

# Exploiting Application Tunability for Efficient, Predictable Resource Management in Parallel and Distributed Systems

Fangzhe Chang, Vijay Karamcheti, and Zvi Kedem

*Department of Computer Science, Courant Institute of Mathematical Sciences,  
New York University*

E-mail: {fangzhe,vijayk,kedem}@cs.nyu.edu

---

Parallel and distributed computing is becoming increasingly mainstream, driven both by the widespread availability of commodity SMP and high-performance cluster platforms, as well as the growing use of parallelism and distribution in networked applications such as image recognition, media processing, virtual reality, and telepresence. However, many of these applications impose soft timeliness and output quality constraints on top of the traditional performance requirements, necessitating efficient, predictable management of system resources. Existing techniques are inadequate to simultaneously support these twin requirements of efficiency and predictability. In this paper, we propose a novel approach for increasing system efficiency while meeting application timeliness and quality constraints. Our approach exploits the application tunability found in many general-purpose computations. Tunability refers to an application's ability to trade off resource requirements over several dimensions including time, quality, and resource type; the resulting flexibility enables the underlying resource management system to choose an application operating point best suited to available resource characteristics. We describe language and scheduler extensions to support tunability in the MILAN metacomputing environment, and then systematically characterize performance benefits of tunability using a parameterizable task system. Our results show that application tunability is easily expressible and can significantly improve resource utilization.

---

*Key Words:* application tunability; parallel and distributed computing; resource management

## 1. INTRODUCTION

Parallel and distributed computing has become central and mainstream, driven both by a decrease in cost and a growth in its general-purpose applicability. The commodity nature of small-scale symmetric multiprocessors (SMPs) and high-performance cluster platforms [10] has ensured the widespread availability of relatively inexpensive hardware platforms. Moreover, these platforms are increasingly being used to run networked ap-

plications such as image recognition, media processing, virtual reality, and telepresence. However, these latter applications introduce additional demands on the management of computing and network resources in a parallel and distributed system. On top of the performance requirements traditionally associated with parallel applications (e.g., floating-point intensive scientific codes), these applications impose additional soft real-time constraints requiring completion of different portions of the application within specific intervals of time, and/or quality constraints requiring that the output possess certain attributes. For example, an application that is trying to analyze a live video feed to recognize important artifacts needs to complete its processing of one frame by the time the next frame arrives. In addition, it must assert with a level of confidence that no important objects are left unrecognized. Thus, such applications necessitate *efficient, predictable resource management* that must simultaneously optimize resource utilization and ensure that applications meet their timeliness and quality constraints.

Unfortunately, traditional resource management approaches in both parallel and distributed systems as well as real-time systems are inadequate for meeting these objectives. The parallel system approaches focus primarily on improving application performance and/or system utilization at the cost of providing only *best effort* guarantees to the application. For example, a specific application can experience arbitrary delays that may grow with the number of applications contending for the resources. Clearly, such delays are unacceptable for soft real-time applications that work with continuous media. Distributed system approaches, on the other hand, have traditionally emphasised functionality over performance. However, the explosive growth of the internet and the demands of applications that run over it indicate a pressing need for the latter. Predictability has typically been the focus of real-time systems, which are better at providing timeliness and quality guarantees to applications. However, they do so by being overly conservative, ensuring that enough resources are available for each application to meet its deadline. Admission control is used to ensure an underloaded system, providing application predictability at the cost of system utilization.

In this paper, we propose a novel approach for increasing parallel system utilization while meeting application timeliness and quality requirements. Our approach exploits the *application tunability* found in many computations such as the ones described above. Tunability refers to an application's ability to *trade off resource requirements over several dimensions*, including time, output quality, and resource type. Tunable applications are able to compensate for a lower allocation of resources in a stage of the computation either by requiring a higher allocation in another stage, or by lowering output quality, or by raising demand for resources of a different type. Application tunability provides flexibility to the underlying resource management system, which can now select an application operating point that improves overall system utilization, while still ensuring that an application meets its predictability requirements.

Specifying and exploiting tunability requires some changes, both in the way applications are expressed and the way system resources are managed. However, these changes can be easily incorporated into existing parallel and distributed programming languages and resource management systems. This paper describes support for predictable, tunable applications in the context of the MILAN parallel and distributed computing project [21]. We first describe language extensions to the Calypso programming language [2] for expressing application tunability, and evaluate their effectiveness by describing the construction of a tunable image processing application (junction detection). We then describe the extended

MILAN resource management architecture for exploiting tunable applications. Finally, we systematically characterize the performance benefits of tunability, using a parameterizable task system. We find that tunability helps improve both system utilization and job throughput (the number of jobs which meet their predictability requirements) over a wide range of task parameters, such as arrival rate, deadline laxity, and the shape of the required resource profile. Our results are very encouraging: application tunability is convenient to express, and can significantly improve system utilization for parallel and distributed computations with predictability requirements.

The rest of this paper is organized as follows. Section 2 provides relevant background on the MILAN system and the Calypso programming language. In Section 3, we present the idea of tunability in more detail. Sections 4 and 5 describe extensions to the Calypso source language and the MILAN resource management architecture, respectively, for supporting tunability. The performance impact of tunability is characterized in Section 6. Section 7 places our work in context with related efforts and we conclude in Section 8.

## 2. BACKGROUND AND PROJECT CONTEXT

The MILAN metacomputing system [1] provides middleware layers that enable the efficient, reliable, predictable execution of applications on an unreliable and dynamically changing set of machines. MILAN takes advantage of two execution techniques with strong theoretical foundations [17, 11]—*two-phase idempotent execution strategy*, and *eager scheduling*—to provide programmers with the view of a fault-free virtual shared memory environment, even when the underlying parallel and distributed system resources may incur faults and exhibit wide variations in processing speeds. This support is exposed to the programmer in the form of a number of programming systems: Calypso [2] described in further detail below, Chime [24] which supports distributed execution of C++ [6] programs, and Charlotte [4] which provides a web-based metacomputing infrastructure. In addition, the MILAN system consists of supporting infrastructure such as the Resource-Broker [3], a system for dynamically managing the association and integration of resources into multiple parallel computations according to user-specified policies.

### 2.1. The Calypso Programming System

The research described in this paper uses the Calypso programming system, which extends standard C++ with simple parallel programming structures. Calypso programs, similar to the BSP model [27], view computations as consisting of several parallel tasks inserted into a sequential program. These parallel tasks are responsible for performing the computationally intensive work, while the sequential code is responsible for the high-level control-flow and I/O. Within a parallel step, Calypso supports concurrent read, concurrent write (CRCW with identical write) semantics to shared data structures, with updates visible only at the end of the current step. Additionally, the execution is idempotent, implying that a parallel task (i.e., the concurrent task in a parallel step) can be executed multiple times (with possibly some partial executions), with exactly-once semantics. These multiple executions mask any faults in the underlying resources. A Calypso program executes using a master-worker model. The master executes the sequential portion of the program, and the workers execute the parallel tasks.

The Calypso programming language augments standard C++ with four keywords: `shared`, `parbegin`, `parend`, and `routine`. Globally shared variables are declared using

the keyword `shared`. `parbegin` and `parend` help delimit a parallel step consisting of a sequence of routine statements:

```
parbegin
  routine [int-exp](int width, int number)
    routine-body1
  routine [int-exp](int width, int number)
    routine-body2
parend;
```

The `routine` statements specify the tasks within the parallel step: `routine-body1` and `routine-body2` are sequential C++ program fragments, `int-exp` specifies an integer expression indicating the number of copies of each routine to be created within the parallel step, and `width` and `number` are arguments provided to each task denoting, respectively, the number of tasks created and the sequence number of the specific task among these tasks. As shown in the code fragment above, each parallel step may consist of multiple `routine` statements. Concurrency exists both inside one routine, as well as among multiple routines within the same parallel step.

