. For our first four applications, the whole data space is traversed on each iteration. Therefore, steady-state operation is reached after the first iteration after a reconfiguration. Because of the small and highly variable iteration times for CHOLESKY, we do not have this result for the last application.

The breakdown of reconfiguration and redistribution times into their components is shown in **Figure 5**. The notation $P_1 : P_2$ at the top of each bar denotes a reconfiguration from P_1 to P_2 processors. The horizontal line and number at the top of each vertical bar represent the total time (in seconds) of each operation (reconfiguration or redistribution). The different shadings on the bars represent the time required for each of the four major components of reconfiguration or redistribution. The blank space between the top of the bar and the horizontal line, for reconfiguration, is the time required for the operations that we have lumped together into the "other" category. For clarity of presentation, Figure 5 shows only the expected values of the components. For completeness, **Tables 4** and **5** show the mean and standard deviation for each individual component we measured. The % column in Table 4 lists the percentage of total reconfiguration time represented by the sum of switch, exit, and spawn times, which are the

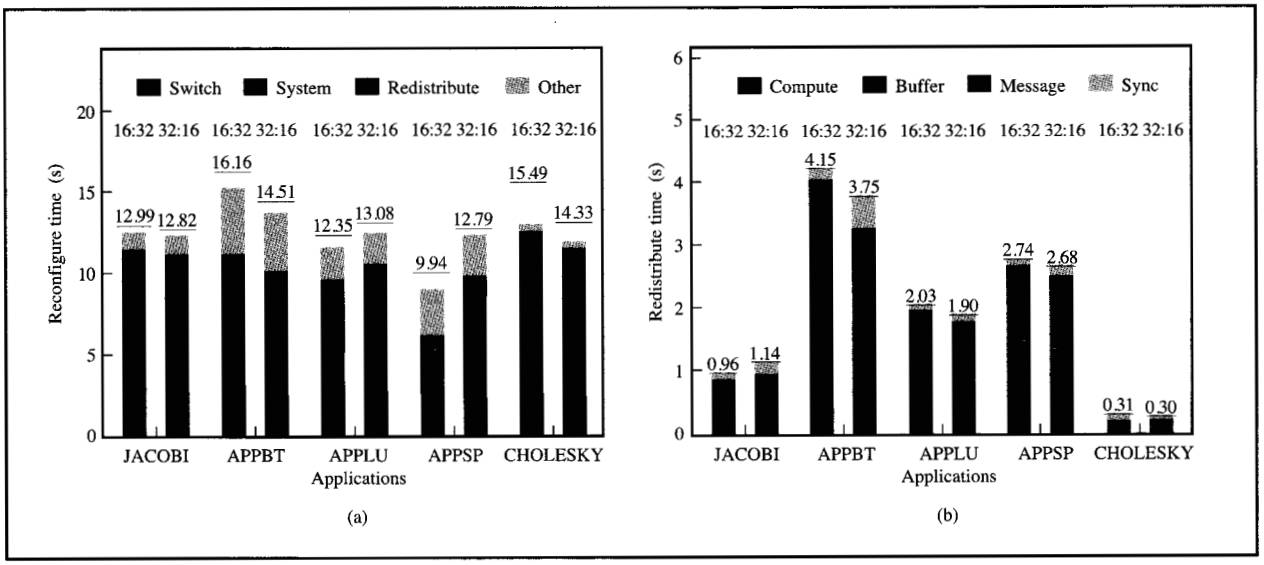

Figure 5

Component costs of (a) reconfiguration and (b) redistribution times using 16- and 32-processor partitions.

system-related, as opposed to application-related, components of reconfiguration.

We note, from Table 3 and Figure 5, that the total reconfiguration time is much larger than the redistribution time, and also has a much larger variability. The coefficient of variation C_x for reconfiguration time can be as large as 0.32 (APPSP, $16 \rightarrow 32$). Because of this large variability, our worst-case 95% confidence interval for the mean μ is $\mu \pm 0.07\mu$. We note that the redistribution times have much smaller variance. The largest C_x is only 0.02 (JACOBI, $16 \rightarrow 32$), and the 95% confidence intervals for the mean are all better than $\mu \pm 0.004\mu$. Redistribution times show little variance because this operation involves only processors that, at the moment, are exclusively assigned to the application and communicate only through the high-performance switch.

The variability in reconfiguration time comes from components other than redistribution, specifically exit and spawn. This variability is caused mostly by external factors that are outside the control of DRMS and influence a reconfiguration. The 32-processor partition we used in our experiments is part of a much larger RS/6000 SP system, shared by many users. Although the processors in the partition were assigned exclusively to DRMS, other resources are shared (e.g., the interconnection network, file system, Ethernet). The *partition manager*, which configures the partition in preparation for job execution (and for DRMS reconfigurations), is shared among all jobs in the RS/6000 SP. The number of requests that the partition manager receives and network delays in contacting it (it executes on one particular node of the RS/6000 SP) account for variability in the switch time. The exit and spawn times involve socket communication through the local area network (Ethernet) among the various TCs. The traffic in this Ethernet in particular, and the contention for socket operations in general, explain both the variability and large values for the exit and spawn components. We observe in Figure 5 that, for a given application, the exit time grows with the size of the source partition, since more processors must exit. Conversely, the spawn time grows with the size of the destination partition. The switch time is more symmetrical.

Note that for most applications, the "other" component of the resize time is approximately 0.5 s, and thus a small fraction of the overall reconfiguration time. CHOLESKY has a more elaborate application reinitialization procedure; therefore, the "other" time is much larger, of the order of 2.5 s.

From the redistribution results in Table 3, we note that the redistribution rates vary from 5.5 MB/s/PE up to 17 MB/s/PE. The three NAS benchmarks have approximately the same rate. The slightly larger efficiency of APPBT can be explained by its much larger data space. JACOBI has a much higher redistribution rate than the NAS applications. From Figure 5 we note that the NAS applications spend a much larger fraction of their time computing the slices and copying data to/from buffers than JACOBI. This difference can be explained because the

Table 4 Analysis of reconfiguration cost.

