On Adaptive Resource Allocation for Complex Real-Time Applications*

Daniela Roşu, Karsten Schwan, Sudhakar Yalamanchili
Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30332-0208
{ daniela,schwan }@cc.gatech.edu
sudhakar.yalamanchili@ee.gatech.edu

Rakesh Jha
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN-55418
jha@src.honeywell.com

Abstract

Resource allocation for high-performance real-time applications is challenging due to the applications' data-dependent nature, dynamic changes in their external environment, and limited resource availability in their target embedded system platforms. These challenges may be met by use of Adaptive Resource Allocation (ARA) mechanisms that can promptly adjust resource allocation to changes in an application's resource needs, whenever there is a risk of failing to satisfy its timing constraints. By taking advantage of an application's adaptation capabilities, ARA eliminates the need for 'over-sizing' real-time systems to meet worst-case application needs. This paper proposes a model for describing an application's adaptation capabilities and the runtime variation of its resource needs. The paper also proposes a satisfiability-driven set of performance metrics for capturing the impact of ARA mechanisms on the performance of adaptable real-time applications. The relevance of the proposed set of metrics is demonstrated experimentally, using a synthetic application designed to represent time-critical applications in C3I systems.

1. Introduction

Motivation. Resource management problems for real-time and embedded applications are exacerbated by dynamic changes in their external environments and by restrictions on resource availability. Solving such problems by using worst-case needs analysis [6] is typically not viable because of excessive resource estimates resulting from complex application behavior. Instead, adaptive methods [3, 10, 11] must be used to adjust resource allocation to changes in an application's needs and to insure the satisfiability of its real-time constraints.

*Funded in part by DARPA through the Honeywell Technology Center under Contract No. B09332478 and Contract No. B09333218, and by NSF equipment grants CDA-9501637, CDA-9422033 and ECS-9411846.

Contributions. This paper describes and evaluates models and mechanisms for *Adaptive Resource Allocation* (ARA) in the context of high performance, embedded applications. We consider applications with data-dependent behavior, driven by event streams, and composed of multiple, possibly parallel interacting components. Runtime changes in event rates and more importantly, in the data content of these events cause significant changes in the resource needs of various application components. For such applications, it is difficult to closely estimate their worst-case event processing and communication needs. This class of applications includes radar systems [16], robot control [21], target recognition, multi-object tracking, and hypothesis testing [15].

ARA mechanisms can be used to promptly adjust resource allocation to changes in an application's resource needs, whenever there is a risk of failing to satisfy the application's timing constraints. Such runtime adjustments, constraint by the application's adaptation capabilities, eliminate the need for 'over-sizing' real-time systems to meet worst-case application needs.

This paper describes a novel model for capturing an application's adaptation capabilities by specifying the resource needs corresponding to each of its acceptable configurations. In addition, the model captures the runtime variation of the application's resource needs.

Given the real-time nature of the applications targeted by this research, we propose to evaluate ARA mechanisms by their impact on the satisfiability of the applications' real-time constraints. We submit that it is essential to consider the latencies with which ARA mechanisms respond to changes in application needs when attempting to restore the satisfiability of real-time constraints. Namely, the quality of ARA decisions should be evaluated with respect to both how fast an application returns to some acceptable performance and how good its performance is in steady state compared to the one imposed by the application's real-time requirements. In response, this paper identifies several elements that contribute to the effectiveness of ARA methods for detecting changes in resource needs and for making resource allocation decisions.

Assumptions and Experimental Environment. This work assumes a multi-machine environment used by a single, complex application. As a result, performance perturbations are produced only by dynamics in the application's external environment or by changes in resource availability due to failures or explicit removals/additions. Explicit admission control guarantees sufficient resources for meeting the application's initial needs.

The models and heuristics proposed here are evaluated in the context of a centralized ARA controller. Online monitoring is performed with the mechanisms described in [8]. Experiments are conducted with a synthetic application running on a cluster of workstations. The application is designed by Honeywell in the context of high performance C3I[1] applications [15].

Related research. Previous work has described frameworks and mechanisms that facilitate the creation and use of online adaptation heuristics for real-time applications [3, 11, 13], including mechanisms for runtime monitoring, adaptation enactment, and mechanisms that ensure the reliable execution of applications [3, 13] or that maintain high application throughput [11]. In comparison, the focus of this paper is not to define new frameworks, but instead, to define models and methods to be used in such frameworks and to analyze their effect on adaptive applications.

Extensive research has addressed the problem of dynamic resource allocation for both the real-time [1, 9, 10, 18, 22] and the non-real-time [7, 14, 17] domains. The methods developed in these studies do not fit our target application model, because our model assumes that the resource needs of a time-constrained task, even when generated by the same type of event, may vary throughout the execution of the application. This variability prevents us from using a periodic task model [9, 10] in which performance requirements are fixed throughout an application's execution, and therefore worst-case needs have to be considered. It also prevents us from using a sporadic task model, as in the real-time [18, 22] or the non-real-time [7, 14] domains, because of the high overhead of taking resource allocation actions at each task arrival.