## 2.2. Resource Management in MILAN

System resources in the MILAN system are allocated to computations using a two-level strategy. At the first level, an application conveys its requirements at start time to the ResourceBroker, which monitors system-wide resource availability and partitions the resources among multiple competing computations by dynamically growing and shrinking the resource set for each application. The ResourceBroker allocates resources both to computations that require a fixed amount of resources over their lifetime (e.g., a typical PVM or MPI program), and to those that are capable of adapting to changing resource availability (e.g., Calypso programs).

The second level of the resource management strategy consists of an application manager (one per application) that partitions the resource set among the individual tasks of the computation. For example, the application manager for Calypso programs is responsible for dispatching the parallel tasks to the processors in the resource set. It is able to utilize a number of machines that is fewer than, equal to, or more than the number of parallel tasks of the step. When the number of tasks exceeds the size of the resource set, the application manager automatically bunches tasks and assigns them to one machine as a group, effectively converting a fine-grained expression of parallelism into a coarse-grained form at run time.

## 3. APPLICATION TUNABILITY

Tunability refers to an application's ability to trade off resource requirements over several dimensions, including time, quality, and resource type, while still producing an output of adequate quality. In this section, we first discuss these tradeoffs in more detail, and then describe two examples of large-scale tunable parallel and distributed applications.

### 3.1. Flexible Resource Requirement Profiles

*Application tunability* is a characteristic of several parallel and distributed computations. The key attribute unifying all tunable computations is the availability of *alternate application configurations*, each corresponding to a different path of execution. Each such

execution path corresponds to a different resource utilization profile. A resource management architecture, which is aware of the multiple configurations, can exploit the differences among their resource utilization profiles to select a configuration and thereby a profile that best matches the characteristics of available system resources.

The differences in the resource utilization profiles of the alternate configurations can be characterized as tradeoffs along three dimensions: (i) *time*, (ii) *resource types*, and (iii) *output quality*.

Trading off resource requirements over the time dimension implies that a large allocation of resources in one stage of the computation's lifetime can compensate for a smaller allocation in another stage, or vice versa. For example, the artifact recognition application discussed earlier may first sample different portions of the image to decide on interesting regions, and then run a resource-intensive algorithm on these regions; spending more resources on the sampling step reduces the work that will need to be performed in the analysis step.

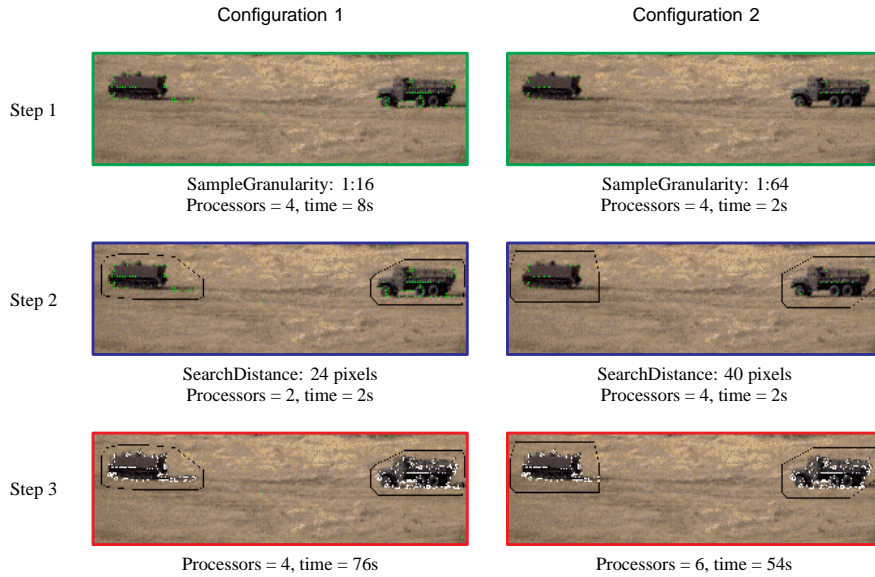
Trading off resource requirements over resource types implies that a large allocation of a particular type of resource can compensate for a small allocation of another type of resource, either in the same or a different stage of the computation's lifetime. For example, a multimedia data transmission application may send data either in compressed format or in its original form. Choosing a compressed format would save network bandwidth at the expense of more computational power required because of on-the-fly compression and decompression. The application can thus trade off computational resources versus network resources by deciding whether or not to compress the data before injecting it into the network.

Trading off resource requirements over output quality implies that the application can compensate for a reduced resource allocation by varying the quality of its output, while still operating in an acceptable range. For example, in several scientific computations where approximate results are computed, an execution of a mandatory amount of time can produce an acceptable result. Investing additional compute resources beyond that point will improve the accuracy of the result. The application can thus utilize such tunability to optimize output quality levels given available resources.

Along these three dimensions, application tunability provides flexibility to the underlying resource management system, which can now use the choice in resource allocation profiles to increase the number of applications that can be admitted into the system, while still ensuring that an application meets its predictability requirements.

### 3.2. Examples of Tunable Applications

*Junction detection: A tunable parallel application.* The junction detection [16] application detects distinguished sets of pixels in an image where the intensity or color changes abruptly. Junction detection is a core component of several image-processing applications, often serving as a precursor to shape construction and classification tasks. Our junction detection computation consists of three steps. The first step samples a subset of the pixels in parallel and performs a quick test to determine whether or not the tested pixel is of interest. A pixel is of interest if the difference among intensities/colors of its neighbor pixels is beyond a threshold. The second step draws a region of interest around a cluster of interesting pixels. The region is essentially a convex hull containing at least a certain number of interesting pixels in close proximity. Finally, the third step runs a compute-intensive algorithm for every pixel in the regions of interest.



**FIG. 1.** Junction detection—a tunable parallel application.

Junction detection is a tunable application in that the granularity of sampling in the first step can be parameterized, resulting in different resource requirements. For this application, processors are the primary resource of interest. The computation can compensate (with respect to result quality) for a coarser sampling in the first task by possibly drawing additional and/or larger regions of interest. Thus, a smaller allocation of processors in the first step (for coarser sampling) is compensated for by requiring a larger allocation in the third step. Figure 1 demonstrates this tunability, showing two configurations with different sampling granularities, different thresholds for drawing the regions of interest, and consequently different resource requirements for the third step. Tunability is represented in terms of two parameters: the sampling granularity that affects the number of pixels sampled in the first step, and a search distance metric that determines how regions of interest are constructed in the second step of the algorithm. The resource requirements, deadlines, and output qualities associated with each alternate execution path are assumed to be available a priori (these can be obtained by profiling on a training set of representative images).

*Active visualization: A tunable distributed application.* The active visualization application [8] is a general-purpose client-server application for interactively viewing, at the client side, large images stored in the server. Active visualization exploits several multi-resolution and progressive transmission techniques to reduce client latency. First, images are stored at the server as wavelet coefficients [7], enabling the construction of images at different levels of resolution (including the original one). Second, it uses progressive transmission, making it unnecessary to fetch the entire image at a single time. Based upon an initial specification of the highest resolution required by the client, the server constructs a pyramid of images ranging from the finest to the coarsest resolution. The server uses this pyramid to transmit an area of the image that corresponds to the user's fovea (the user's



FIG. 2. Active visualization—a tunable distributed application

focus of interest as identified by location of the mouse cursor), starting from the coarsest resolution and progressing up to the user-preferred resolution. If the user's fovea does not change, the client requests the server to send it an incremental region surrounding the fovea, ensuring that eventually the entire image is transmitted at the highest resolution. To further reduce client latency, the application can optionally compress the data before injecting it into the network, reducing network bandwidth at the expense of requiring decompression at the client. Figure 2 shows a snapshot of the client display highlighting the foveal region.

