An Adaptation System for High-Performance Parallel Applications

Prabha Gopinath
Philips Laboratories
A Division of North American Philips
Briarcliff Manor, NY 10510
psg@philabs.philips.com

Adaptations are changes that are made to programs to improve performance. We discuss the utility of programmed adaptations in high-performance, parallel, real-time applications. The adaptation components of an existing system are described. We present algorithms that enable an application to maintain specified behavior in spite of overload conditions.

1 Motivation

Both existing and proposed real-time systems have such stringent performance and functionality requirements that the associated hardware, system software, and control software have come to be exceedingly complex in nature. Examples of such systems are the Adaptive Suspension Vehicle (ASV) [2], automated factories, where robot manipulators transfer workpieces between Automated Guided Vehicles (AGVs) and machining cells [8], and the NASREM NASA space station [1], where teams of robot workers will cooperate on maintenance tasks.

Both the hardware and the application software have complex interaction patterns. The application software is often organized in a hierarchical manner. Additionally, the subsystems of the computer hardware are often selected so as to best suit a specific task and also display a hierarchical structuring. Hence, both the hardware and the software exhibit significant inhomogeneities. As an example, in the ASV and the Automatic Land Vehicle robots, Lisp machines perform the high-level planning tasks and interact with tightly-coupled multiprocessor systems that control the mechanical systems of the robots.

Karsten Schwan
Georgia Institute of Technology
schwan@gatech.edu

Requirements for embedded systems often change during the lifetime of the product. Such changes may result from changes in the environment of the product, changes in the performance desired of the product, or perhaps changes that are made to other components that constitute the product. In any case, both the system software and the application software must be easily adaptable to the new set of requirements. Such adaptations may be dynamic - during a mission, or static - prior to a mission or between missions.

2 Adaptations and Adaptability in Real-Time Systems

In general, adaptation and support for adaptation is dictated by requirements of predictable behavior in real-time applications. Thus, programmed adaptations in an application allow the system to demonstrate pre-defined behavior in response to varying operating or external conditions. Examples of adaptations are:

- Configuration adaptations that involve one of the following:
  - Substitution of a set (subset) of components for other components with different performance characteristics; such as the substitution of a component that performs extensive error checking at the expense of latency, for one that performs very little error checking but has a quick response.
interactions.

- Changes to the interaction structure of an application; such as changing the target of an invocation so as to use an object that is less loaded and so can offer better response times.

- Changes to the attributes of an interaction. These include modification of deadlines, deadline types, preemption switches, etc.

- Changes, at run-time, to the task-processor map, to avail of a lightly loaded processor (load balancing).

Adaptations may be performed *statically* - on an attached development system between execution cycles of the application, or *dynamically* - during the execution cycle of the application. Static adaptations, by their very nature, can be more extensive, since they are performed off-line, and since there is rarely any real-time constraint associated with performing them. In addition, it is possible to use software development tools, such as compilers, to perform them. Since dynamic adaptations are enacted during execution of the application, they have to be performed within the real-time constraints imposed on the application. Dynamic adaptations, therefore, are more restrictive and limited in scope. The changes occurring as a result of performing a dynamic adaptation are usually incremental. Again, in contrast to static adaptations, the actual cost of adaptation enactment is significant, since it is performed in conjunction with the executing application.

Adaptations consist of two distinct phases, the decision-making phase - where the actual decision to perform the adaptation is made, and the enactment phase - where the adaptation is actually performed. The decision-making process can either be automatic - where data generated by the execution of the application triggers appropriate algorithms, or by user-request - at the explicit request of application programmers or maintenance personnel.

The availability of algorithms and tools for expressing and performing adaptations is a desirable attribute of any application development effort. During the software development phase, adaptations make experimental programming [4] convenient. During the operational phase, adaptation mechanisms allow an application to be tailored, or to tailor itself, to suit prevalent operating conditions.