Resource reallocation triggered by runtime variation of application needs has received less attention. Previous schemes proposed for both real-time [10, 20] and non-real-time [11, 14, 17] domains do not consider the transitory effects of adaptation mechanisms on the satisfiability of application's performance constraints. Their primary interest is to attain optimal average-case performance.

Overview of paper. In the remainder of this paper, we first identify the application and the ARA model driving our research (Section 2). In Section 3, two important components of the application model used for ARA are described: the application resource usage model and the application adaptation model. Section 4 identifies specific ARA performance criteria derived from the real-time nature of our target applications. Last, in Section 5, we demonstrate by experiments the relevance of these criteria and identify methods that help improve ARA performance.

[1]Command, Control, Communications and Intelligence

2. Real-Time Applications and ARA

Application Model. Our research targets reactive, high performance applications that must meet well-defined real-time constraints in dynamic execution environments. Each such application consists of multiple interacting *components* capable of executing in a distributed environment consisting of parallel machines, embedded-system components (e.g., signal processors), and user interface stations (e.g., workstations). Components are either sequential or parallel tasks and their resource needs may be data-dependent, varying with changes in the rate or content of data inputs. In response, many components are programmed such that they can adapt their resource needs at runtime, by changes in their execution mode, algorithms or specific attributes such as the level of parallelism or communication protocols.

An application's execution is driven by *event streams* produced by the external environment or the application itself. Each event stream is processed by a fixed set of components, with fixed precedence constraints described by a *communication graph*. The input pattern of a stream may vary with changes in the execution environment. In the following, the communication among parallel modules of the same application component is called *intra-communication*, and the communication between a component and its neighbors in the communication graph is called *inter-communication*. We assume that, for each event, the intra-communication happens throughout the event processing while the inter-communication happens in a burst, at the end of the event processing.

The application's *performance requirements* are defined by constraints with respect to event rate, end-to-end latency, and component relative completion times. Each timing constraint has specific miss rate and miss burst bounds.

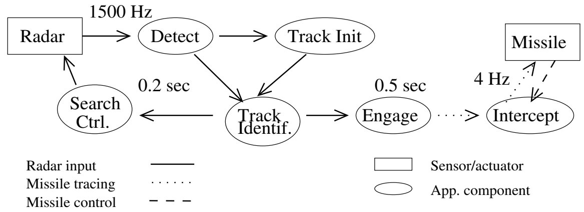

Figure 1. Radar Application

Sample Application. One sample application driving this research is a radar system. Figure 1 presents part of such a system, as described in [16]. *Detection*, *Track Init*, and *Track Identif* are computationally intensive tasks, each well suited for parallel implementation [15]. Over time, their processing and communication needs vary with the number and characteristics (e.g., amplitude, direction) of dwells. Given the nature of their computation [15], these tasks can adapt by changing their internal levels of parallelism.

The main event streams in the radar system are (1) the input from the radar, (2) the input from the missile tracking device, and (3) the missile control requirements. Timing constraints concern necessary event rates and processing latencies. For instance, the required rate of the radar input is 1500Hz, and the required missile control rate is 4Hz. Additional constraints are: a 0.2 second-bound on the latency between *Detect*-ing a potential missile and engaging *Search Control*, and a 0.5 second-bound on the execution of *Engage*.

The radar system is one of many applications concerned with processing signals from a sensor suite, forming hypotheses about and assessing the situation, and taking an appropriate response based on data observed and processed over a period of time. Other examples are multi-hypotheses tracking and image understanding [15]. Often the front end of these applications consists of signal processing stages whose computational needs are predictable, as they are independent of the signal values. However, computations at the back end depend on the semantic content of the signal values, being often heavily data-dependent.

Specific Resource Allocation Problems. The application model presented above raises interesting resource allocation problems. First, the event stream-based execution makes viable the option of using long term resource allocation. Alternatively, a short term resource allocation based on dynamic real-time scheduling decisions [18, 22] is prone to add a large overhead to each event's processing.

Second, worst case-based allocation may not be appropriate for our target applications. In the context of data-dependent resource needs, it is difficult to evaluate the worst case needs with sufficient accuracy to ensure both a safe execution and acceptable resource utilization. For example, in the radar system (see Figure 1), the resource needs of *Track Init* are highly data-dependent as they vary with the number of dwell returns above a selected threshold and the ambiguity of spurious tracks. Therefore, the worst case needs depend on the worst case execution scenarios, which makes them hard to evaluate.

Our solution to these problems is to use adaptive resource allocation (ARA). By taking advantage of the application's adaptation capabilities, ARA permits using long-term resource reservations [12] while accommodating runtime changes in resource needs.

Adaptive Resource Allocation. ARA is a resource management paradigm that takes advantage of an application's runtime adaptation capability in order to accommodate its dynamic resource needs and to satisfy the system goals with respect to performance and resource utilization. In the context of our target application model, the goal of ARA is to insure that, at any time, the performance requirements of the application are satisfied.

An ARA infrastructure can satisfy two types of resource requests: *explicit* and *implicit*. An explicit request is issued by the application upon a new component arrival or whenever the application deems it necessary to adjust resource usage. An implicit request is issued by the ARA infrastructure itself, when changes in a component's resource usage considerably increase the likelihood of failing to satisfy the application's performance requirements.