Application	*Operation*	*Reconfigure* $\mu \pm \sigma$ (s)	*Switch* $\mu \pm \sigma$ (s)	*Exit* $\mu \pm \sigma$ (s)	*Spawn* $\mu \pm \sigma$ (s)	*%*	*Redistribute* $\mu \pm \sigma$ (s)	*Other* $\mu \pm \sigma$ (s)
JACOBI	16 → 32	12.99 ± 2.24	3.05 ± 0.63	1.95 ± 2.15	6.47 ± 1.83	88	0.96 ± 0.02	0.56 ± 1.95
	32 → 16	12.82 ± 2.35	2.76 ± 0.28	6.22 ± 0.02	2.21 ± 2.28	87	1.14 ± 0.00	0.50 ± 0.71
	8 → 16	5.94 ± 2.17	1.77 ± 0.24	0.15 ± 0.01	1.82 ± 1.99	63	1.68 ± 0.00	0.52 ± 0.91
	16 → 8	6.00 ± 1.99	1.70 ± 0.33	1.57 ± 1.81	0.64 ± 0.04	65	1.57 ± 0.03	0.51 ± 0.80
APPBT	16 → 32	16.16 ± 2.96	3.07 ± 0.70	2.98 ± 2.04	5.16 ± 2.64	69	4.15 ± 0.01	0.80 ± 2.54
	32 → 16	14.51 ± 2.27	2.53 ± 0.36	5.76 ± 1.26	1.86 ± 1.96	70	3.75 ± 0.01	0.61 ± 1.08
	8 → 27	13.16 ± 2.63	2.00 ± 0.21	0.31 ± 0.16	4.15 ± 2.73	49	6.34 ± 0.02	0.35 ± 0.48
	27 → 8	13.69 ± 2.47	2.23 ± 0.37	4.23 ± 2.59	0.66 ± 0.02	52	6.13 ± 0.06	0.44 ± 0.68
APPLU	16 → 32	12.35 ± 3.68	3.24 ± 0.64	1.94 ± 1.87	4.49 ± 2.79	78	2.03 ± 0.01	0.65 ± 1.53
	32 → 16	13.08 ± 1.79	2.87 ± 0.36	6.32 ± 0.06	1.58 ± 1.66	82	1.90 ± 0.01	0.42 ± 0.63
	24 → 32	15.79 ± 3.04	2.76 ± 0.53	5.77 ± 1.31	5.17 ± 2.71	87	1.70 ± 0.01	0.40 ± 1.72
	32 → 24	14.66 ± 2.97	2.75 ± 0.35	6.31 ± 0.04	3.49 ± 2.79	86	1.57 ± 0.01	0.54 ± 0.72
APPSP	16 → 32	9.94 ± 3.21	2.74 ± 0.57	0.88 ± 0.86	2.62 ± 2.23	63	2.74 ± 0.01	0.96 ± 1.18
	32 → 16	12.79 ± 0.53	2.47 ± 0.36	6.25 ± 0.06	1.05 ± 0.16	76	2.68 ± 0.01	0.33 ± 0.13
	8 → 32	10.45 ± 2.77	2.20 ± 0.34	0.33 ± 0.02	3.24 ± 2.52	55	3.95 ± 0.01	0.72 ± 0.94
	32 → 8	13.13 ± 0.61	1.95 ± 0.43	6.24 ± 0.04	0.73 ± 0.08	68	3.74 ± 0.05	0.47 ± 0.12
CHOLESKY	16 → 32	15.49 ± 2.82	2.70 ± 0.67	4.61 ± 2.16	5.40 ± 2.60	82	0.31 ± 0.00	2.46 ± 3.29
	32 → 16	14.33 ± 1.73	2.61 ± 0.40	7.64 ± 0.05	1.39 ± 1.23	81	0.30 ± 0.00	2.39 ± 0.80
	4 → 16	6.40 ± 1.85	1.92 ± 0.28	0.18 ± 0.01	1.68 ± 1.70	59	0.55 ± 0.00	2.06 ± 0.58
	16 → 4	10.33 ± 2.33	2.12 ± 0.40	5.38 ± 2.45	0.48 ± 0.02	77	0.48 ± 0.02	1.88 ± 0.64

Table 5 Analysis of reconfiguration cost.

Application	*Operation*	*Redistribute* $\mu \pm \sigma$ (s)	*Compute* $\mu \pm \sigma$ (s)	*Buffer* $\mu \pm \sigma$ (s)	*Message* $\mu \pm \sigma$ (s)	*Sync* $\mu \pm \sigma$ (s)
JACOBI	16 → 32	0.96 ± 0.02	0.08 ± 0.00	0.14 ± 0.00	0.68 ± 0.00	0.05 ± 0.01
	32 → 16	1.14 ± 0.00	0.09 ± 0.00	0.07 ± 0.00	0.79 ± 0.00	0.19 ± 0.00
	8 → 16	1.68 ± 0.00	0.04 ± 0.00	0.27 ± 0.00	1.33 ± 0.00	0.04 ± 0.00
	16 → 8	1.57 ± 0.03	0.05 ± 0.00	0.14 ± 0.00	1.23 ± 0.02	0.15 ± 0.01
APPBT	16 → 32	4.15 ± 0.01	0.45 ± 0.00	1.59 ± 0.00	1.97 ± 0.01	0.14 ± 0.00
	32 → 16	3.75 ± 0.01	0.45 ± 0.00	1.33 ± 0.00	1.45 ± 0.01	0.52 ± 0.00
	8 → 27	6.34 ± 0.02	0.26 ± 0.00	1.41 ± 0.00	4.19 ± 0.02	0.48 ± 0.01
	27 → 8	6.13 ± 0.06	0.28 ± 0.00	1.46 ± 0.01	2.92 ± 0.03	1.47 ± 0.02
APPLU	16 → 32	2.03 ± 0.01	0.34 ± 0.00	0.68 ± 0.00	0.98 ± 0.01	0.04 ± 0.00
	32 → 16	1.90 ± 0.01	0.35 ± 0.00	0.56 ± 0.00	0.94 ± 0.01	0.05 ± 0.00
	24 → 32	1.70 ± 0.01	0.43 ± 0.00	0.66 ± 0.00	0.57 ± 0.00	0.03 ± 0.00
	32 → 24	1.57 ± 0.01	0.44 ± 0.00	0.59 ± 0.00	0.50 ± 0.01	0.03 ± 0.00
APPSP	16 → 32	2.74 ± 0.01	0.58 ± 0.00	0.91 ± 0.00	1.18 ± 0.00	0.06 ± 0.00
	32 → 16	2.68 ± 0.01	0.59 ± 0.00	0.80 ± 0.00	1.16 ± 0.01	0.13 ± 0.00
	8 → 32	3.95 ± 0.01	0.41 ± 0.00	0.93 ± 0.00	2.56 ± 0.01	0.05 ± 0.00
	32 → 8	3.74 ± 0.05	0.42 ± 0.00	0.74 ± 0.00	2.27 ± 0.04	0.31 ± 0.01
CHOLESKY	16 → 32	0.31 ± 0.00	0.11 ± 0.00	0.02 ± 0.00	0.13 ± 0.00	0.05 ± 0.00
	32 → 16	0.30 ± 0.00	0.11 ± 0.00	0.01 ± 0.00	0.14 ± 0.00	0.04 ± 0.00
	4 → 16	0.55 ± 0.00	0.03 ± 0.00	0.05 ± 0.00	0.41 ± 0.00	0.07 ± 0.00
	16 → 4	0.48 ± 0.02	0.03 ± 0.00	0.02 ± 0.00	0.34 ± 0.01	0.09 ± 0.01

CFD applications have more distributed arrays, and each array is of higher dimensionality (up to five dimensions for APPBT). JACOBI only has two bidimensional distributed arrays, so all the steps in redistribution execute more

Table 6 Results for steady-state performance without any reconfiguration.

Application	*PEs*	*drms* $\mu \pm \sigma$ (ms)	*spmd* $\mu \pm \sigma$ (ms)	%
JACOBI	16	123 ± 11	112 ± 1	9
	8	239 ± 22	214 ± 1	12
APPBT	27	1458 ± 4	1425 ± 3	2
	8	4570 ± 15	4418 ± 17	3
APPLU	32	534 ± 3	542 ± 3	−1
	24	680 ± 4	687 ± 4	−1
APPSP	32	467 ± 10	440 ± 4	5
	8	1671 ± 7	1609 ± 6	4
CHOLESKY	16	592 ± 1	591 ± 1	0
	4	1706 ± 1	1720 ± 1	−1

efficiently. CHOLESKY only has one distributed data structure; however, its efficiency is less than that of JACOBI because its dataset is smaller and because the computation of slices of data to be transferred is more elaborate in the presence of arbitrary distribution.

Analyzing columns *drms* and *spmd* of Table 2, we observe that in one case the steady-state performance of the DRMS version was better than that of the corresponding SPMD version. The DRMS version of APPLU is 1% faster than the corresponding SPMD version. On the other hand, the DRMS version of JACOBI is up to 9% slower than its SPMD counterpart, with smaller penalties for APPBT and APPSP. We note that the iteration step in JACOBI is less elaborate than in the NAS parallel benchmarks, thus emphasizing the impact of DRMS transformations. These results show that the impact of code transformations in DRMS is minimal, and that the dense storage created by the array handles can sometimes have a beneficial effect, at least on the type of benchmarks we analyzed. We note that for CHOLESKY, where there is no difference in storage patterns for both versions, the steady-state performance was the same. In analyzing the *First* column of Table 3 (column 6), we note that for three of the applications the impact of the transient after a reconfiguration is very small. For JACOBI, APPBT, and APPSP, the first iteration after a reconfiguration executes in nearly the same time as steady-state iterations. For APPLU, the first iteration after a resize executes 2.5 times slower than steady-state. We are currently investigating this behavior to determine whether this is strictly application-dependent or whether there is some other cause in DRMS.