To summarize, an adaptable software system is essentially a feedback control system, as shown by Figure 1, driven by data generated by the execution of the application. The purpose of performing adaptations is to reconcile a mismatch between the desired performance of the application software, the actual observed performance of the application software, and the prevalent operating conditions. The adaptation software system monitors the behavior of the application software system (via the sensory devices embedded in the plant), compares it to the desired behavior (using external sensory devices), and in the event of a mismatch, attempts to modify (using the embedded actuators) the characteristics of the application software system so that the actual observed performance approaches the desired performance.

In embedded real-time systems, predictable behavior, in spite of changing operating conditions, is desirable, since unpredictable behavior may have severe consequences. An example of operating conditions that may result in unpredictable behavior are overloads. During an overload condition, the system is subjected to increased demands and more severe timing constraints for a short period of time. Adaptations, and dynamic adaptations in particular, are especially useful in dealing with such overload conditions. In general when an overload situation is detected, an adaptation is triggered which modifies the characteristics of the
executing software. In the remainder of this paper we describe an example of an adaptation control system designed specifically to deal with overloads.

Figure 1: An Adaptable Software System

2.1 Automatic Adaptations

Automatic adaptations are especially appropriate in the context of embedded real-time applications - where human intervention may not be possible owing to several reasons such as the following:

- The control path of the application may have stringent time constraints, and human intervention, in response to changing conditions, may not be fast enough to prevent catastrophe.
- The very remoteness of the application execution site may preclude human intervention, such as in unmanned space probes.¹
- The complexity of sophisticated target hardware, may prevent application programmers from understanding the target system enough to perform the required adaptations manually. In such cases it is desirable that the adaptation algorithms be built into the software system ab initio.

Of particular interest to this document is the fact that automatic adaptations allow a real-time application to behave in a specified manner in response to situations such as described above. Such adherence to a specified pattern of behavior is crucial to the success of any embedded real-time system.

2.2 Adaptation Mechanisms

To write adaptable application software the application programmer needs support from the environment that allows him to conveniently monitor the behavior of the target system, analyze the observed behavior, and issue adaptation commands that will modify the target software in a controlled manner. Figure 1 may be used to suggest a minimal set of mechanisms required to support adaptations in real-time applications.

Monitoring mechanisms are required to monitor both the executing application as well as the external environment. The external environment in some cases would correspond to operator input. A data representation, storage, and management system is needed to handle the monitored data. A collection of algorithms that analyze the monitored data, and select a set of suitable adaptations, is also required. Finally, we need mechanisms that accept commands from the analysis module and perform the adaptations.

Figure 2 shows the adaptation control system provided by an existing environment. This is a feedback control system along the lines of Figure 1 and consists of the following components:

1. The Monitoring System (MON) - This collects data from the executing target system and transfers it to

¹ "Your Control-C will reach the probe in six hours. Thank You."
2. **The Adaptation Controller (AC)** - The adaptation controller accesses the data management system and retrieves values that have been stored there by the monitoring system. It analyzes these values and decides when it is necessary to make an adaptation and what the nature of the adaptation should be. The adaptation controller is application-specific.

3. **The Adaptation Enactor (AE)** - The adaptation enactor's function is to accept the adaptation commands from the AC, make the necessary changes to the executing software (in the case of dynamic adaptations), and update the database to reflect the changed state of the system.

4. **The Data Management System (DMS)** acts as a common repository for information retrieved by MON, used by AC, and modified by AE. The DMS is implemented as an augmented Entity-Relationship database with action routines associated with the entities and relationships. In fact the DMS contains both the AC, called from within the DMS process, and the AE, coded as the action routines.

### 2.3 Static vs Dynamic Cycles

Static adaptations are performed between runs of the application software. In the static adaptation cycle, the AC collects information about the executing application by querying the DMS. At some point the AC makes a decision to adapt. The new software configuration is generated, the target system may be stopped or is allowed to complete, the new configuration is loaded and the target system is restarted. In a dynamic adaptation cycle, the AC collects information about the executing application by querying the DMS. As before, the AC makes a decision to adapt the target software. In this case however, it triggers the AE to enact the adaptation while the target system is still executing.