Implicit requests, and sometimes also explicit ones, are satisfied by adjustments of the resource allocation of one or more application components, that are decided by the ARA infrastructure itself. Such adjustments are called *automatic* because they are not explicitly requested by the application. They are performed only to prevent violations of the application's performance and they are constraint by the application/component-specific adaptation capabilities. For example, an automatic adjustment may be performed when, due to the lack of resources, a new application component cannot be accommodated unless other components' allocations are reduced. Similarly, an automatic adjustment may be triggered by an unexpected change in the execution environment that causes a change in resource needs which cannot be accommodated in the current configuration. One example is a change in the input data content that causes an increase of event processing time for a particular component. This change might require extending the component's level of parallelism in order to meet the required event rate.

Our approach to resource management attempts to move part of the burden of making adaptation decisions from the application to the resource management infrastructure. A similar approach is taken in [10, 11] and, also, in our previous work [20]. The benefits of this approach are a reduction in application perturbation plus the fact that unexpected changes in the application's resource needs are likely to receive fast response. This is due to the resource management infrastructure's fast access to all the necessary information related to resource availability and current resource usage patterns of application components. A drawback is that, compared to application-level decisions [3], ARA decisions may fail to produce the most appropriate resource assignment for each particular situation. Likewise, ARA may result in resource allocation changes not necessary for achieving acceptable application performance. The models and mechanisms embedded in an ARA infrastructure mitigate these potential drawbacks.

In order to achieve its functionality, the ARA infrastructure should include mechanisms for: (1) collecting information about application resource usage and resource avail-

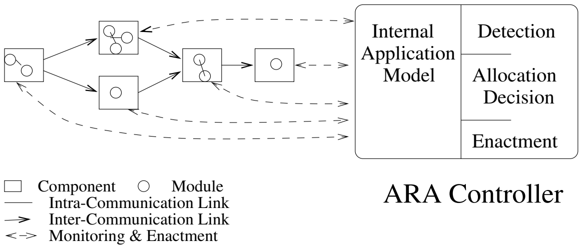

Figure 2. Centralized ARA controller

ability; (2) detecting significant variations in application resource usage; (3) inferring the cause of observed variations and assessing the necessity of an automatic adjustment of the resource usage; (4) making decisions about automatic adjustments and resource allocation; (5) notifying the application about significant changes in its resource usage; (6) notifying the application and resource providers about changes in resource allocation; and (7) assisting them in the enactment of these changes.

ARA is based on knowledge of the application's characteristics. These characteristics are described by an *application model* internal to the ARA infrastructure. Besides the structure of the application (components, event streams, communication graphs) and its performance requirements, this model describes, for each application component, the *acceptable configurations* (i.e., those instances of resource allocation that permit the component to perform correctly) and the runtime variation of its resource usage. The model is used to interpret the monitored information, to estimate the system performance expected upon changes in resource allocation, and to guide the decision heuristics.

The performance of the ARA is determined both by the appropriateness of its resource allocation decisions and by the delay with which it responds to unexpected changes in application behavior. A short response time helps reduce the duration of intervals in which the application fails to satisfy its performance constraints. Delayed ARA decisions, large decision times, or decisions with high enactment overhead are less likely to help the application cope with its immediate performance constraints.

In our work, ARA functionality is provided by a module called the *ARA controller*. This module may have a distributed or a centralized architecture. Figure 2 depicts a centralized controller, similar to the one used in our experiments. The controller's interaction with the application is restricted to monitoring and allocation enactment.

In the next sections, we address the internal application model and the performance evaluation of an ARA infrastructure.

3. Internal Application Model

This section describes the first contribution of our research. We propose models that describe the application's resource usage and its adaptation capabilities. These models are part of the application model internal to the ARA infrastructure:

- The *resource usage model* (RUM) describes an application's expected computational and communication needs, and the runtime variation of both.
- The *adaptation model* (AM) describes an application's acceptable configurations in terms of expected resource needs and application-specific configuration overheads.

The RUM is used in the ARA decision making process to evaluate the current application's resource needs and to determine how its performance requirements will be satisfied. The AM permits the ARA controller to decide appropriate resource allocation adjustments without incurring any negotiation overhead. In addition, the knowledge of configuration overheads permits the ARA controller to understand and evaluate tradeoffs between alternative adaptation strategies.

3.1. The Resource Usage Model

Background. The *resources* available to an application are nodes and the communication links between them. A node is characterized by its speed (MIPS or MFLOPS) and the size of its local memory. Each node uses a scheduling policy able to guarantee the resource reservations and to provide feedback to the application on its actual resource usage, such as those proposed in [12]. A communication link provides a unidirectional connection between two nodes. It is characterized by one or more protocols (e.g., reliable, FIFO unreliable), with known available bandwidth and cost of I/O operations at each end-point – a constant per-message overhead and a per-byte overhead. For simplicity, we currently consider only uniprocessor nodes.

Model Formulation. The RUM describes the resource needs of each (component, event stream)-pair. In the following, such a pair is called "a component".