To gain further insight into reconfiguration and redistribution times, we performed a set of experiments in which each application was submitted to a different kind of reconfiguration. We chose partition sizes that exercised many different reconfigurations. Again, each reconfiguration was performed at least 100 times. The observations for this second set of experiments are summarized in **Tables 6** and **7**, which are equivalent to Tables 2 and 3 for the first set of experiments. The breakdown of reconfiguration and redistribution into components is shown in **Figure 6**. As before, Tables 4 and 5 contain the details of each component.

Comparing Tables 3 and 7, we observe that redistribution rates for a given application are smaller when a larger source or target partition is involved. This is to be expected because, for a given application, the amount of data per processor decreases with a larger partition size. During a redistribution, the data are also divided into more, and thus smaller, chunks for interprocessor communication when a larger partition is involved. All of these factors contribute to decrease the efficiency in data exchange between processors when the partition size is larger.

From Figures 5 and 6 we note that exit and spawn times are quite small when the partition involved has eight or fewer processors. These components then become very large for partitions of size 16 or larger. These results indicate that our mechanism for inter-TC communication, using sockets through the local area network, is not very scalable and that we should look for alternative solutions.

We can compute the *payoff* point for an expansion from P_1 to P_2 processors as the number of iterations n_p that it takes to amortize the cost of the reconfiguration. Let t_P be the time to execute one iteration on P processors, in steady state. Let t_r be the time for reconfiguration, and let us ignore the transient in the first iteration. Then n_p can be computed by solving $t_r + n_p t_{P_2} \leq n_p t_{P_1}$:

$$n_p = \left\lceil \frac{t_r}{t_{P_1} - t_{P_2}} \right\rceil . \tag{4}$$

We also define the cost in iterations n_c for a reconfiguration as

$$n_c = \left\lceil \frac{t_r}{t_{P_2}} \right\rceil , \tag{5}$$

that is, the number of iterations that would have been executed in a partition of size P_2 during time t_r. We note that P_2 processors are busy during the expansion, and are therefore prevented from doing useful work in the form of iterations. **Table 8** shows the values of n_p and n_c for our applications for expansions from 16 to 32 processors. We compute n_p and n_c using both the mean value for t_r, as reported in Table 3, and the minimum observed time from our set of samples. The results are shown in the *Mean* and *Minimum* columns of Table 8, respectively.

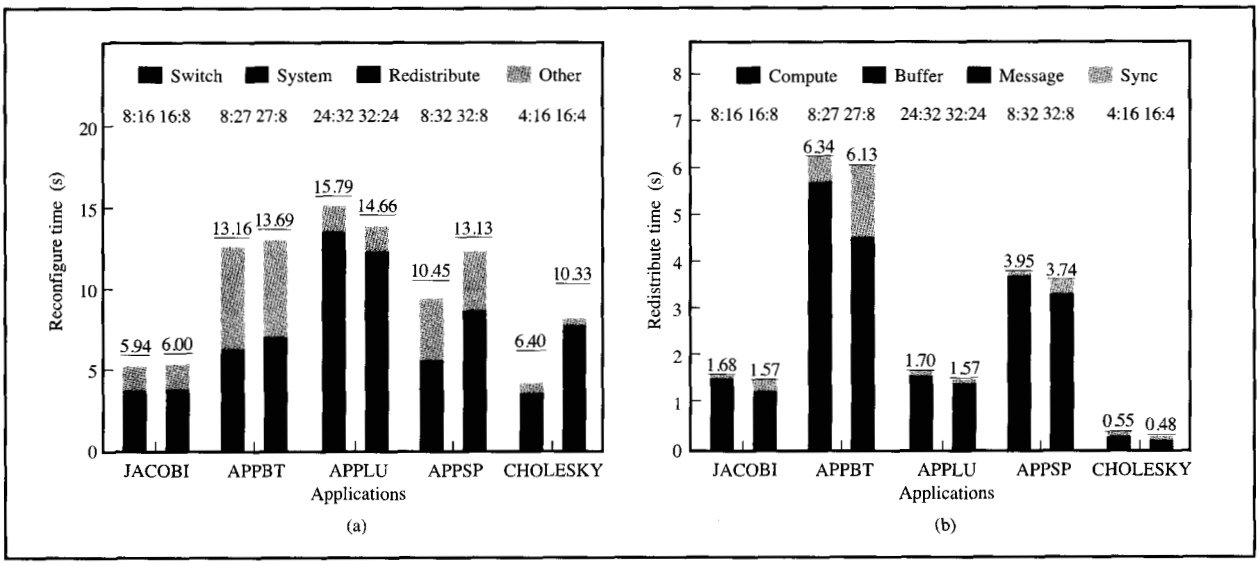

Figure 6

Component costs of (a) reconfiguration and (b) redistribution times using 4-, 8-, 16-, 24-, 27-, and 32-processor partitions.

Table 7 Reconfiguration results—II.

Application	*Operation*	*Reconfigure* $\mu \pm \sigma$ (s)	*Redistribute* $\mu \pm \sigma$ (s)	*Rate* (MB/s/PE)	*First* $\mu \pm \sigma$ (ms)
JACOBI	$8 \rightarrow 16$	5.94 ± 2.17	1.68 ± 0.00	19.15	114 ± 1
	$16 \rightarrow 8$	6.00 ± 1.99	1.57 ± 0.03	20.42	219 ± 1
APPBT	$8 \rightarrow 27$	13.16 ± 2.63	6.34 ± 0.02	8.41	1488 ± 12
	$27 \rightarrow 8$	13.69 ± 2.47	6.13 ± 0.06	7.78	4606 ± 19
APPLU	$24 \rightarrow 32$	15.79 ± 3.04	1.70 ± 0.01	4.82	1355 ± 13
	$32 \rightarrow 24$	14.66 ± 2.97	1.57 ± 0.01	4.89	1730 ± 16
APPSP	$8 \rightarrow 32$	10.45 ± 2.77	3.95 ± 0.01	7.98	484 ± 3
	$32 \rightarrow 8$	13.13 ± 0.61	3.74 ± 0.05	7.29	1727 ± 11
CHOLESKY	$4 \rightarrow 16$	6.40 ± 1.85	0.55 ± 0.00	18.30	—
	$16 \rightarrow 4$	10.33 ± 2.33	0.48 ± 0.02	21.31	—

Table 8 Payoff and cost results.

Application	t_{16} (ms)	t_{32} (ms)	*Mean*			*Minimum*		
			t_r (ms)	n_p	n_c	t_r (ms)	n_p	n_c
JACOBI	123	63	12,990	217	207	5117	86	82
APPBT	2335	1198	16,160	15	14	8079	8	7
APPLU	968	534	12,350	29	24	6298	15	12
APPSP	874	467	9940	25	22	6553	17	15
CHOLESKY	0.592	0.362	15,490	67,348	42,791	6305	27,414	17,418

Comparing the number of iterations in each benchmark, as reported in Table 1, with the payoff and cost values in Table 8, it initially appears that it would not make sense to perform such an expansion for JACOBI and CHOLESKY (especially if the mean times for t_r are used). We should note, however, that these benchmarks are often used as kernels in much larger applications. For example, CHOLESKY might be inside a loop that performs many different factorizations. The ability to reconfigure CHOLESKY in the middle of its execution means that the application can respond much more rapidly to changes in the availability of resources. The application as a whole can then benefit greatly by an increase in the number of processors. We also note that the payoff point decreases with an increase in the difference between the source and target partitions of a reconfiguration. For an APPBT reconfiguration from 8 to 27 processors (Table 7), the payoff point is only four iterations using the mean t_r.