Active visualization is a tunable application in that its behavior is affected by the values associated with parameters such as preferred resolution level, the initial size of the foveal region, and the region increment. In addition to reflecting different user quality preferences vis-a-vis the visualization, these parameters influence the resource utilization profile of the application. A lower preferred resolution value implies a smaller amount of data processing and transmission time, and consequently a lower utilization of processing and network resources. Both the initial foveal region size and the region increment parameters affect responsiveness and total download time. The bigger the value of these parameters, the longer the round-trip time, but the shorter it takes to receive the entire image. In addition, the value of the optional compression parameter, which controls choice of the compression algorithm, impacts resource utilization at the client and server nodes as well as of the network connection between them.

#### 4. LANGUAGE SUPPORT FOR TUNABILITY

In its most general form, an application can be viewed as a collection of modules— independently compiled units of program functionality. Tunable applications are characterized by the availability of multiple execution paths, each with a different resource utilization and output quality profile. An execution path is defined in terms of the flow of control among the application modules and the behavior of each individual module. Language support is required to specify alternate behaviors of an individual module and the associated resource utilization profiles. These specifications are used by the MILAN resource management architecture, described in Section 5, to select an appropriate application execution path at run time in response to system conditions.

In this section, we describe extensions to the Calypso parallel programming language to support tunable, predictable applications. These extensions primarily associate resource requirements and output quality values with individual parallel tasks, specify how tasks may be configured as a function of available resources, and finally specify alternate control-flow

paths through the program module. The Calypso preprocessor uses these extensions to construct an application-level agent, which negotiates with the MILAN resource manager for an appropriate level of resource allocation.

#### 4.1. Different Models of Tunability

Tunability at the level of an application module refers to a modification in its behavior in response to variation in input data, user preference, or available resources. There are two independent dimensions along which such behavior modification can be expressed: granularity of code modification, and the granularity of resource change which can be detected and acted upon by the program. The first dimension refers to how module behavior modification is effected. Module behavior is controlled both at the macro-level by the algorithms it uses (*coarse tunability*), and at the micro-level by certain parameters that affect control-flow decisions (*fine tunability*). The second dimension refers to a module's ability to respond to a change in available resources. Some application modules may be able to take advantage of even a very small change in resources (*continuous tunability*), while others can only respond to a finite set of specific resource levels (*discrete tunability*).

Application modules typically exhibit tunability corresponding to three of the possible four combinations: coarse-discrete, fine-discrete, and fine-continuous. The first two situations are probably most common where a module can use different algorithms (e.g., compression algorithms in a continuous media application) or modify its control parameters in response to a change in resource availability. However, the module can only react to discrete resource levels, not the entire range. An example of the third situation is the sampling step of our junction-detection program: the sampling granularity serves as a knob which can vary application resource requirements over a continuous range.

#### 4.2. Calypso Extensions for Tunability

These extensions to the Calypso programming language identify program control parameters, alternate execution paths, and the corresponding resource and predictability requirements of program tasks. To keep our preprocessor simple, we focus on supporting only two models of tunability: coarse-discrete (where tunability is achieved by using different algorithms), and fine-discrete (where tunability is effected by modifying parameter values). Supporting fine-continuous tunability requires the preprocessor to handle symbolic expressions for resource requirements and deadlines. This is more an implementation rather than a fundamental limitation. Resource requirements are expressed in terms of number of processors over time, and predictability requirements are expressed in terms of the timeliness demands. Moreover, we restrict our attention to computations where the sequence of parallel steps is independent of program control flow. Many applications in the image recognition and continuous media processing domain satisfy this requirement.

The language extensions can broadly be classified into three categories: *declaration* constructs (for identifying control parameters), *task* constructs (for associating resource and timeliness requirements with tasks), and *structure* constructs (for specifying alternate execution paths in the code). Control parameters are declared (and optionally initialized) within the **task\_control\_parameters** block:

```
task_control_parameters {
    ...
};
```

These parameters are used by the application-level agent to appropriately configure the module.

The task construct, **task**, acts as a wrapper around a sequential or parallel Calypso step, and specifies the task deadline, its control parameters, and resource requirements and output quality values for each of the acceptable task configurations. The latter correspond to the assignment of specific values to the control parameters. The following code fragment demonstrates usage of the task construct:

```
task name [ deadline] [ parameter-list
    [({ param-values},{ resource-request}, quality) , ... ]
    // Calypso code for the task
taskend
```

The *name* argument refers to the task name, and the *deadline* argument denotes the time within which this task should complete. The *parameter-list* is a list of control parameters which will be assigned a value exactly before this task starts at the execution time. The acceptable task configurations are shown as a list in the last argument of the **task** construct. Each configuration consists of the *param-values*, which is a list of value assignments to the control parameters identified in *parameter-list*, the *resource-request*, which is a vector of values of size equal to the number of resource types, and the *quality* which describes the quality value of the task output for the current configuration. For the purposes of this paper, *resource-request* is a processor-time pair, denoting the number of processors required for the task and the time duration they are required for. At run time, a configuration can be uniquely identified by concatenating the task name with the values of its control parameters.

Two structuring constructs are provided to enable selection of an execution path through the program. The **task\_select** and **task\_loop** constructs are used to represent choice of configurations within a parallel step, and overall iterative structure of the program, respectively. The following code fragment demonstrates their use:

```
task_select
  when when-expr
    task, task_select, or task_loop constructs
  finally finally-code
  ...
  when when-expr
    task, task_select, or task_loop constructs
  finally finally-code
task_selectend;

task_loop (loop-expr)
  task, task_select, or task_loop constructs
task_looepend;
```

**task\_select** permits selection among multiple tasks within the same step whose readiness is checked using the *when-expr*. The *finally-code* is executed upon completion of the task and together with the **when** construct permits execution paths to be defined in the program. The **task\_loop** construct is used to express an iterative structure surrounding the sequential and parallel steps. Its argument, *loop-expr*, denotes the number of iterations. Both *when-expr* and *loop-expr* can only include constants and control parameters, facilitating their evaluation at module start-up time.

The language extensions are independent of the Calypso parallelism constructs, enabling tunability to be incrementally incorporated into an application program.

### 4.3. Tunability in the Junction Detection Program

We next demonstrate the use of the above language constructs in the context of the junction detection program described in Section 3. As mentioned earlier, the program is tunable with respect to two parameters, controlling the sampling granularity in the first step, and the search distance in the second step. It is assumed that the resource requirements and deadlines for each step are obtained by separately profiling the program. Here we focus on how application flexibility and these predictability requirements can be expressed.

```

task_control_parameters {
  int sampleGranularity;
  int searchDistance;
  int c;
};

task_loop( 1000 )
  for ( i=0; i<1000; i++ ) {
    task sampleImage[10][sampleGranularity][({16},{4,8},1.0), ({64},{4,2},1.0)]
      routine for sampling the image
    taskend;

    task_select
      when (sampleGranularity==16) task markRegionA[60][searchDistance][({24},{2,2},1.0)]
        algorithm A for finding region of interest
      taskend; finally {c = 0;}
      when (sampleGranularity==64) task markRegionB[60][searchDistance][({40},{4,2},1.0)]
        algorithm B for finding region of interest
      taskend; finally {c = 1;}

    task_selectend;

    task_select
      when (c==0) task computeJunctions[100][][({},{4,76},1.0)]
        routine for computing junctions
      taskend; finally {}
      when (c==1) task computeJunctions[100][][({},{6,54},1.0)]
        routine for computing junctions
      taskend; finally {}

    task_selectend;
  }
task_loopend;

```

FIG. 3. Expression of application tunability in the Junction Detection program.