Each component is described as an internally parallel task, with multiple cooperating modules that are independent from the point of view of resource allocation. The component's resource needs are described by two models – *static RUM* and *dynamic RUM*. The static RUM describes the expected computation and communication needs, while the dynamic RUM captures the runtime variation of these needs with respect to the static RUM.

The parameters of the static RUM are the following:

- parallelism level,
- execution time,

- intra-communication protocol,
- maximum outgoing intra-communication message size,
- total number of outgoing intra-communication messages,
- total amount of outgoing intra-communication data,
- inter-communication protocol,
- total number of outgoing inter-communication messages,
- total amount of outgoing inter-communication data, and
- processor speed factor.

A set of inter-communication related parameters is defined separately for each successor in the communication graph.

The static RUM is specified by the application as part of an explicit request for resources. Its parameters may be estimated using traditional approaches like algorithm analysis or code profiling. The processor speed factor describes the performance of the node used for profiling.

Each parameter of the static RUM is assumed to be the largest value over the corresponding parameters of all of the component's modules. This is equivalent to assuming that all modules have identical resource needs, the intra-communication between any pair of modules is identical, and a module's incoming intra-communication is the sum of all messages sent by all other modules.

The *dynamic* RUM refers to those parameters of the static RUM that are likely to vary at runtime due to unexpected changes in input data content. This model is described by:

- execution factor,
- total amount of intra-component data factor,
- maximum intra-component message size factor, and
- total amount of inter-component data factor.

Each factor represents the ratio between the static RUM specifications and the maximum monitored value of the corresponding metric over an application-specific time interval. The dynamic RUM is maintained by the ARA controller based on monitoring data received from the application.

Model Discussion. The static RUM can be easily extended to describe the needs of each module of the application component or to include other resource types such as memory.

Given the static RUM, the ARA infrastructure can obtain a good estimate of each component's computation and communication needs. It may then use this information, together with information on the event's input pattern and on the component's deadline, to perform per-resource schedulability analysis and reservations. The component's computation needs include its execution time and its communication related computation. The latter is estimated based on the number of I/O operations and the total amount of data transferred. The communication needs result directly from the model. In contrast to typical real-time connection models [2], we ignore the intra-communication burst because it influences only the memory requirements on the nodes and network routers. For a node, such needs may be described by adding a memory parameter to the static RUM, and for the network, by specifying a 'maximum message size' large enough to cover the maximum burst.

The dynamic RUM permits the ARA controller to make appropriate automatic adjustments even when the observed resource needs are larger than those specified by the application. Such a situation may occur when the static RUM does not describe worst-case needs, either because it is not possible to estimate them accurately or because the programmer decided it, possibly due to the small likelihood of runtime behavior in which worst-case needs arise.

The information needed to maintain the dynamic RUM is obtained with low monitoring overhead from the instrumentation of the communication library.

Related Work. The resource usage model introduced here improves upon the deficiencies of real-time task models used in previous research [6] that do not permit a low-complexity description of a parallel component. According to such models, a parallel application component is described by a set of tasks with precedence constraints, each with fixed computation and communication needs, and with the I/O operations occurring only at the beginning and the end of a task (or event) execution. This would require each parallel component to be decomposed into multiple, small granularity tasks which leads to significant increases in ARA decision-making overheads. Instead, our ARA approach advocates using a RUM with reduced levels of detail and low decision overheads, yet able to provide good estimates of application performance.

The RUM also improves on previous parallel task models used in load balancing or task assignment problems [10, 4, 17] that describe the communication needs only by the time taken to perform it. By providing a more detailed description, the RUM can better estimate communication related resource needs in terms of multiple resource types and in a heterogenous environment.

3.2. Adaptation Model

Background. Each adaptive application component has several acceptable configurations. In general, the overhead of instantiating a new configuration has an application-independent and an application-dependent part. The application-independent part includes the overheads of starting-up a new parallel module and of reserving resources (on the host and in the network). The application-dependent part, henceforth *adaptation overheads*, are determined by the component-specific reconfiguration procedures. We assume that these overheads are primarily due to state transfers and initializations, and are significant when switching between configurations with different levels of parallelism.

Model Formulation. The adaptation model describes the acceptable configurations and the corresponding adaptation overheads for each (component, event stream)-pair.

An acceptable configuration is described by: (1) a *config-*

uration id, which is used by the ARA infrastructure to notify the application about changes in its resource allocation; (2) a *static RUM*, which specifies the resource needs as described in Section 3.1; and (3) *adaptation overheads,* which describe the module start-up and shut-down procedures. For each procedure, the adaptation overhead is described by the amount of state to be transferred and by the execution time (excluding communication).

The adaptation model is specified by the application upon an explicit request for resources. For each application component, several acceptable configurations may be described. The ARA assumes that the static RUMs for all configurations in an adaptation model are compatible, in the sense that all describe the resource needs for solving the same problem, but in different configurations.

Model Discussion. The knowledge of the acceptable configurations permits automatic adjustments of a component resource usage without negotiation. The adaptation overhead permits the ARA infrastructure to estimate the enactment overheads and their effects on the application performance.