9. Impact of application reconfiguration on system performance

Now we demonstrate that the ability to dynamically reconfigure parallel applications can lead to overall better system performance. As noted before, these results have previously been discussed in [19]. We compare the performance of three job-scheduling policies in a 32-processor RS/6000 SP when scheduling a variety of workloads typical of scientific and technical computing. Two of these are static scheduling policies that assign processors at job startup time and make no subsequent changes to the processor partition of a job. The third policy uses application reconfiguration to dynamically add processors to and remove them from partitions during the execution of a job.

The system-level performance parameter we use is the average job response time. Let a_i be the time a job J_i arrives in the system for execution. Let e_i be the time job J_i exits the system (i.e., the time at which it completes execution). Then the response time r_i for job J_i is computed as $r_i = e_i - a_i$. Note that r_i has a queueing (waiting) time component as well as an actual execution time component. The average response time for the system is simply the average of r_i for all jobs comprising the workload.

- *Workload specification*

To produce our workloads of parallel jobs we used a mix of 21 different computational fluid dynamics (CFD) applications. The 21 applications were obtained by varying the problem size and number of iterations of the three NAS parallel benchmarks APPBT, APPLU, and APPSP (see Section 8). The applications have been grouped into three categories qualitatively representing small, medium, and large jobs.

Jobs from Category I (small) can run on partitions of 1, 2, or 4 processors and require less than 1500 seconds of execution when computed on a single processor. They are representative of small interactive applications and program development runs. Jobs from Category II (medium) can run on partitions of 4, 8, 12, or 16 processors and require between 3000 and 4000 seconds of execution on four processors. They are representative of trial runs of applications. Finally, jobs from Category III (large) can run on 8, 12, 16, 20, 24, 28, or 32 processors. They require between 13000 and 20000 seconds of execution on eight processors and are representative of production runs.

We define system utilization as the job arrival rate times the average ideal service time of a 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. In our experiments, we varied the utilization from 0.1 to 0.8 in steps of 0.1, obtaining the average job response time for each utilization.

In our studies we used six different workloads, characterized by the percentage of utilization produced by jobs from each category. To define a workload we use the notation $(p_I{:}p_{II}{:}p_{III})$ to represent the percentage of utilization produced by jobs from categories I, II, and III, respectively. The six workloads we consider are (05:25:70), (15:25:60), (25:25:50), (35:25:40), (45:25:30), and (55:25:20). That is, we keep the percentage of utilization from jobs of Category II fixed at 25%, and we vary the percentage from Category I from 5% to 55%. Correspondingly, we vary the percentage from Category III from 70% to 20%.

- *Scheduling policies*

The three scheduling policies we consider for this study are lazy scheduling (LS), adaptive scheduling (AS), and reconfigurable scheduling (RS). In all cases, the scheduler maintains arriving jobs in a queue prioritized on the arrival time.

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 the available processors, giving it as many processors as it can take. LS is a *first-to-fit* policy, with an allocation preference toward the maximum eligible number of processors. Once a job is scheduled to run on a set of processors, that job runs until completion on those processors.

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; however, 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. AS is a *maximum-to-fit with priority* policy. Once a job is scheduled to run on a set of processors, that job runs until completion on those processors.

RS policy Under the RS policy, when there are processors available, jobs are scheduled in the same way as with the AS policy. In addition, when not enough free processors are available and there are jobs waiting to run, it tries to free up processors from jobs that are currently running on more than their minimum 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 running jobs to run on a larger set of processors. The RS policy adapts processor partition sizes of new and existing jobs to dynamic changes in the system load.

• *Simulation experiments*

Performance parameters of the various CFD applications and of DRMS reconfiguration operations were measured directly from the system in operation. These performance parameters were used in a DRMS system-level simulator to obtain the steady-state performance characteristics of the three scheduling policies. The length of experiments necessary to obtain steady-state results with tight confidence intervals can sometimes reach into the hundreds of thousands of job executions. Therefore, this combination of direct measurement of some parameters and simulation represents a practical trade-off to evaluate the overall system performance.

The central part of the simulator is the job scheduler, which mimics the actions of the JSA in DRMS (Section 6). The simulator is event-driven, with each event triggering an action in the job scheduler to schedule jobs for execution and decide which jobs must be reconfigured. Job arrival times are generated using an exponential distribution, with the mean interarrival time computed to deliver the desired system utilization. Workload parameters control the percentage of utilization produced by each job category.

We measure both the mean (μ) and standard deviation (σ) of n samples of job response time. To reduce the correlation between successive samples, we use the batch means method as described in [31]. After the system achieves steady state, the simulation runs until the standard error ($\delta = \sigma/\sqrt{n}$) of the response time is less than 0.01μ (1% of the mean).

• *Results*

Figure 7 shows plots of response time as a function of utilization for the three scheduling policies in each of the workloads we consider. In the experiments that generated these results, we did not allow jobs from Category I to be reconfigured (in the RS policy). We found no significant difference in results when we allowed reconfiguration of those jobs. Jobs from Category I have short execution times and use at most four processors. Therefore, reconfiguring them has only a small impact on processor allocation.

As a general observation, we note that, for any given scheduling policy and workload, average job response time increases with utilization. This is explained by longer queueing delays as the job arrival rate increases, and longer execution times as jobs have to execute on smaller partitions. We also note that, for any given scheduling policy and utilization, the average response time increases with an increase in the percentage of utilization produced by Category III jobs. Since Category III jobs have a much larger execution time than jobs from Category I, a larger percentage of longer jobs increases the overall average response time.

The most important result from our experiments is that we can establish a well-defined relation among the performances of the three scheduling policies. For any given workload and utilization, RS always performs better than AS, and AS always performs better than (or at least equal to) LS. At low utilization, the three scheduling policies perform similarly. AS performs similarly to LS because on average there are few jobs in the waiting queue. RS performs similarly to the other two because there are few situations that cause reconfiguration. The advantages of AS and RS over LS increase monotonically with utilization. The higher the utilization, the larger the average length of the waiting queue, and the first-to-fit policy of LS causes more job delays. The advantage of RS over AS increases at medium utilization and then decreases again at higher utilization. At high utilization, jobs tend to be scheduled on smaller partition sizes, and there are usually jobs in the waiting queue to fill holes left by departing jobs. Therefore, at medium utilization there are more opportunities for reconfiguration than at either low or high utilization. Overall, we note that the ability to reconfigure running parallel jobs in RS can lead to reductions of the average job response time by a factor close to 2 over a range of system utilization values.