Figure 3 shows the pseudo code for the tunable version of the junction detection program (the original code fragment is highlighted using boxes). The sampling granularity and search distance parameters are identified as control parameters whose values will be provided by the extended MILAN resource management architecture described in Section 5. The rest of the program uses these parameters to control the application behavior.

The tunability in the first step, `sampleImage`, is expressed using the `task` construct: the arguments state that the deadline for the step is 10.0 units, the step behavior is controlled by the `sampleGranularity` parameter, and allowable configurations correspond to `sampleGranularity=16` and `sampleGranularity=64`. The first configuration requires 4 processors for 8 units of time, while the second requires 4 processors for 2 units of time.

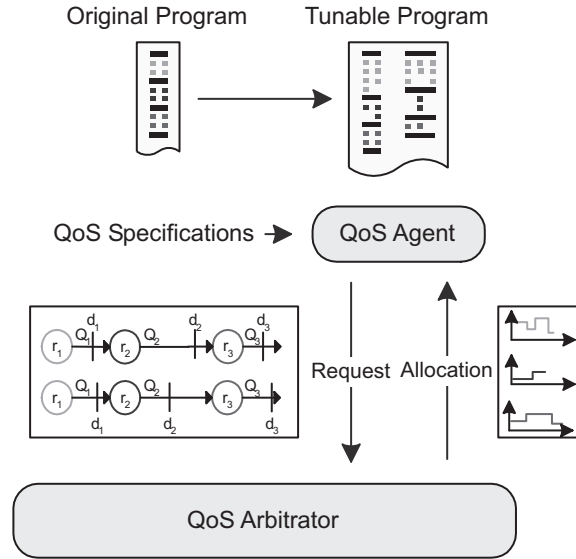
The second step, `markRegion`, admits coarse-discrete tunability, allowing use of different algorithms based on the sampling granularity used in the first step. As the code fragment shows, this level of tunability is conveniently expressed using the `task_select` construct. The `finally` code fragments are used to set up another control parameter, `c`, which is used to describe allowable task configurations in the third step. Based on the value of `c` (i.e., the task configuration chosen in the second step), only one of the `computeJunctions` configurations is allowed. This restriction of configurations based on which configurations were selected in an earlier step make explicit the application's ability to tradeoff resource requirements over its lifetime. A lower allocation of resources in the sampling step requires a larger allocation of resources in the junction computation step.

As the above description demonstrates, it is relatively straightforward to transform the original junction detection application into a tunable form. For general applications, however, this process requires some understanding of the application to discover tunable parameters and identify resource usage. Additional support (e.g., via profiling) is also required to characterize the application behavior for different configurations of these parameters and various resource availabilities; this knowledge is essential to take advantage of application tunability. Finally, exploiting tunability requires execution flexibility in both the application logic and in the underlying execution environment. For instance, dynamically changing the number of processors allocated to a parallel application not only requires that the application be able to reconfigure itself, but also that the execution environment maintain the correct run-time state. Our work has benefited from the natural availability of this support in the Calypso system; in general, the support may be application- and execution-environment dependent.

## 5. MILAN RESOURCE MANAGEMENT ARCHITECTURE

As described earlier, exploiting application tunability requires changes in how system resources are managed. At job start-up time, and optionally during execution, the resource management system must be able to influence which configuration of the tunable application is used for execution. The configuration must be chosen keeping in mind the variance in input data, user preferences, and available resources. To achieve these goals, the extended MILAN resource management architecture, shown in Figure 4, consists of two major components: an application-level *QoS agent* and the system-level *QoS arbitrator*. The component names signify our focus on providing predictable *quality of service* (QoS) for applications. The QoS agent communicates the application resource and predictability requirements to the QoS Arbitrator, which satisfies this request (and those from other applications) by providing an appropriate resource allocation. We discuss these components in detail below.

*QoS agent.* The QoS agent, automatically generated from the application's specification by a preprocessing step (see Section 4 for details), describes the application's real-time constraints, its resource requirements, its output quality, and more importantly its tunabil-



**FIG. 4.** The MILAN QoS arbitration architecture permits application tunability to be exploited to improve overall system utilization.

ity. Tunability represents choices in the execution paths available for the application. As Figure 4 shows, from the perspective of the QoS agent, the application is viewed as a collection of alternate execution paths (a chain, or more generally, a DAG) comprising several tasks, each with their own resource requirements and deadlines. Resource requirements can be thought of as a vector of values, one for each resource in the system. Each task also has an associated output quality, closely related to the requested resources. The quality value of the execution path is obtained by composing the output qualities of each of the tasks. Tunability is expressed by specifying multiple such execution paths, each with its own resource requirement and deadline profiles, representing alternate ways in which the application can consume resources in order to produce outputs with the desired quality.

The QoS agent acts on behalf of the application to negotiate with the QoS arbitrator an appropriate level of resource reservation/allocation for each task, maximizing the application output quality. In general, this negotiation involves an initial allocation that gets revised as a function of changing application demands and/or changing system conditions. For the results reported in the rest of the paper however, we restrict our attention to a relatively static negotiation model: the QoS agent communicates all the possible application execution paths and their resource requirements up front, and receives in return (from the QoS arbitrator) a resource allocation profile for one of these paths.

*QoS arbitrator.* The QoS arbitrator takes advantage of the flexible program specification provided by QoS agents to enhance system utilization while satisfying the predictability requirements of each application. In MILAN, this flexibility comes from two aspects. First, application tunability provides the freedom to choose a resource allocation over time for each application. And second, the adaptiveness of the underlying fault-masking techniques (two-phase idempotent execution and eager scheduling) provide micro-level flexibility, permitting preemptive allocation, deallocation, and reallocation of resources to

each parallel step of the application. In this paper, we restrict our focus to the flexibility obtained from application tunability.

Upon job arrival, the QoS arbitrator first performs admission control to check whether or not application resource requirements can be satisfied. Application tunability increases the likelihood that an application can be admitted into the system. The QoS arbitrator scheduling algorithms (discussed in Section 6) first choose the best execution path, and then make an assignment of which processors will execute which application tasks and for what time. These decisions are communicated back to the application QoS agent which configures the application appropriately. In general, the QoS arbitrator also monitors system resources, and triggers renegotiation on detecting a significant change in resource levels (e.g., on a fault, or when new resources become available as in the metacomputing environment). For the purposes of this paper however, we assume that the underlying system is fault-free and has a fixed amount of available resources.

*Example.* In the example of the junction detection application, we annotate the source code to make it tunable. The annotated code goes through a preprocessor to generate the corresponding QoS agent. The QoS agent for this application represents tunability in terms of two parameters: the sampling granularity and a search distance metric which determines how regions of interest are constructed in the second step of the algorithm. The choice of different values for these two parameters controls the execution path. All the execution paths of this application form a task system, a DAG, which is annotated with per-path resource requirements, deadlines, and output qualities.

The QoS agent communicates this task system to the QoS arbitrator, which chooses the execution path that will be executed. Note that depending upon system load, different paths may be chosen for junction-detection jobs that arrive at different times. The QoS agent then configures the application to execute along that path. In this example, application configuration just requires setting values for the sampling granularity and search distance parameters.