### 2.4 The CHAOS Adaptation System

In this section we briefly outline the CHAOS (Concurrent Hierarchical Adaptable Object System) system. CHAOS [6,7,10] is an object-based system which allows real-time applications to be programmed as abstract objects interacting using invocations. Adaptation and support for adaptation is a central aspect of the design of CHAOS. An object in CHAOS has a variety of attributes. These include its location, its degree of internal concurrency, and the performance attributes of its operations. Objects that have internal concurrency have a coordinator that controls the multiple threads of execution within the object. The behavior of the coordinator is specified by the application programmer.

Invocation primitives in CHAOS are tailored towards the real-time domain and support efficient control interactions, data streaming, and data and control interactions. CHAOS also provides a mechanism by which invocation primitives may be synthesized on an application-specific basis. Invocation primitives have attributes associated with them. These include a deadline, nature of the deadline (hard/soft), and
an indication of the criticality of the invocation.

The adaptation mechanisms identified in the previous section are incorporated in CHAOS and are used for both static and dynamic adaptations. Static adaptations are typically used for experimental programming, which in a CHAOS application consists of modifying the attributes of objects, changing the behavior of an object's coordinator, and experimenting with various invocation primitives. Dynamic adaptations in CHAOS are used either for anticipating and preventing deadline failures or for detecting and recovering from them. Dynamic adaptation mechanisms are discussed in a later section.

3 Suitability for Adaptations

Not all real-time applications are amenable to adaptations. In this section we identify some essential criteria for adaptability. Regardless of the efficiency of adaptation mechanisms, monitoring, analysis of monitored data, selection of adaptation(s), and enactment of selected adaptations requires a finite number of processing cycles.

The criteria for static adaptations are trivial. The application would typically be in a prototyping phase, where the effects of various system constructs on the performance of the application are being investigated. The development system, with attendant tools such as compilers, linkers, etc., is still coupled to the target machine.

We impose restrictions on the class of applications that are dynamically adaptable. Typically such applications are long-lived and repetitive in nature. There would be a single trigger event initiating an application cycle. Subsequent cycles in the application could be caused either by repetitions of the trigger, or by feedback from the end of the preceding cycle. The duration of a cycle is constrained either by the physical characteristics of mechanical devices controlled by the application, or by requirements imposed by the application programmer on the application. Spare processing time, that can be utilized by the adaptation mechanisms, should be available in each application cycle.

There are several real-world applications, such as parts tracking in robotics applications [5] and missile defense systems, that exhibit what is essentially an event-action behavior. For example in a robotics tracking application the initiator event is the detection of the part on the conveyor by the camera. The constraint on the application is the time required by the robot to achieve a track on the part. The application is cyclic and as a confidence measure the robot typically attempts to achieve a track twenty times before it attempts to pick the part up.

4 Performance Adaptations

4.1 Introduction

In this section we describe adaptation mechanisms that ensure that overall deadlines specified by an application programmer are satisfied both under normal operating conditions as well as under conditions of overload. Delivering specified performance under normal conditions of operation implies that when demands made on the system by external agencies stay within anticipated values, the system meets its specified overall deadline. Delivering specified performance under conditions of overload implies that when more stringent demands are made on the system by external agencies, the system is capable of adapting its behavior and thus continues to deliver specified performance. The two algorithms presented here should be considered as specific instances of the general notion that application software can be designed to respond to various classes of perturbations and still meet specified goals.

The first algorithm, which recomputes invocation dead-
lines so that they remain consistent with the overall specified deadline for the application, we call the Deadline Recomputation Algorithm or DRA. DRA uses knowledge of the invocation patterns of the application, execution time attributes of objects, and attributes of invocations to recompute invocation deadlines. DRA has a static component, the Static Deadline Recomputation Algorithm or SDRA, which is used during the initial configuration generation phase after the application software has been loaded onto the target hardware. The dynamic component, the Dynamic Deadline Recomputation Algorithm or DDRA, recomputes deadlines in the executing target system. DRA is described in detail in [6].