We focus on response time as the performance parameter when comparing the different scheduling policies. It is important to note, however, that there are other important criteria for comparing scheduling policies in a system, including, in particular, fairness. While we do not attempt to perform a formal study of the fairness of the policies, some conclusions can be drawn from our experiments. First, because our measurements are from steady-state behavior, verified for each job category, we know that there is no job starvation. Second, we have measured job response times not only for the entire

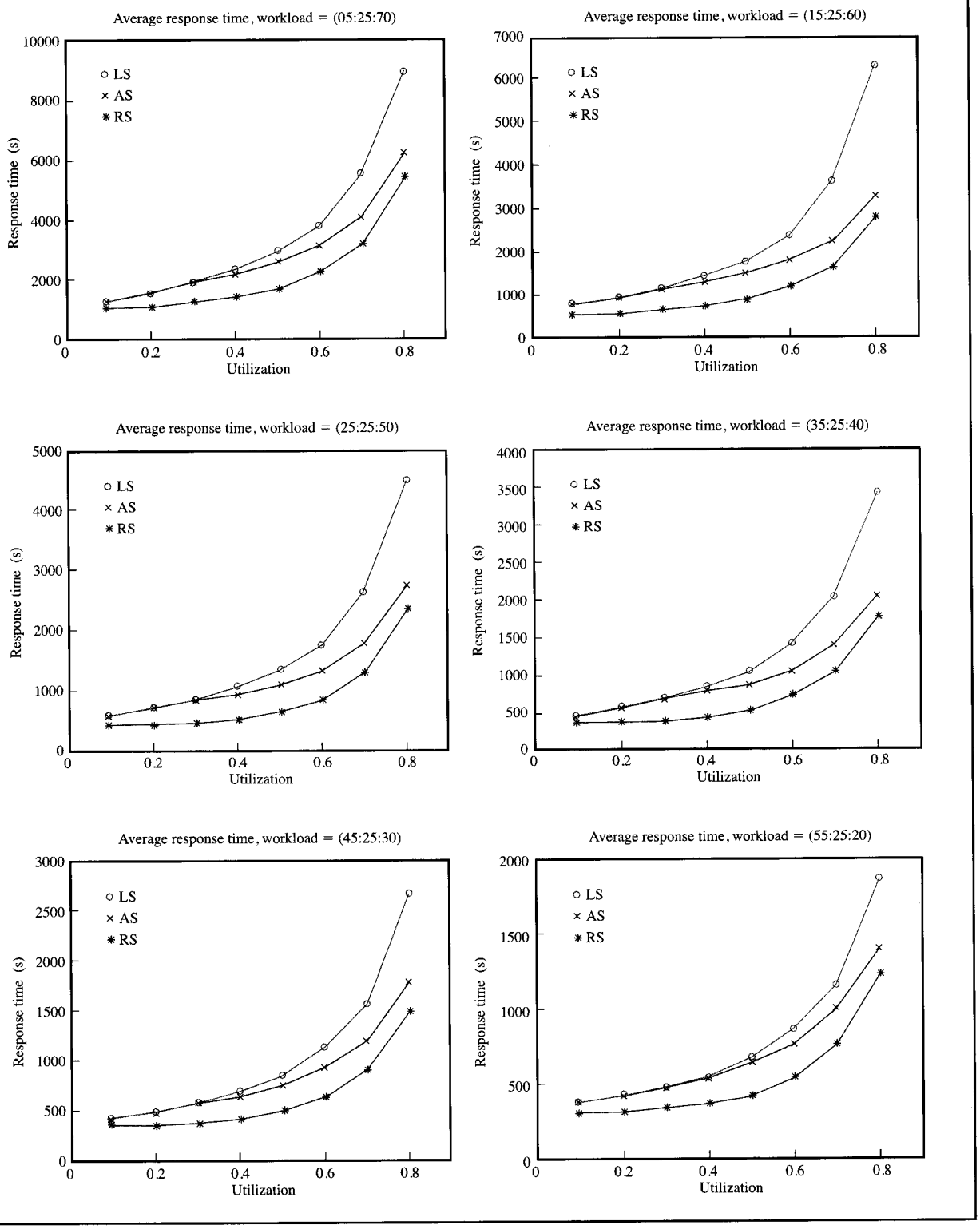

Figure 7

Average response times for various workloads with LS, AS, and RS policies.

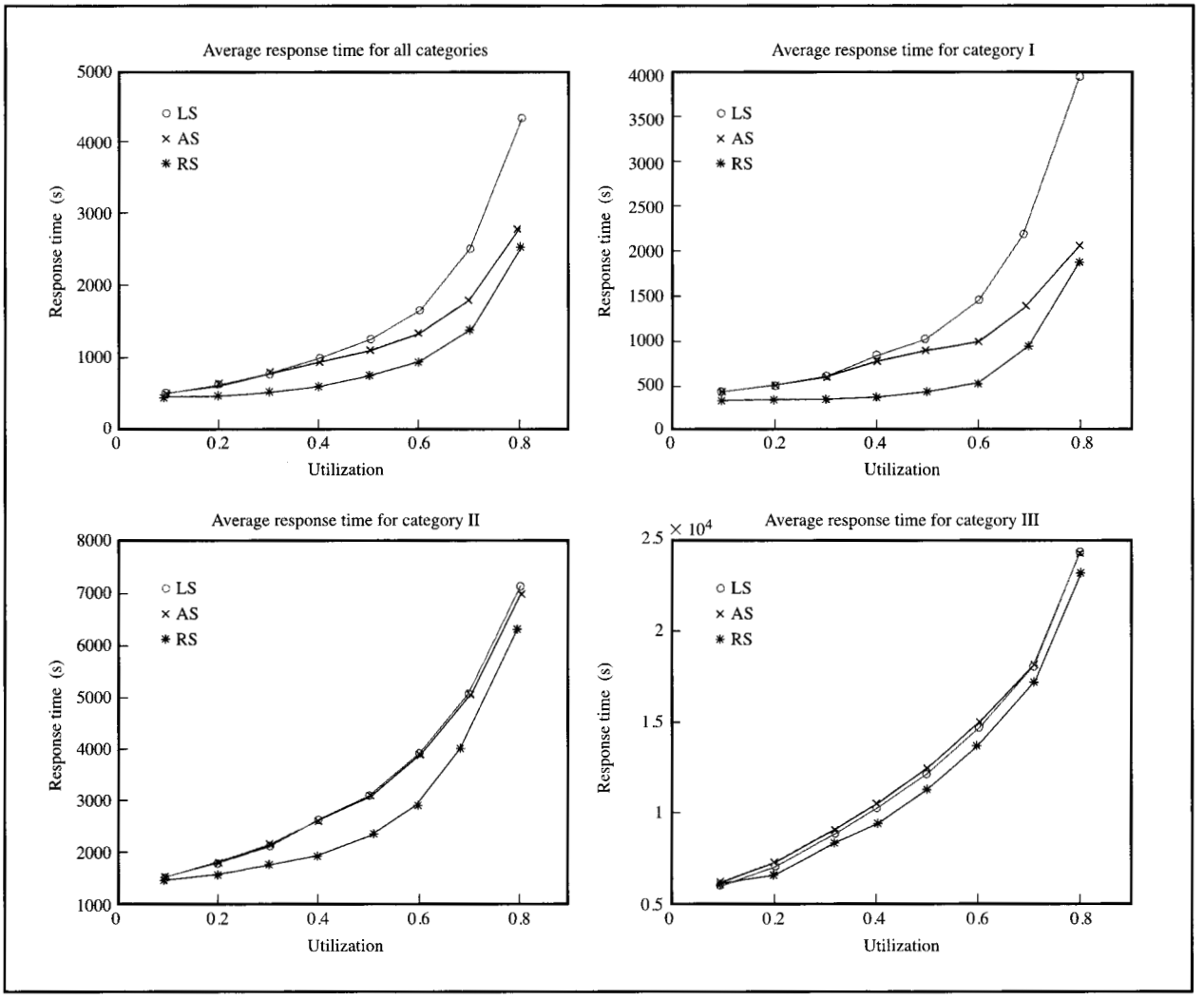

Figure 8

Average response times for different categories of jobs with LS, AS, and RS policies.

workload, but also for the individual categories in each workload. **Figure 8** shows those results for workload (25:25:50), but the same overall behavior is found in the other workloads.