## 6. PERFORMANCE IMPACT OF TUNABILITY

To understand the performance impact of tunability, we consider the system utilization achieved by tunable parallel Calypso programs running in a dedicated cluster environment managed by the MILAN resource management architecture. We first propose a greedy heuristic for scheduling tunable parallel applications with predictability requirements, and then using a parameterizable task system systematically quantify the benefits and shortcomings of tunability. The QoS arbitrator component of the resource management architecture described in Section 5 incorporates this heuristic.

### 6.1. Scheduling Formulation

Without tunability, the underlying scheduling problem we address is one of dynamically scheduling parallel real-time jobs in a system with a fixed amount of homogeneous processing resources. As described in Sections 4, we restrict our attention to jobs which can be represented as a chain of tasks where each task has an associated deadline. Each task is assumed to be non-preemptible. We study tunability benefits for both non-malleable (Section 6.3) and malleable (Section 6.4) executions of these tasks. Non-malleable tasks, characteristic of most current-day parallel applications written in a message-passing style using systems such as PVM [12] or MPI [14], require a fixed resource “shape” in terms

of the number of processors required over a time period. In contrast, malleable tasks such as in Calypso programs, can execute on any number of processors up to their degree of concurrency. In both cases, we assume that information about all tasks of the job is available upon job arrival. The primary objective of this scheduling problem is to maximize the number of on-time jobs. A secondary objective is to maximize system utilization.

With tunability, the only change to the scheduling problem is that a job is now represented by an OR task graph instead of a chain. The multiple paths in the task graph represent the various alternate executions of a tunable program. For uniformity, we assume that all paths through an OR graph have been enumerated, so a tunable application is represented by multiple task chains. For the purpose of this paper, we assume that each chain requires the same total amount of resources and achieves the same output quality value. Note that in practice, task chains of a tunable application are likely to have different overall resource requirements and output qualities: the issue then is of maximizing the achieved job quality.

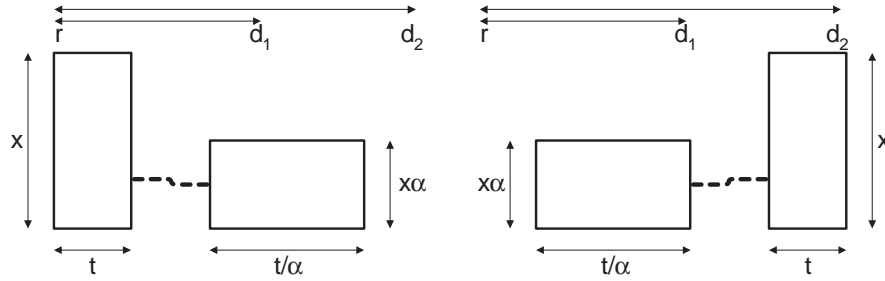
As with most non-trivial scheduling problems, all of the above formulations are NP-hard. Consequently, we next describe a greedy heuristic for the above scheduling problem.

*A Greedy Heuristic.* The heuristic allocates resources to jobs using a first fit policy. For a tunable job with multiple schedulable configurations, the heuristic finds among all of them the one that most efficiently uses the system. The heuristic keeps track of available *maximal holes* in the processor-time 2D space: each hole is represented by a triple  $(m, t_b, t_e)$  (denoting that  $m$  processors are available from beginning time  $t_b$  until the end time  $t_e$ ), and is maximal if it is not contained within any other hole. A job is *schedulable* if all the tasks on its task chain (any one of the task chains for a tunable job) can be scheduled into available holes while meeting the task deadlines. Ties between schedulable configurations are broken in favor of chains which maximize system utilization (over a time window defined by the job’s release time and scheduled finish time) and require fewer total resources for some prefix of their tasks. Under the assumptions of our task model, the heuristic finds the job configuration which achieves the earliest finish time.

## 6.2. A Parameterizable Task System

To systematically explore the space of application behaviors, we consider a task system that consists of a parameterizable tunable job shown in Figure 5. The job parameters enable the convenient simulation of a range of job shapes and deadline characteristics.

The job consists of two chains, each with two tasks. The two configurations simply transpose the positions of the two tasks. Each task requires the same total amount of resources but with different shapes. One task asks for  $x$  processors for time  $t$ , whereas the other task requests  $x\alpha$  processors for time  $t/\alpha$  amount of time. The value of  $\alpha$  is chosen in the interval  $(0, 1]$  such that both  $x$  and  $x\alpha$  are integers. Modifying  $x$  and  $\alpha$  allows the simulation of a variety of task shapes. The task deadlines are set in terms of another parameter, the *laxity* of the job, expressed as the ratio of the slack time in the time period from the release time to the deadline. For a job released at time  $r$ , the deadline of the first task is set to  $d_1 = r + \max(t, t/\alpha)/(1 - \text{laxity})$ ; the deadline of the second task is set to  $d_2 = r + (t + t/\alpha)/(1 - \text{laxity})$ . Since a task can begin execution as soon as its immediate predecessor completes, the task deadline denotes the time by which the task and all its predecessors must finish. The laxity parameter allows the systematic modeling of different amounts of slack time, and hence the constraints that exist for “fitting” a job into the system.



**FIG. 5.** A parameterizable tunable job. The job parameters  $x$ ,  $\alpha$ , and laxity, permit the convenient simulation of a range of job shapes and deadline characteristics.

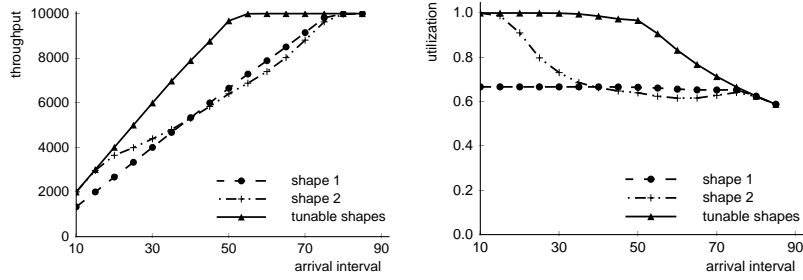
With the above parameterizable job, we construct three task systems: the first task system is tunable, consisting of both job configurations, while the other two systems are non-tunable, containing one configuration apiece. Jobs in each task system arrive according to the Poisson distribution. We quantify the benefits of tunability in terms of two metrics—*system utilization* and *job throughput*—measuring the performance of the tunable task system as compared to the non-tunable task systems as a function of four parameters: *mean arrival interval*, *laxity*, the *number of processors* in the system, and  $\alpha$  which controls the job shape. All experiments reported in this section assume  $x=16$  processors,  $t=25$  time units, and 10,000 job arrivals.

### 6.3. Tunability Benefits for Non-malleable Tasks

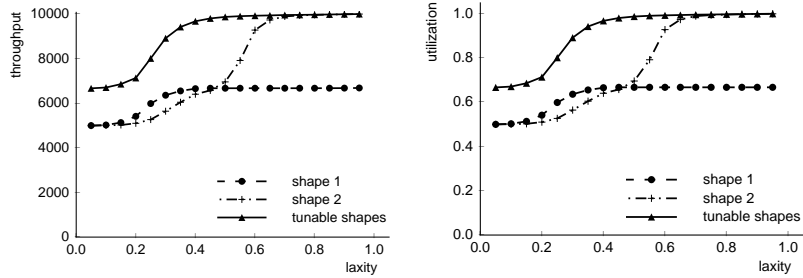
We first study the benefit of tunability for jobs consisting of tasks that require a fixed number of processors over a period of time. Figure 6 shows the system utilization (left) and throughput (right) as the mean arrival interval, the laxity, the number of processors, and the job shape are varied one parameter at a time, keeping the others fixed.