Related Work. Our model is different from other schemes [1] that allow the application to specify a set of acceptable configurations at resource request time, by its description of adaptation overheads. The current model does not permit the specification of the "value" each particular configuration brings to the application, as in [1]. This is motivated by the current goal of our ARA: to satisfy the application's performance requirements with no concern for the overall application "value". However, such mission level information may be easily added to the model.

3.3. Using the Models

We briefly describe how the RUM and the adaptation model are used by the ARA controller (see Section 2). Details appear in [19].

The application requests an initial resource allocation by specifying an adaptation model. The ARA controller receives this request, and based on current resource availability, it chooses an acceptable configuration, performs the corresponding reservations and notifies the application.

At runtime, each component is described by a *current RUM*. The current static RUM corresponds to the acceptable configuration selected by the last allocation decision. The current dynamic RUM is maintained based on the current static RUM and monitoring information.

When the performance requirements are not satisfied or are considered to be too close to the acceptable threshold, the ARA controller may decide to adjust the application's resource allocation. During this decision, for each component, the static RUMs of its acceptable configurations are scaled by the corresponding dynamic RUM parameters. If some component experiences current usage larger than its current static RUM, the scaled static RUMs describe needs larger than the initial specifications, thereby prone to better fit with the application's current behavior. If the current needs of some component are lower than its specifications, the scaled static RUMs describe reduced needs, thereby enabling the ARA controller to evaluate the amounts of unused resources and to use them for providing other components/applications with better service.

4. ARA Performance Characterization

The formulation of suitable resource usage and adaptation models is the first contribution of our research. A second contribution is our proposal of a satisfiability-driven approach to evaluating the performance of an ARA infrastructure. This is in contrast to the optimality-driven approaches used in past research [10, 11]. In the context of a real-time application, we claim that the reactivity of an ARA infrastructure is often more important than the optimality of its decisions. In addition, each ARA decision instance is equally important to the application, so that we do not consider it appropriate to measure ARA performance by averages over a large set of instances.

Our experiments show that delays in adjusting resource allocation to changes in application behavior increase the time interval during which the application is exhibiting unacceptable performance. Since optimal decisions are likely to be associated with large decision and enactment overheads, such decision making also increases the likelihood of failing to satisfy the application's timing constraints. For instance, in a heterogeneous distributed system, an optimal minimization of end-to-end latency may require migrating all or many application components to more appropriate nodes. Such a reallocation decision may not be appropriate if during the enactment more events than acceptable miss their deadlines.

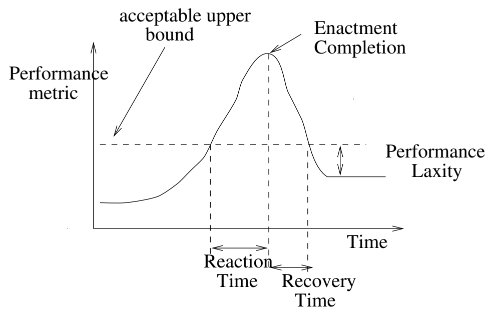

Figure 3. Metrics for ARA performance

Focusing on the satisfiability of an application's performance requirements, we evaluate the performance of the

ARA infrastructure (on short, ARA performance) by its response to a *single* variation in the application behavior that increases the risk of violating a performance requirement, called *critical variation*. Specifically, we consider the following metrics (see Figure 3):

- *reaction time* – the interval between the occurrence of a critical variation and the completion of the correcting reallocation enactment;
- *recovery time* – the interval between enactment completion and the restoration of an acceptable performance level; and
- *performance laxity* – the difference between the required performance and the steady state performance after reallocation.

A good ARA controller is expected to exhibit low reaction time, low recovery time, and large performance laxity.

These metrics reflect the performance of ARA mechanisms. Namely, recovery time and performance laxity relate to the quality of ARA reallocation decision, while reaction time captures the effectiveness of detection, decision, and enactment mechanisms.

The proposed set of metrics is relevant for a real-time application. Large reaction and recovery times increase the time interval during which the application's performance constraints are not satisfied. Low performance laxity increases the risk of failing to satisfy these constraints. The experiments in the next section demonstrate the relevance of the reaction time metric.

None of the metrics listed above can completely describe the ARA performance. Specifically, performance laxity cannot measure the transitory effects of reallocation, while reaction time and recovery time do not reflect steady state improvements. In addition, trade-offs exist between focusing on performance laxity vs. reaction time. Optimal performance laxity may result in reaction times that exceed acceptable delays due to high decision or enactment overheads.

When interested in characterizing the ARA performance over a long interval of time, reaction time and recovery time may be estimated by their maximum values and performance laxity by its minimum value over all instances of critical variations.

Another interesting issue about the ARA infrastructure performance is the *necessity* of automatic adjustments. The perturbation induced on the application by an unnecessary adjustment increases the risk of failing to meet its performance constraints. To assess the necessity of an adjustment often requires knowledge about the future evolution of the system, which is typically not available. For instance, a singular spike in CPU needs should not trigger an increase in CPU resource allocation for the corresponding component. Although we do not include in our set a metric capturing the necessity, we consider it when designing the ARA mechanisms, primarily those related to detection and state assessment.