From the plots in Figure 8 we observe that the advantage of RS over the other policies decreases as the size of the job increases. Category I (small) jobs benefit the most, while Category III (large) jobs benefit the least. Basically, small jobs are benefiting from the ability to reconfigure medium and large jobs to smaller partitions, freeing up processors for the execution of the small jobs. However, we note that, for each individual category, the response time under RS is always better than under the other policies. Therefore, all job types benefit from the RS policy, even though the benefits are not uniform.

Concluding our performance evaluation, **Figure 9** presents a breakdown of the average job response time into its two components: wait time in the queue and service time in the processors. Again, the results are for workload (25:25:50), with the same behavior being observed for the other workloads. The letter on top of each bar identifies the policy: L for LS, A for AS, and R for RS. We observe that both service time and wait time increase with utilization, for all policies. The increase in service time with utilization indicates that the average

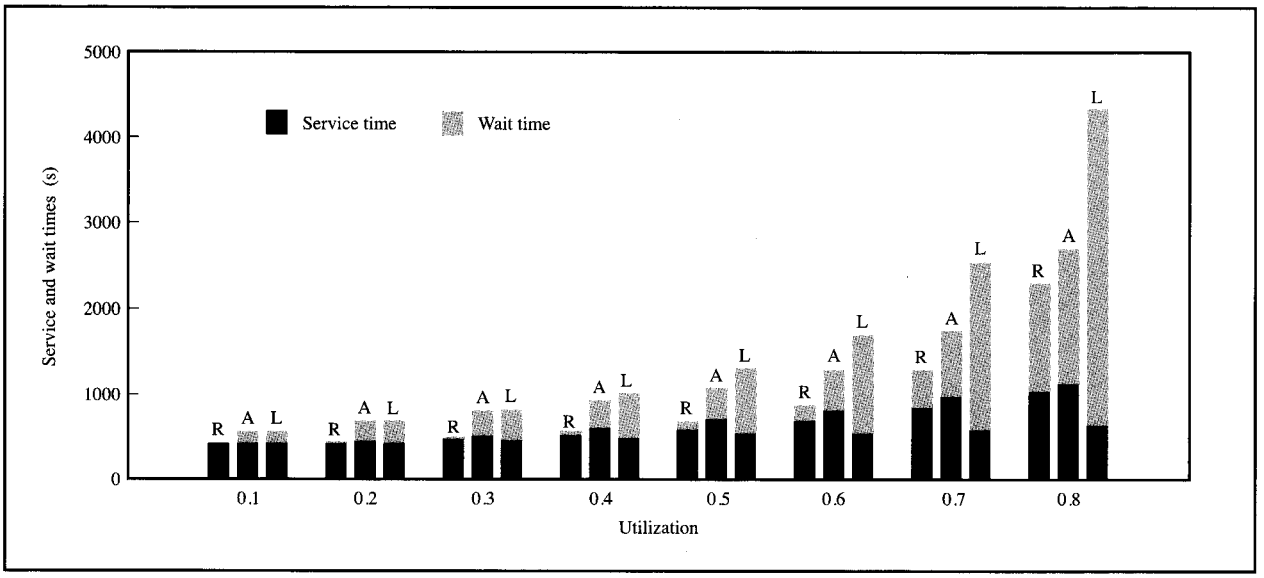

Figure 9

Component costs of average job response time into service time and wait time components.

partition size allocated to a job decreases with utilization, as expected. Wait times increase with utilization because higher contention for processors causes the jobs to wait longer in the queue.

The service times for the LS and RS policies start similarly for very small loads, and then the difference between the two increases as the load increases. The service times for RS are larger than for LS, indicating a smaller average partition size. The reduction of total response time for RS compared to LS is a result of much smaller job wait time, caused by the ability of RS to pack the jobs better and start their execution more promptly. Because of the sublinear speedup behavior of our CFD applications, efficiency improves when jobs run on smaller partitions. This factor also helps RS to perform better than LS. We note, however, that under the AS policy jobs have an even larger service time than under RS, indicating a smaller average partition size. Still, wait times and total response times are smaller for RS than for AS, thus showing the importance of reconfiguration.

10. Related work

The SOP model of programming and the DRMS framework that supports this model cover several topics of interest to researchers in the area of parallel and distributed computing. Five main areas are relevant to the topic of this paper:

- Programming models that support dynamic reconfiguration of parallel applications.
- Run-time environments that support the execution of such reconfigurable applications.
- Language support for data distribution on multiprocessor systems.
- Scheduling aspects of parallel applications.
- Reconfiguration of parallel and distributed applications for software engineering purposes.

In Section 4, we discussed four other parallel programming models (workers, fork-join, HPF, and AMP) that support dynamic reconfigurations, and also the run-time environments that support these programming models.

Extensive work has been done on language support for data distribution in multiprocessors. DRMS data-distribution annotations were strongly influenced by FORTRAN D [32] and HPF [15], primarily because we wanted to present a programming environment that was already familiar to some users. Other languages with data-distribution features include Vienna FORTRAN [33], FORTRAN 90D [34], and pC++ [35]. We note that our work differs substantially from these languages in that we provide data distribution as a support for generating dynamically reconfigurable explicit SPMD parallel applications. Our language extensions incorporate mechanisms for specifying resource requirements that are used by the scheduling module of DRMS in dynamically reallocating processors among competing applications.

One of the benefits of dynamic application reconfiguration is the ability to implement dynamic scheduling policies. Many studies have shown that processor-scheduling policies supporting dynamic reconfiguration of processor partitions can alleviate the problem of adapting to workload changes, at the expense of additional reconfiguration overhead [1–8]. In particular, [2–5] 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, it has been demonstrated [6] 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 [7, 36]. Our work differs from the previously mentioned research in that we have implemented a working environment which supports dynamic reconfiguration of processor partitions on a commercial message-passing system (IBM RS/6000 SP). We provide the language extensions and run-time services that allow users to easily port their existing SPMD applications to execute on reconfigurable partitions. We also provide all resource control and scheduling mechanisms to coordinate the execution of these jobs.

As we mentioned in the introductory section, the term *application reconfiguration* is also used in the context of software engineering of distributed applications: e.g., addition or replacement of modules, communication channels, process migration, fault recovery, and load balancing for performance tuning. Significant among these are the Regis and its predecessor programming environments [37] and Polylith and its extensions to support dynamic reconfigurations [38, 39]. In all of these environments, a separate language is provided for writing scripts external to the application for describing the application structure, its run-time behavior, and the system-level actions to be taken at reconfiguration points. Using these scripts, the run-time environment performs the desired reconfiguration actions described above. In Polylith, programmers can specify reconfiguration points, which are similar to SOPs, in the application modules.

In this paper we have not addressed the issue of application reconfiguration for fault tolerance. *Task-reconfiguration* schemes, in which tasks that are performing on a faulty processor are moved to a healthy one, are a well-known approach to reconfiguration for fault tolerance [40]. More related to our work, [41] addresses the issue of reconfiguration of SPMD programs for fault tolerance. It uses dynamic data redistribution to move the global data structures into a target healthy processor partition, where the computation is continued from the last checkpoint. Different reconfiguration strategies are discussed in [41], but it does not provide the programming environment and resource manager to automatically perform checkpoint or data redistribution. The implementation of those is left entirely to the application.