*Sensitivity to inter-arrival time.* In Figure 6 (a), the arrival interval varies from 10 to 85 units (note that  $t=25$ ), with the other parameters fixed as follows: *number of processors*=16, *laxity*=0.4, and  $\alpha=0.5$ . When the arrival interval is small, the system is overloaded and only a small portion of the tasks are admitted in all three systems. Tunability has negligible performance impact, since the system is already being fully utilized. When the arrival interval is very high, the system is underloaded and all three task systems can admit all the jobs. Tunability does not yield much benefit since resources are abundant compared to requests. It is in the middle range of arrival intervals however, that the tunable system achieves the largest improvement in both utilization and throughput: at its peak, it can admit 3000 more jobs and achieve 30% better system utilization. The tunable system can decide which of the task configurations to use based on resource availability, resulting an efficient utilization of resources.

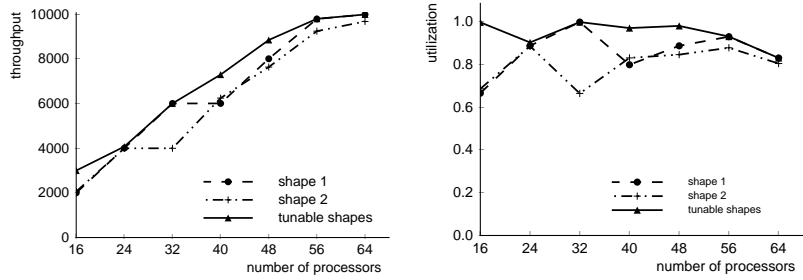
*Sensitivity to laxity.* In Figure 6 (b), the laxity varies from 0.05 (a slack time of 5%) to 0.95 (a slack time of 20 times the processing time), with the other parameters fixed as follows: *number of processors*=16,  $\alpha=0.5$ , *mean arrival interval*=50 time units. If there are no timeliness requirements (i.e., laxity is 1), all jobs would be admitted and the resources



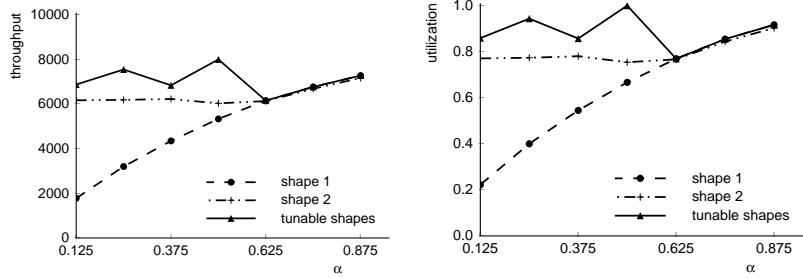
(a) Variation in average arrival interval



(b) Variation in laxity



(c) Variation in number of processors



(d) Variation in shape

**FIG. 6.** Performance impact of tunability for non-malleable tasks as average arrival interval, laxity, number of processors in the system, and job shapes are varied, in terms of throughput (left) and system utilization (right).

are fully utilized. In the presence of timeliness requirements, some jobs would be rejected because their deadlines cannot be met. When laxity is small, the job deadlines are tight and because there is little processor-time space left to fulfill resource requirements, the tunable system yields only a small improvement. However, this improvement goes up with an increase in laxity, decreasing to zero only when there is enough space to admit even non-tunable tasks. For shape 2, this happens when the laxity is above 60%. In contrast, shape 1 requires a larger number of processors for its first task, preventing it from a good packing (due to the greedy nature of the heuristic) even when deadlines are loose. The latter situation demonstrates the performance handicap of an inflexible (i.e., non-tunable) application.

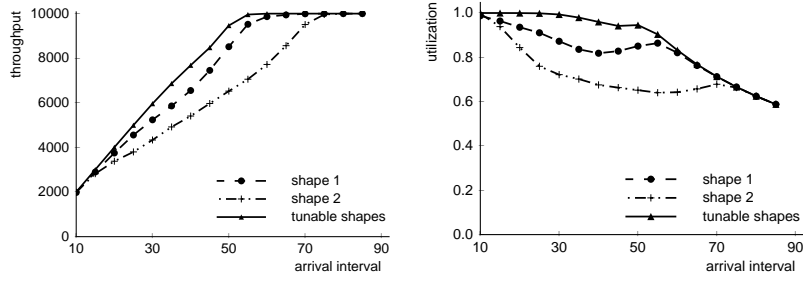
*Sensitivity to the number of processors.* Figure 6 (c) shows the benefits of tunability when the number of processors in the system are increased from 16 to 64 (recall that  $x=16$ ), with the other parameters fixed as follows:  $\alpha=0.5$ , *mean arrival interval*=15 time units, and *laxity*=0.25. Throughput increases as more processors become available. However, for some values, the non-tunable systems fail to gain from more processors, resulting from a drop in resource utilization. In these cases, the packing of job shapes does not fit the resource capacity well and the extra resources are simply wasted. The tunable system diminishes the penalty of this effect, yielding robust performance. When the number of processors grows significantly larger than individual task concurrency, even non-tunable jobs are able to utilize the available resources well; consequently, the benefits from tunability decrease.

*Sensitivity to the job shape.* Figure 6 (d) shows the benefits of tunability as a function of the job shape, determined by  $\alpha$ . The other parameters are kept fixed as follows: *number of processors*=16, *mean arrival interval*=40, and *laxity*=0.5. The utilization achieved by a particular job shape depends upon how well the shape can be packed given the number of processors in the system. Consequently, even when two job shapes have the same total resource requirement, they can yield very different overall system utilizations because one of the shapes might be a better fit. We see that shape 1 performs far worse than the other two when  $\alpha$  is small. Due to the suboptimal nature of the heuristic, a lot of resources are wasted for shape 1 jobs, proportional to the value of  $\alpha$ . When  $\alpha$  is not too large (up to 0.625), tunability improves performance achieving better packing than either of the individual shapes. For values of  $\alpha$  that produce similar task resource profiles, as expected, tunability yields few benefits.

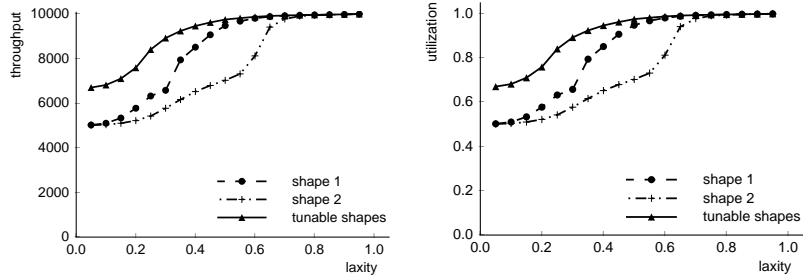
#### 6.4. Tunability Benefits for Malleable Tasks

We conducted the same set of experiments for tasks that can be executed in a malleable fashion. These experiments, whose results are shown in Figure 7, differ from the non-malleable execution model only in that, instead of requiring a fixed task shape, they permit the task to change its shape to any area-preserving rectangle with a height less than its degree of concurrency. When allocating resources to a malleable task, the heuristic is extended to try various configurations of the task, starting from the highest number of processors the task can use (which corresponds to the shortest execution time).

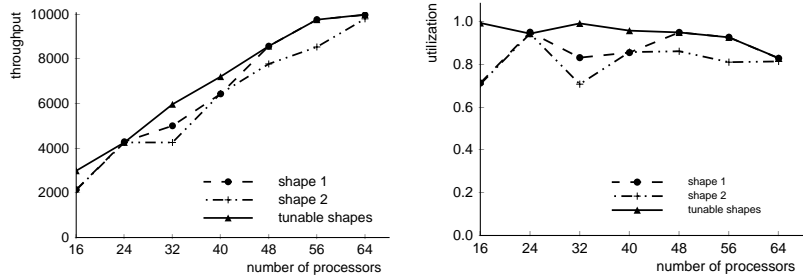
Comparing the results in Figure 7 and Figure 6, we find that both the tunable system and the non-tunable shape 2 job system perform similarly for both non-malleable and malleable tasks. What is noticeable however, is that the performance of the non-tunable shape 1 job system improves by a significant amount. Malleability ensures that these jobs achieve better packing as compared to their non-malleable versions, improving system utilization.