Related Work. Previous studies considering automatic ARA adjustments for real-time applications [10, 11] typically compare the performance attained with ARA against optimal solutions. [10] proposes algorithms for optimal resource allocation decisions, but which can trade optimality for short decision times. [11] evaluates the ARA performance by the loss in application performance with respect to the performance enabled by an ideal ARA infrastructure with instantaneous detection, optimal decision, and no overheads. In contrast, we submit that satisfying the application's performance requirements is more important than achieving optimal application performance.

5. Factors for ARA Reaction Time

In this section, we consider the detection and the reallocation decision mechanisms, and show how their design can affect the reactivity of the ARA controller, and consequently, the satisfiability of an application's performance requirements. In addition, our experiments show that reaction time is an important performance metric: improved ARA reaction time implies improved application performance.

The experimental results reported in this study are obtained with a synthetic, distributed application designed by Honeywell in the context of high performance C3I applications [15]. The application performs on a cluster of eleven UltraSPARC-I Model 170 workstations with an MPI-1 interface over 100Mbit switched Ethernet links. The application consists of multiple communicating components connected by an acyclic graph of communication links. Each component can adapt its execution to span over any number of processors. Components do not share nodes, and each node is executing the same program. Each component module executes the following steps: (1) receive a message from each module of its component immediate predecessors, (2) execute according to the computation and intra-communication pattern specific to its component, (3) send a message to each module of its component immediate successors.

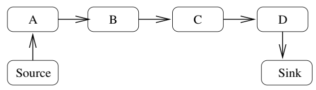

Figure 4. Synthetic Application Example

In the following experiments, the synthetic application has a pipeline configuration (see Figure 4). All events have the same type. They are produced periodically by the *Source*, consumed by the *Sink*, and processed by the intermediate components. For each component, the step (2) above consists of: (2.1) exchanging a message with all of the modules in the same component; (2.2) computing for an amount of time that depends on the component's parallelism level and speedup coefficient; and (2.3) exchanging messages as in (2.1). A stochastic model is used to emulate a step-like data-dependent variation of computation and communication needs.

The enactment overhead is minimal as no synchronization is required among resource donors, receivers, and the components with which they communicate. Adaptation overhead is small and identical for all components.

In the following, the "acceptable limit" for a particular performance metric is the upper bound derived from a corresponding performance constraint. In all of the experiments, the acceptable miss burst is set to one. Also, any detection signal triggers an automatic adjustment.

5.1. Detection

In this section, we address the effects of early detection. The performance of a detection method is evaluated by: *promptness* – how soon after its occurrence, a critical variation is signaled; *trustworthiness* – what ratio of signaled variations is critical. A prompt detector implies rapid detection, and thereby low ARA reaction time. Detector trustworthiness is related to the necessity of allocation adjustment: the trustworthier the detector, the lower the likelihood of making an unnecessary adjustment.

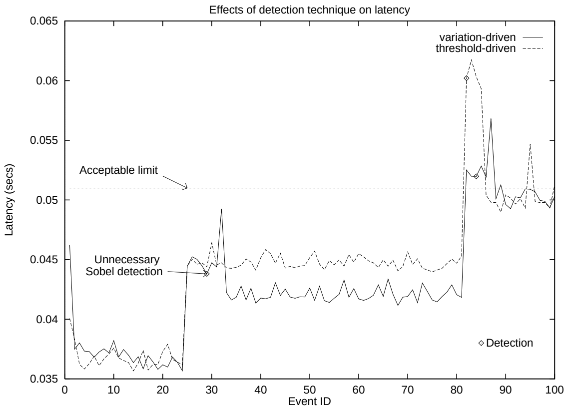

Figure 5. Promptness vs. trustworthiness

Promptness is more important than trustworthiness when performance constraints are being violated. Figure 5 presents the influence of detection promptness on end-to-end latency. We experiment with two detectors: a *threshold-driven* detector, which checks the current sample of the metric of interest against the acceptable limit, and a *variation-driven* detector, which checks whether a significant variation of the metric of interest has occured. Our variation-driven detector is similar to the Sobel detector used for edge-detection in computer vision [5]. This detector is trustworthier than the variation-based detector because it employs smoothing techniques to eliminate the effect of noise, and its decisions are based on a range of samples around the sample of interest. Unfortunately, these techniques result in poor promptness. The threshold-driven detector, on the other hand, is prompt, but is likely to be untrustworthy because it is sensitive to noise. The impact of prompt detection is demonstrated in Figure 5 which shows (e.g., see Event ID 80) that the number of events failing the end-to-end latency constraint can be larger with the variation-driven detector (with smoothing size 5, and sample-range size 11).

On the other hand, a trustworthy detector can be used to detect changes in the application behavior which do not immediately cause the performance constraints to be violated, but which increase the risk of such a situation. In our experiment, a change in execution time which causes the end-to-end latency to get within 10% of the acceptable limit (see Event ID 25), is signaled by the variation-driven detector and triggers a reallocation which reduces the latency to more than 15% below the acceptable threshold. A threshold-driven detector cannot be used for detecting similar changes because it can not distinguish between a spike and a steady variation.