Octopus [42, 43] is another resource-management system in which application reconfiguration plays a central role. A system being managed by Octopus is divided into independent domains, which can be further divided into partitions. The domains and partitions can have their own schedulers; thus, the system is controlled and managed in a hierarchical manner. Octopus supports gang-scheduling to perform both time-sharing and space-sharing of resources among subpartitions. A mechanism called *flexible dynamic partitioning* (FDP) is used to dynamically reallocate resources among subpartitions. If a partition has to have its resources reallocated while an application is executing, the application must be reconfigured. Octopus supports both *bag-of-tasks* (workers model) and *symmetric* (SPMD model) reconfigurable applications. DRMS is different from Octopus in that DRMS defines and implements a new programming model specially designed for the development of reconfigurable applications by making only a few additive changes to existing parallel applications. This programming model is extensively supported in DRMS through a variety of data distributions and resource-control constructs that can be used directly by applications. Another difference is in the scheduling component of the two systems. In Octopus, the scheduling component is a central and integral part of the system, whereas in DRMS, resource and job coordination is central to the system. As a result, in the case of DRMS, any external site-specific scheduler can be plugged in to drive the system. This may be advantageous when reconfiguration facilities are to be made available to existing job-scheduling environments.

Finally, the Dome project [44] is also similar to DRMS in that it supports the development of reconfigurable applications using distributed data structures. It is oriented toward new C++ applications that make use of the Dome class libraries and preprocessor.

11. Conclusions

In this paper we have described the Distributed Resource Management System (DRMS), a comprehensive framework that supports dynamic reconfiguration of data parallel applications. We have developed and tested our particular implementation on a large-scale IBM RS/6000 SP system. The reconfiguration abstractions provided by the SOP programming model, described in this paper, are easy to use and are well suited for the SPMD applications that constitute the bulk of the load in such systems. We have seen, from our system-level performance evaluation, that job-scheduling policies using the application reconfiguration provided by DRMS can substantially

improve the average job response time in an IBM RS/6000 SP parallel system.

For reconfiguration purposes, in this paper we considered parallel applications based on the SPMD paradigm. To make these applications reconfigurable, we used the manual-annotations approach, in which appropriate annotations are inserted in the programs manually. In the recent past, we have extended this work on two fronts. First, we have successfully extended the SOP programming model to HPF programs, as described in [45]. We have also developed and implemented low-level FORTRAN, C, and C++ APIs for application reconfiguration that allow a finer resource and task control than the language extensions discussed in this paper. The API, which we have not discussed in this paper, broadens the scope of the reconfigurable parallel applications beyond the SPMD paradigm. On a second front, using the SOP programming model, we have been able to design and implement *scalable* checkpoint and restart facilities for parallel applications under the DRMS environment. With these facilities, a parallel application can checkpoint its execution state at an SOP in a task-independent manner. Using this state information, the application can be restarted with a different set of tasks on a different set of processors. These facilities are useful in providing fault tolerance, and for purposes of recovery and migration.

Application reconfiguration must be performed efficiently for it to deliver the promise of better system and job performance. Although we have shown DRMS reconfiguration to be effective for medium to large applications, much work remains to be done. Our main performance objective is to reduce further the relative cost of reconfiguring a parallel application. As Table 8 shows, that cost today is close to 10% of the execution time of the applications on a large processor partition. We first note that even the NAS class B benchmarks are relatively small applications. We are currently developing reconfigurable versions of production-strength applications that run for thousands of iterations and take several hours of CPU on large processor partitions. Since these applications have a dataset not much larger than APPBT, we can expect to reconfigure them in 15 to 20 s, which represents just a small fraction of their execution time.

In most cases the data redistribution time is not a limiting factor in reconfiguration performance, especially when the partitions involved are large. However, this redistribution time scales with the problem size per processor, evident for APPBT reconfiguring between 8 and 27 processors. As more powerful nodes in a parallel system are used to address larger problems, we can expect the percentage of reconfiguration taken by data redistribution to increase. Theefore, we are taking steps to optimize the redistribution operation by simplifying the computation of slices, eliminating some of the intermediate buffer copying, and performing more message aggregation.

More pressing at this time is to reduce both the total cost and variability of the other components of reconfiguration. From Table 8 it is clear that we can double the performance of reconfiguration by eliminating the external factors that cause variability. As discussed in Section 8, most of these costs are outside the direct control of DRMS, since they involve access to shared resources such as the partition manager and the Ethernet. Also, the processor partition we used for our experiments consisted of nodes scattered throughout many physical frames of an RS/6000 SP. We expect somewhat better behavior from a full set of frames. Our strategy for reducing the cost for partition reinitialization consists of letting each node reconfigure its own partition data structures directly. This avoids access to the shared partition manager and also parallelizes the operation, thus reducing its total time. We also wish to perform all inter-TC communication and synchronization over the high-performance switch instead of the Ethernet. We expect this to reduce both the time and variability of the operations, since communication over the switch is much faster and subject to less interference from other nodes. These improvements in inter-TC communication will result in reductions in exit and spawn times. The time to spawn a job on a new set of processors can be further reduced by speeding up the process of loading the code in this new set. This can be accomplished by starting the load while the partition is being initialized, effectively overlapping the two operations, and providing access to the code, specifically for the new processors in a partition, through the high-performance switch instead of the file system. One of the reasons for first implementing inter-TC communication over the Ethernet was to avoid interference with application-domain communication. As we move the TC communication to the high-performance switch, we must reevaluate the impact on application performance.

In addition to tuning DRMS for performance, we are also working toward improving the functionality provided by DRMS. As mentioned earlier, by providing a low-level API, we are extending some of the capabilities to non-SPMD applications that are developed using more general programming paradigms such as MPMD and master–slave programming models. Our future work includes development of high-level programming abstractions for these and other types of programs that are more commonly in transaction-based applications. Another important aspect of our future work is the development of a compiler which, as discussed in Section 6, automatically converts SPMD programs into the SOP programming model. Similarly, we are working on developing true

computational steering and interapplication coordination facilities using the data distribution abstractions developed for application-reconfiguration purposes. Some of the issues involved have already been discussed in [46].

A unique feature of DRMS environment is that the system provides a closer coupling between applications and the resource coordination and scheduling activities which are normally associated with the operating system. With this arrangement we have seen that both the system and the applications can benefit. However, this close coupling can be extended further. Using the Performance Data Gatherer (PDG) component of DRMS, JSA can acquire performance characteristics of applications in the system and the information on the current load on each processor. By combining this type of past information with active performance-predication capabilities about each running application in the system, intelligent scheduling decisions can be made. Currently, we are working toward developing an efficient PDG subsystem and fast on-line performance-prediction tools that can be accessed by JSA.