The second algorithm, which selects and performs version changes of object operations, we call the Version Control Algorithm or VCA. We do not distinguish between a Static VCA and a Dynamic VCA. VCA is described in detail in a later section.

Neither algorithm purports to address all possible conditions that could cause an executing application to experience overloads. Rather we restrict the scope of overload handling to detecting increases (or decreases) in the rate at which the application is triggered by an external initiator event. We feel that this constitutes adequate 'proof of principle', while at the same time allowing us to handle both deadline-failure prevention as well as deadline-failure avoidance.

Performance perturbations could also occur as a result of changes internal to the executing application such as increased execution time owing to data dependencies. Such a situation does not lend itself to deadline-failure prevention, but can be handled by deadline-failure avoidance. A trivial extension to our adaptation algorithms will handle this.

The E-R database, that comprises the DMS described earlier, contains a complete representation of all objects and their attributes, and also descriptions of all interactions among objects and their attributes. This information is stored in the database as an Application Graph which is defined as follows:

\[
\text{Application Graph } = \{V, E\}
\]

where:
- \(V:\{v \mid v = \langle \text{object-operation}\rangle\}\).
- \(E:\{e \mid e \text{ is an interaction}\}\).

DRA can only be used when there is sufficient spare processing time (or Slop) available on the nodes. Once the Slop falls below a threshold, DRA will not function. This threshold, which can be set arbitrarily high by the AC, has a lower bound which is the execution overhead of DRA itself. When the Slop falls below the threshold defined in the AC, two options are available to the AC. It can either abandon the attempt to keep deadlines from being missed or it can try and reduce the overall processing time requirements of the application by switching in low overhead, low functionality versions of object operations. Similarly when the Slop exceeds a threshold, the AC can elect to switch in high functionality, high overhead versions of object operations. The two thresholds can be set to be a time \(T\) ms apart, where \(T\) would determine a hysteresis required to prevent instability.

### 4.2 Version Control Algorithm - VCA

The key concept behind VCA is the notion of Parallel Interaction Graphs\(^2\) which are specified by the programmer. Each parallel interaction graph represents one version of the application program and possesses a unique performance and functionality characteristic. Thus, each parallel interaction graph represents a point on a continuum extending from an

---

\(^2\)We have resisted the urge to define an acronym for this phrase.\n
interaction graph with high functionality, and hence high overhead, to an interaction graph with low functionality, and correspondingly low overhead. In theory a programmer could specify an arbitrary number N of such interaction graphs. Since all graphs are loaded onto the target system, in practice N is limited by system memory. VCA adapts an executing real-time application by dynamically moving it along this continuum of interaction graphs. It is activated when timing constraints are so critical that there is no longer enough slack time in the system for DDRA to execute. After a new version has been selected by VCA, DDRA is again reactivated.

4.3 Discussion of VCA

The function of VCA is to move the application program along the continuum of interaction graphs, ensuring that at any particular instant, all active entities in the system belong to the same interaction graph. As an example of changing interaction graphs, consider the interaction graphs for the tracking application shown in Figures 3 and 4.

Figure 3: Interaction Graph 1 for Robot Tracking

The two graphs correspond to the same application, but exhibit different functionality and performance behaviors. The version of the AdjustKin object in the interaction graph in Figure 3 has high functionality and generates a robot position where the end-effector is properly oriented with respect to the part on the conveyor. The overhead of this object is high. In contrast, the interaction graph shown in Figure 4 implements a low functionality version of the InvKin object.

Figure 4: Interaction Graph 2 for Robot Tracking

While this object generates a valid track on the part, it does not attempt to correct the orientation of the end-effector for each new position of the robot arm. Suppose that the track application experiences an overload condition and DRA has terminated since the Slop on the node has fallen below the threshold. Now the AC can activate VCA which would switch in the interaction graph of Figure 4 instead of the interaction graph of Figure 3. Since a low overhead version of InvKin has been switched in, the Slop on the node where InvKin resides will increase, perhaps over the threshold specified for the application. If this occurs DRA can be restarted.