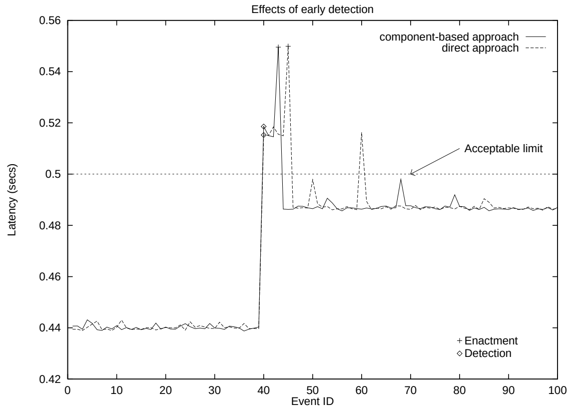

Figure 6. Effects of metric evaluation method

Promptness can be affected by the method used to evaluate the metric of interest. The performance metric of interest for a detector may be evaluated either directly – *direct* approach, or by composing several independent metrics – *component-based* approach. In the former case, the metric is tested against its acceptable limit any time a new sample becomes available. In the latter case, the metric is *evaluated* and tested any time a new sample of one of its components becomes available. For instance, in order to detect variations of the end-to-end latency, one can use its direct measurement, or the sum of the most recent measurements for the execution and communication overheads of each application component on the event's critical path.

The component-based approach is prompter than the direct approach and results in improved performance due to shorter reaction times (see Figure 6, where component A's execution time increases). This difference is particularly significant when the event path is long (in terms of latency) and the critical variation occurs early on the path.

5.2. Reallocation Decisions

In this section, we address the effects of considering the enactment overheads and of using application state-specific incremental heuristics in the ARA decision making.

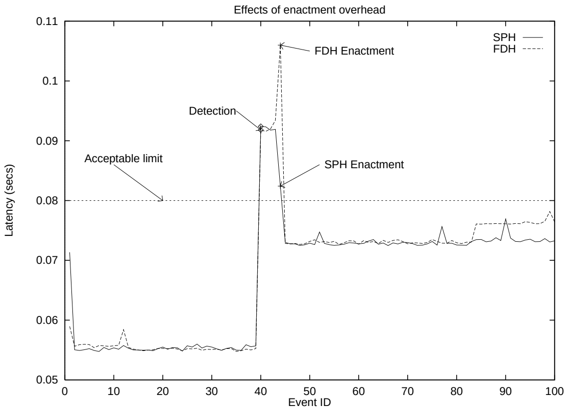

Figure 7. Influence of enactment overhead

Reallocation heuristics aware of enactment overheads result in improved performance. Figure 7 depicts the end-to-end latency variation with two decision heuristics distinct in their awareness of enactment overheads: First, the 'single-pair' heuristic (SPH) tries to accommodate a critical variation with node reallocations involving only two components, thereby likely to result in lower enactment overhead than reallocations involving more components. Second, the 'fair-decrease' heuristic (FDH), ignores enactment overhead, and uses reallocations involving multiple components, as it tries to be fair about reducing the number of nodes available to different application components.

To accommodate an increase of component A's computation needs (see Figure 4), the SPH decides a 2-node transfer from component C to A, while the FDH decides a 1-node transfer from each of the components C and D to B. Both heuristics lead to similar steady state performance. However, the enactment overhead with FDH (23 msecs) is larger than with SPH (18 msecs). Thus, the number of events failing their latency requirements is larger with FDH, the heuristic not accounting for enactment overhead.

Application state-driven incremental decisions can reduce reaction time. Incremental decisions can improve ARA performance because they can take advantage of current resource allocation. We show that incremental decision algorithms aware of the current state of the application permit even better performance.

We experiment with a simple incremental decision algorithm that repetitively selects pairs of potential receiver and donor components and tests whether the performance metric(s) under constraints can be improved by a 1-node transfer. The time taken to produce an acceptable reallocation depends on the order in which the components are selected, while the effectiveness of an ordering criterion varies with the state of the application.

We experiment with two ordering criteria: (1) by *current execution time, CE*, and (2) by *expected execution time variation upon reallocation, EV*. CE ranks by the component current execution time, donors in increasing order, and receivers in decreasing order. EV ranks donors in increasing order of the expected increase in their execution time that may occur when releasing a node, and receivers in decreasing order of the expected reduction in their execution time that may occur when being assigned a node. We use these criteria to decide reallocation decisions in two types of application states: a rate-critical and a latency-critical state.

In a rate-critical state, the primary goal of reallocation is to reduce the maximum execution time in the system. Thus, the receiver should be the bottleneck component (i.e., largest execution time) and the donor may be any other component, provided the resulting maximum execution time satisfies the acceptable event rate constraints. The CE ordering helps to focus immediately on the components with the highest and lowest execution times. In our experiment, an acceptable pair is found with the CE ordering after testing one (receiver, donor)- pair (1.34 msecs), and with the EV ordering after testing four pairs (1.58 msecs). Note that in this experiment, it takes approx. 0.080 msecs to test a pair, the rest of the reported time being spent with other ARA mechanisms or parts of the decision procedure.