Acknowledgment

The authors wish to thank the anonymous referee for suggestions that led to improving the style and the presentation of this paper. This work is partially supported by NASA under HPCCPT-1 Cooperative Research Agreement No. NCC2-9000.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. K.-H. Park and L. W. Dowdy, "Dynamic Partitioning of Multiprocessor Systems," *Int. Parallel Programming* **18,** No. 2, 91–120 (1989).
2. A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 159–166.
3. S. T. Leutenneger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 226–236.
4. A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 120–132.
5. C. McCann, R. Vaswami, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors," *ACM Trans. Comput. Syst.* **11,** No. 2, 146–178 (May 1993).
6. V. K. Naik, S. K. Setia, and M. S. Squillante, "Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Computer Systems," *Technical Report RC-20239*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, October 1995. To appear in *J. Parallel & Distributed Computing.*
7. C. McCann and J. Zahorjan, "Processor Allocation Policies for Message-Passing Parallel Computers," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 19–32.
8. M. S. Squillante, "On the Benefits and Limitations of Dynamic Partitioning in Parallel Computer Systems," *Job Scheduling Strategies for Parallel Processing*, Volume 949 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1995, pp. 219–238.
9. D. Gelernter and D. Kaminsky, "Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha," *Proceedings of the International Conference on Supercomputing*, ACM, July 19–23, 1992, pp. 417–427.
10. N. Carriero and D. Gelernter, *How To Write Parallel Programs: A First Course*, The MIT Press, Cambridge, MA, 1990.
11. K. A. Robins and S. Robins, *The Cray X-MP/Model 24*, Volume 374 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1989.
12. A. Gupta, A. Tucker, and L. Stevens, "Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach," *Technical Report CSL-TR-91-475A*, Computer Systems Laboratory, Stanford University, Stanford, CA, 1991.
13. J. E. Moreira, *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors*, Ph.D. thesis, University of Illinois at Urbana–Champaign, 1995.
14. C. Polychronopoulos, "Auto-Scheduling: Control Flow and Data Flow Come Together," *Technical Report 1058*, Center for Supercomputing Research and Development, University of Illinois at Urbana–Champaign, December 1990.
15. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, Cambridge, MA, 1994.
16. E. Edjlali, G. Agrawal, A. Sussman, and J. Saltz, "Data Parallel Programming in an Adaptive Environment," *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995, pp. 827–832.
17. K. Eknadham, V. K. Naik, and M. S. Squillante, "PET: Parallel Performance Estimation Tool," *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, 1995, pp. 826–831.
18. J. E. Moreira, V. K. Naik, and R. B. Konuru, "A Programming Environment for Dynamic Resource Allocation and Data Distribution," *Technical Report RC-20461*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 1996. To appear in Proceedings of LCPC96, Ninth International Workshop on Languages and Compilers for Parallel Computing.
19. R. B. Konuru, J. E. Moreira, and V. K. Naik, "Application-Assisted Dynamic Scheduling on Large-Scale Multi-Computer Systems," *Proceedings of the Second International Euro-Par Conference (Euro-Par '96)*, Lyon, France, Volume 1124 of *Lecture Notes in Computer Science*, August 1996, pp. II:621–630.
20. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP2 System Architecture," *IBM Syst. J.* **34,** No. 2, 152–184 (1995).
21. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, MA, 1994.
22. IBM Corporation, *IBM Parallel Programming Environment for AIX, MPL Programming and Subroutine Reference*, first edition, August 1995; Order No. GC23-3893-00; available through IBM branch offices.
23. A. L. Hageman and M. D. Young, *Applied Iterative Methods*, Academic Press, Inc., New York, 1981.
24. D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," *Technical Report RNR-91-002*, NASA Ames Research Center, Sunnyvale, CA, 1991.

25. D. Bailey, B. E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lasinski, R. Schreiber, H. Simon, V. Ventakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," *Technical Report RNR-94-007*, NASA Ames Research Center, Sunnyvale, CA, 1994.
26. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results," *IEEE Parallel & Distributed Technol.* **1,** 43–51 (1993).
27. G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.
28. P. Sadayappan and V. Visvanathan, "Distributed Sparse Factorization of Circuit Matrices via Recursive E-Tree Partitioning," presented at the SIAM Symposium on Sparse Matrices, Gleneden Beach, OR, 1989.
29. I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse Matrix Test Problems," *ACM Trans. Math. Software* **15,** 1–14 (1989).
30. J. E. Moreira, K. Eswar, R. Konuru, and V. K. Naik, "Supporting Dynamic Data and Processor Repartitioning for Irregular Applications," *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems (Irregular '96)*, Santa Barbara, CA, Volume 1117 of *Lecture Notes in Computer Science*, August 1996, pp. 237–238.
31. R. Jain, *The Art of Computer Systems Performance Evaluation*, John Wiley & Sons, Inc., New York, 1991.
32. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D Language Specification," *Technical Report COMP TR90-141*, Department of Computer Science, Rice University, Houston, TX, December 1990.
33. S. Benker, B. M. Chapman, and H. Zima, "Vienna Fortran 90," *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*, 1992, pp. 51–59.
34. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu, "Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers," *J. Parallel Distr. Computing* **21,** 15–26 (1994).
35. F. Bodin, P. Beckmann, D. Gannon, S. Narayana, and S. X. Yang, "Distributed pC++: Basic Ideas for an Object Parallel Language," *Sci. Programming* **2,** No. 3, 7–22 (1993).
36. M. Madhukar, M. Leuze, and L. Dowdy, "Petri Net Model of a Dynamically Partitioned Multiprocessor System," *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, October 3–6, 1995, pp. 73–82.
37. J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Programs," *Proceedings of the Second International Workshop on Configurable Distributed Systems*, March 21–23, 1994, pp. 4–14.
38. J. Purtilo and C. Hofmeister, "Dynamic Reconfiguration of Distributed Programs," *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp. 560–571.
39. J. M. Purtilo, "The POLYLITH Software Bus," *ACM Trans. Prog. Lang. Syst.* **16,** No. 1, 151–174 (January 1994).
40. M. Peercy and P. Banerjee, "Design and Analysis of Software Reconfiguration Strategies for Hypercube Multicomputers Under Multiple Faults," *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, July 8–10, 1992, pp. 448–455.
41. M. Angelaccio, M. Colajanni, and V. Grassi, "Dynamic Data Reconfiguration for SPMD Programs in Faulty Multicomputers," *Proceedings of Fault-Tolerant Parallel and Distributed Systems*, June 12–14, 1994, pp. 151–160.
42. N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal, "Extensible Resource Management for Cluster Computing," *Technical Report RC-20526*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, June 1996.
43. N. Islam, A. L. Prodromidis, M. S. Squillante, A. S. Gopal, and L. L. Fong, "Extensible Resource Scheduling for Parallel Scientific Applications," *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997. CD-ROM Proceedings available from SIAM, 3600 University Science Center, Philadelphia, PA 19104.
44. J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel Programming in a Distributed Computing Environment," *Proceedings of the 10th International Parallel Processing Symposium*, 1996, pp. 218–224.
45. S. Midkiff, J. E. Moreira, and V. K. Naik, "Run-Time Support for Dynamic Processor Allocation in HPF Programs," *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 14–17, 1997. CD-ROM Proceedings available from SIAM, 3600 University Science Center, Philadelphia, PA 19104.
46. J. E. Moreira, V. K. Naik, and D. W. Fan, "Design and Implementation of Computational Steering for Parallel Scientific Applications," *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 14–17, 1997. CD-ROM Proceedings available from SIAM, 3600 University Science Center, Philadelphia, PA 19104.

Received August 8, 1996; accepted for publication June 4, 1997

José E. Moreira *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (moreira@watson.ibm.com).* Dr. Moreira is a Research Staff Member in the Scalable Parallel Systems Department at the IBM Thomas J. Watson Research Center. He received B.S. degrees in physics and electrical engineering in 1987 and an M.S. degree in electrical engineering in 1990, all from the University of São Paulo, Brazil. He received his Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign in 1995. Since joining IBM in 1995, Dr. Moreira has studied various topics related to the design and execution of reconfigurable parallel applications. He is coauthor of several papers on task scheduling, performance evaluation, and application reconfiguration.

Vijay K. Naik *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (vkn@watson.ibm.com).* Dr. Naik is a Research Staff Member in the Servers Department at the IBM Thomas J. Watson Research Center. His current research areas include resource management, scheduling, and scalable reconfigurable applications for high-performance parallel and metacomputing environments. He is also working in the area of specification and development of application interfaces, run-time systems for load balancing, performance estimation, and resource sharing on scalable high-performance multiprocessor systems. Dr. Naik is the author of the book *Multiprocessing: Trade-offs in Computation and Communication*. He has also authored several journal and refereed conference articles. He is the principal investigator of the Distributed Resource Management System (DRMS) project, which is part of the Cooperative Research Agreement with the NASA Ames Research Center. Prior to joining IBM, Dr. Naik was a staff scientist at ICASE, NASA Langley Research Center. He received his Ph.D. and A.M. degrees in computer science in 1988 and 1984, respectively, both from Duke University. In 1982, he received his M.S. degree from the University of Miami, and his B.Tech., in 1980, from the Indian Institute of Technology, Madras, both in mechanical engineering.