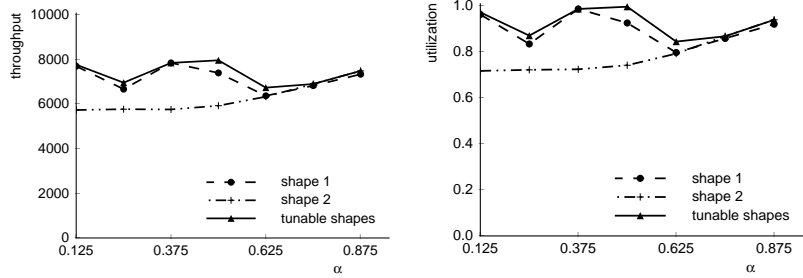
(a) Variation in average arrival interval



(b) Variation in laxity



(c) Variation in number of processors



(d) Variation in shape

**FIG. 7.** Performance impact of tunability for malleable tasks as mean arrival interval, laxity, number of processors in the system, and job shapes are varied, in terms of throughput (left) and system utilization (right).

Based upon the results shown in Figure 7, the same conclusion can be drawn regarding the overall benefits of tunability for malleable tasks as for non-malleable tasks, although the magnitude of the benefit might be smaller. In some situations, the sub-optimality of the heuristic prevents the tunable system from performing as well as one of the non-tunable systems. For example, in Figure 7 (d) when  $\alpha=0.375$ , shape 1 packs slightly better than the tunable system, although the difference is negligible.

The reduced magnitude of improvement from tunability for malleable tasks might suggest that support for malleable tasks alone might be sufficient to improve overall system utilization. While this is certainly true for a range of parameter values, tunability complements malleability over a large range. Figures 8 (a) and 8 (b) show the increased throughput attained by the tunable job system when compared with the non-tunable shape 1 and shape 2 job systems for non-malleable and malleable tasks respectively as the mean arrival interval and laxity parameters are varied. The other parameters are kept fixed as follows: *number of processors*=16 and  $\alpha=0.5$ . We find that for a large range of parameter values, the performance benefits from tunability *augment* those resulting from malleable task execution. This expected behavior is explained by the observation that tunability denotes flexibility in resource utilization both at the individual task level as well as at the global application level. In contrast, although malleability might allow a particular task to execute in a flexible fashion, does not permit this flexibility to extend to the entire application. Thus, tunability complements malleability, exposing additional information about application resource requirements to the underlying resource management system.

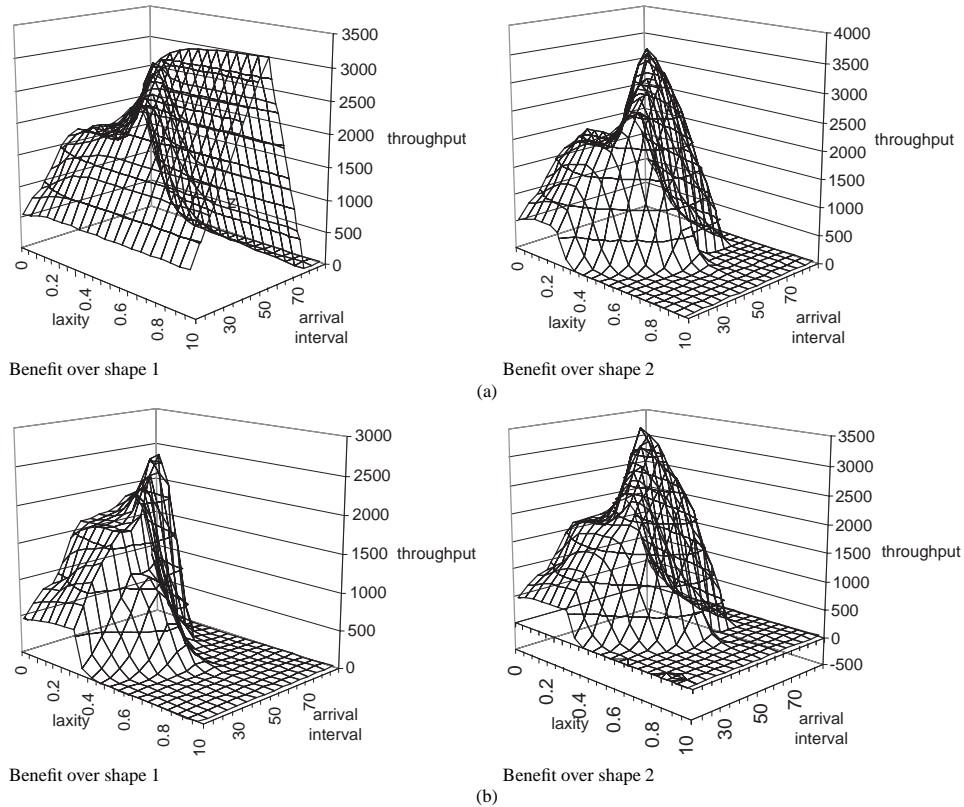
### 6.5. Summary

Our results show that for both non-malleable and malleable task executions, tunability yields substantial performance advantages, with respect to both system utilization and throughput. These improvements are most pronounced in the mid-portion of each parameter value range: when the system is moderately loaded, when the deadlines are not extremely loose, when the number of processors available is not significantly larger than the concurrency degree of a parallel task, and when the job shapes are not too similar. These system characteristics describe the operating point of most practical parallel systems, attesting to the significant performance potential of application tunability.

## 7. DISCUSSION AND RELATED WORK

In this paper, we have examined how to efficiently support parallel and distributed applications with predictability requirements in an environment with dynamically varying resource availability. The ability to efficiently support such applications ensures that general-purpose media-processing applications, which are likely to be one of the largest application domains in the coming years, can benefit from parallel and distributed execution. Our solution, centered on the notion of application tunability, captures the flexibility such applications exhibit with respect to resource requirements. Tunability yields benefits both for the system and the application. While we have demonstrated it in the specific context of predictable parallel computations, the idea of increased flexibility in application specification is relevant in most resource management scenarios; the flexibility can be viewed as increasing an application's likelihood of obtaining good performance with a fixed set of resources.

Predictability for applications has been extensively studied in the real-time systems literature. Several well-known scheduling schemes, such as rate-monotonic [19, 25] and



**FIG. 8.** Performance impact of tunability in non-malleable model (a) and in malleable model (b), as job arrival interval and laxity are varied.

earliest-deadline-first (EDF), exist for scheduling real-time tasks in an uniprocessor environment. Gillies [13] has studied the scheduling of tasks in AND/OR graphs. However, the validity of most of these results is restricted to either uniprocessor systems or a sequential task model. Our interest is in managing parallel real-time tasks in a parallel system: in this scenario, well-known results such as EDF no longer provide the optimality guarantees they do in sequential systems.

As described earlier, the primary focus of almost all parallel resource management systems is on optimizing system utilization, often in the context of relatively static resource requirements. The Distributed Resource Management System (DRMS) [22] is an exception, providing system support and resource management for dynamic reconfiguration of parallel programs (with respect to the number of tasks and mapping of tasks to processors). This reconfiguration is closely related to our notion of application tunability; however, unlike the focus of this paper, DRMS exploits the reconfigurability primarily to optimize system utilization while still providing only best-effort guarantees to the application.