In a latency-critical situation, the primary goal is to improve the sum of the execution times of all of the components on the event's critical path. The best solution attainable with a reallocation involving only two components (i.e, with low enactment) is to select as receiver the component expected to have the largest reduction in execution time, and as donor the component expected to have the lowest increase in execution time. This is the first pair selected by the EV ordering. In our experiment, an acceptable pair is found with the EV ordering after testing one pair (2.56 msecs), and with the CE ordering after testing four pairs (2.83 msecs). In conclusion, the two ordering criteria we experimented with enable minimal decision overhead, but each in a different system state.

To summarize, our experiments show that the ARA performance is improved by considering both the application characteristics and its current state when choosing the methods for detection and allocation decision.

6. Contributions and Future Work

This paper considers the problem of adaptive resource allocation (ARA) for high-performance real-time applications executing in dynamic environments. Applications consist of multiple parallel tasks with data-dependent resource needs. Our contributions are the following:

- An application resource usage model that captures those characteristics of parallel real-time tasks that are required for making good reallocation decisions.
- An adaptation model enabling automatic resource allocation adjustments and the ability to evaluate their enactment overheads.
- Experimental demonstration of the importance of focusing on the response time of resource allocation mechanisms rather than on the optimality of their decisions, when real-time constraints must be satisfied.
- A novel set of performance metrics for evaluating ARA performance that focuses on the satisfiability of the application's timing constraints. These metrics are reaction time, recovery time, and performance laxity.
- Identification of factors related to detection and decision mechanisms that influence the satisfiability of the application's timing constraints. These factors are early detection, enactment overhead, and application state-driven incremental decision heuristics.

The models and heuristics presented in this paper are proved useful in the context of processor reallocation for an adaptive, synthetic application designed to represent time-critical applications in C3I systems. In the future, we plan to apply them to other types of adaptive applications including a complex, distributed computer vision application. We also plan to integrate the insights and mechanisms presented here into a broader framework for resource management destined for systems where multiple real-time applications coexist, and where the ARA mechanisms described in this paper are used in conjunction with online negotiation mechanisms.

References

[1] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control. *Real-Time Technology and Applications Symposium*, 1997.

[2] A. Banerja and D. Ferrari. The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences. *IEEE/ACM Transactions on Networking vol.4, no.1*, Feb., 1996.

[3] T. Bihari and K. Schwan. Dynamic Adaptation of Real-Time Software. *ACM Transactions on Computer Systems*, May, 1991.

[4] S. H. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing. *IEEE Transactions on Computers*, Jan., 1988.

[5] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov., 1986.

[6] S. Chatterjee and J. Strosnider. Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems . *15th International Conference on Distributed Computing Systems*, 1995.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, May, 1986.

[8] G. Eisenhauer, B. Schroeder, K. Schwan, V. Martin, and J. Vetter. DataExchange: High Performance Communication in Distributed Laboratories. *9th International Conference on Parallel and Distributed Computing and Systems*, Oct., 1997.

[9] J. Huang and P.-J. Wan. On Supporting Mission-Critical Multimedia Applications. *3rd IEEE International Conference on Multimedia Computing and Systems*, 1996.

[10] J. Jehuda. Automated Meta-Control for Adaptable Real-Time Software. *Real-Time Systems Journal*, (to appear).

[11] R. Jha, M. Muhammad, S. Yalamanchili, K. Schwan, and D. I. Rosu. Adaptive Resource Allocation for Embedded Parallel Applications. *3rd Int. Conference on High Performance Computing*, 1996.

[12] M. B. Jones, D. Rosu, and M. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *16th ACM Symposium on Operating Systems Principles*, 1997.

[13] K. Marzullo and M. Wood. Making Real-Time Reactive Systems Reliable. *4th European SIGOPS Workshop*, 1990.

[14] C. McCann and J. Zahorjan. Processor Allocation Policies for Message-Passing Parallel Computers. *ACM Sigmetrics*, 1994.

[15] R. C. Metzger, B. VanVoorst, L. S. Pires, R. Jha, W. Au, M. Amin, D. A. Castanon, and V. Kumar. C3I Parallel Benchmark Suite - Introduction and Preliminary Results. *Supercomputing*, 1996.

[16] J. Molini, S. Maimon, and P. Watson. Real-Time System Scenarios. *Real-Time Systems Symposium*, 1990.

[17] D. M. Nicol and P. F. J. Reynolds. Optimal Dynamic Remapping of Data Parallel Computations. *IEEE Transactions on Computers*, Feb., 1990.

[18] K. Ramamritham and J. A. Stankovic. Dynamic Task Scheduling in Hard Real-Time Distributed Systems. *IEEE Software, Vol. 1, No. 3*, Jul., 1984.

[19] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *Georgia Inst. of Technology, GIT-CC-97-26*, Sept., 1997.

[20] D. I. Rosu and K. Schwan. Improving Protocol Performance by Dynamic Control of Communication Resources. *2nd IEEE International Conference on Engineering Complex Computer Systems*, 1996.

[21] K. Schwan, T. Bihari, B. W. Weide, and G. Taulbee. High-Performance Operation System Primitives for Robotics and Real-Time Control Systems. *6th Symposium on Reliability in Distributed Software*, 1987.

[22] H. Zhou, K. Schwan, and I. Akyildiz. Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler. *Real-Time Systems Symposium*, 1992.