Our work is probably most closely related to a few recently-started projects [23, 18, 5, 26, 20, 15, 9] that are looking into the problem of adapting application behavior in response

to variations in system resources. Despite the shared goals, our approach differs from these projects in several important ways:

The Darwin project [23] permits the application to specify its resource requests in the form of a virtual mesh of nodes representing desired services and edges denoting communication flows. The Darwin resource manager places the nodes and route the flows, optionally inserting semantic-preserving transformations (e.g., transcoders between JPEG and MPEG formats) to optimize cost or quality. While the latter transformations are somewhat related to our notion of alternate execution paths, our approach differs in completely insulating the system-level resource manager from the need to be aware of any application-specific information.

The ActiveHarmony [18] and AppLeS [5, 26] projects provide application-level mechanisms and resource monitoring tools to enable an application to adapt itself to changing resource properties. In both systems, such adaptation is achieved by modifying the placement and partitioning of application entities such as threads and data sets on physical system resources. The latter can be considered a very restricted form of tunability, capturing execution paths where application entities are mapped differently but to the same kind of physical resources. Our notion of tunability is much more general, enabling the application to trade off its resource requirements over several dimensions including time, output quality, and resource type.

The EPIQ [20, 15] and ErDos [9] projects are closer to our approach in that they look into the quality aspects of an application, trading off output quality against resource requirements. Both EPIQ and ErDos take the view that the quality of an execution degrades when there are insufficient resources. Our work stresses the tunable aspects of applications, especially the ability to tradeoff resources over time with the same QoS goal. Moreover, the projects emphasize different aspects of the overall problem. EPIQ focuses on a general framework for specifying and negotiating QoS values, and ErDos proposes a recursive model for dynamically structuring QoS-aware applications from individual, relatively independent, modules. In contrast, we adopt a simpler model of resource versus quality tradeoff, focusing instead on the programming language-level expression of such tradeoffs and algorithmic issues surrounding the use of these tradeoffs to improve system performance.

## 8. CONCLUSION

We have presented a novel approach for improving parallel and distributed system utilization while simultaneously meeting the predictability requirements of networked applications such as image recognition, media processing, and virtual reality. Our approach takes advantage of *application tunability*, which refers to a flexibility in the application specification particularly with respect to its ability to trade off resource requirements over several dimensions including time, output quality, and resource type. A resource management system can exploit this flexibility to decide how best to allocate resources to predictable computations so as to maximize both job throughput and overall system utilization. We have described language extensions to the Calypso parallel programming language for expressing tunability, and evaluated its performance impact using a parameterizable task system to systematically identify its benefits and shortcomings. Our results show that application tunability is convenient to express, and can yield significant performance benefits.

## ACKNOWLEDGMENT

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under grant number CCR-9411590 and CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

## REFERENCES

1. A. Baratloo, P. Dasgupta, V. Karamcheti, and Z. Kedem. Metacomputing with MILAN. In *Proc. 8th Heterogeneous Computing Workshop*, 1999.
2. A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.
3. A. Baratloo, A. Itzkovitz, Z. Kedem, and Y. Zhao. Mechanisms for just-in-time allocation of resources for adaptive parallel applications. In *Proc. 13th Intl. Parallel Processing Symposium*, 1999.
4. A. Baratloo, M. Karaul, Z. Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems*, Sep. 1996.
5. F. Berman and R. Wolski. Scheduling from the perspective of the application. In *Proc. 5th IEEE Intl. Symp. on High Performance Distributed Computing*, 1996.
6. K. Chandy and C. Kesselman. A description of CC++. Technical Report CS-92-01, California Institute of Technology, 1992.
7. E.C. Chang and C. Yap. A wavelet approach to foveating images. In *Proc. 13th ACM Symp. on Computational Geometry*, 1997.
8. E.C. Chang, C. Yap, and T.-J. Yen. Realtime visualization of large images over a thinwire. In *IEEE Visualization*, 1997.
9. S. Chatterjee. Dynamic application structuring on heterogeneous, distributed systems. In *Proc. IPPS/SPDP'99 Workshop on Parallel and Distributed Real-Time Systems*, 1999.
10. D. Culler et al. Parallel computing on the Berkeley NOW. In *Proc. 9th Joint Symposium on Parallel Processing*, 1997.
11. P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: Fault-tolerant high performance approach. In *Proc. 15th IEEE Intl. Conf. on Distributed Computing Systems*, 1995.
12. G. Geist and V. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4), 1992.
13. D. Gillies and J. Liu. Scheduling tasks with and/or precedence constraints. In *Proc. 2nd IEEE Conf. on Parallel and Distributed Processing*, Dec. 1990.
14. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994. ISBN 0-262-57104-8.
15. D. Hull, A. Shankar, K. Nahrstedt, and J. Liu. An end-to-end QoS model and management architecture. In *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, California, Dec. 1997.
16. H. Ishikawa and D. Geiger. Segmentation by grouping junctions. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 1998.
17. Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990.
18. P. Keleher, J. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, Jun. 1999.
19. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of ACM*, 20(1), Jan. 1973.
20. J. Liu, K. Nahrstedt, D. Hull, S. Chenand, and B. Li. EPIQ QoS characterization. Draft, Jul. 1997. <http://pertsserver.cs.uiuc.edu/epiq>.
21. The MILAN project: Metacomputing in large asynchronous networks. <http://www.cs.nyu.edu/milan>.

22. J. Moreira and V. Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM J. of Research & Development*, 41(3), 1997.
23. P. Chandra et al. Darwin: Customizable resource management for value-added network services. In *Proc. Sixth IEEE Intl. Conf. on Network Protocols*, 1998.
24. S. Sardesai and P. Dasgupta. Chime : A versatile distributed parallel processing environment. <http://www.cs.nyu.edu/milan>.
25. L. Sha and J. Lehoczky. Performance of real-time bus scheduling algorithms. *ACM Perform. Eval. Rev.*, 14(1), 1986.
26. N. Spring and R. Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *Proc. 12th ACM Intl. Conf. on Supercomputing*, Australia, 1998.
27. L. G. Valiant. A bridging model for parallel computation. *Communications of ACM*, 38(8), Aug. 1990.

## BIOGRAPHIES

FANGZHE CHANG is a Ph.D. candidate in computer science at Courant Institute of Mathematical Sciences, New York University. His current research focuses on distributed computing, especially system support for application composition, configuration, and adaptation in dynamic heterogeneous environments. Fangzhe received his bachelor's degree from Changsha Institute of Technology, his master's degree from Institute of Software, Academia Sinica.

VIJAY KARAMCHETI is an assistant professor of computer science in the Courant Institute of Mathematical Sciences at New York University. From 1992 to 1997, Vijay was involved in the Concert System and Fast Message projects at the University of Illinois. His current research focuses on system support for adaptable parallel and distributed computing in dynamic heterogeneous environments. Vijay received his undergraduate degree from the India Institute of Technology, Kanpur, his master's from the University of Texas, Austin, and his doctoral degree from the University of Illinois at Urbana-Champaign. He is a recipient of a 1999 National Science Foundation CARREER Award.

ZVI KEDEM is a professor of computer science in the Courant Institute of Mathematical Sciences at New York University, where he previously also served as the chair fo the department of computer science. He has conducted research in theory and algorithms, computer graphics, database systems, parallel computing, and distributed systems. He got his bachelor's, master's, and doctoral degrees from the Technion - Israel Institute of Technology. He is a winner of the IEEE Outstanding Paper Award and an ACM Fellow.