4.4 The Algorithm

Assumptions:

1. All operation execution times have been specified by the programmer.
2. All objects in the application belong to the same interaction graph.

3. A version change will affect all objects in the application. Thus VCA cannot be used in its present form to change versions of subgraphs.

Algorithm: VCA(Direction):
Let CurrentVersion be the current version number.
if Direction is Speedup then
  Increment CurrentVersion;
else
  Decrement CurrentVersion;
endif
If CurrentVersion is within bounds then
  Foreach node in the target system Do
    Set the attributes of the application graph to the attributes of the current version.
    Attributes include the slope, invocations per node, and object activations per node.
  End;
  Halt target system.
Else
  End;
End

VCA makes it possible to experiment with different versions of applications at run-time. By appropriately implementing the adaptation controller, an application programmer can experiment with different versions of the executing application code while observing its dynamic behavior.

4.5 The Adaptation Controller
The adaptation controller must do the following:

- Detect the event that initiates an application cycle.
- Detect the termination of an application cycle.
- Execute the adaptation control algorithm.

The function of the AC is to use the above information in conjunction with information maintained by the DMS to detect when the constraint on the application changes, and appropriately select either DRA or VCA to execute. The adaptation controller described below integrates DRA and VCA. In general, an adaptation controller could be written to implement any algorithm appropriate to the application being adapted. For example, an AC could perform adaptations on a CHAOS application which modifies target system behavior to meet reliability criteria [3]. The AC shown below uses DRA and VCA to meet performance criteria.

Algorithm: AC:
PreviousCycleLength = ApplicationGraphDeadline;
CurrentCycleStartTime = StartTime(StartUp);
PreviousCycleStartTime = 0;
Loop Forever Begin
  LatestCycleLength = CurrentCycleStartTime - PreviousCycleStartTime;
  If LatestCycleLength < PreviousCycleLength Then
    dt = PreviousCycleLength - LatestCycleLength;
    If dt GEq Min{Slop on processor nodes) Then
      Enable VCA(SpeedUp);
    Else
      Enable DRA;
    Endif;
  Else
    If LatestCycleLength > PreviousCycleLength Then
      dt = LatestCycleLength - PreviousCycleLength;
      If dt GEq Max(Slop on processor nodes) Then
        Enable VCA(SlowDown);
      Else
        Enable DRA;
      Endif;
    Endif
  Endif
  PreviousCycleLength = LatestCycleLength;
  PreviousCycleStartTime = CurrentCycleStartTime;
  Get CurrentCycleStartTime;
Endif

5 Evaluation of the Adaptation Algorithms
Both DRA and VCA share some common characteristics as regards the way in which they affect the application. Both
algorithms take an 'all or nothing' approach to the application, in that, with the information currently maintained by the DMS neither algorithm can selectively modify subgraphs in the application interaction graph.

5.1 Evaluation of DRA

In the case of DRA this implies that while in certain situations DRA could modify the Slop (and hence the deadlines) on just one node and so meet real-time constraints, it will still recompute Slop values on every node in the configuration. This in turn will change invocation deadlines on all nodes in the system. The problem with this approach is that it is a 'worst-case' approach and so DRA could terminate prematurely simply because it has run out of Slop on nodes that don't even need to be modified.

Another way to handle overloads is to recompute deadlines on a per-invocation call basis. This would result in far greater precision for the deadline values. However, such a method would require significantly more information to be provided by the application programmer. This would include branch probabilities for conditionals and upper bounds on iterations. In addition, all this information together with unique identifiers for each instance of an invocation primitive in an object would have to be maintained within an object control block. The gain in precision is obtained at the expense of increased overhead in recomputing deadlines and in updating invocation tables.

5.2 Evaluation of VCA

VCA, like DRA, operates on an 'all or nothing' basis. Changing a version of an operation involves changing all other operations in the application to the corresponding version number. We adopted this approach so that it would not be necessary for the application programmer to furnish any more complex information. Another reason is that changing one operation to a different version usually means that operations that are upstream from it and downstream to it need to be changed, with regard to their interfaces with the operation just modified. The most pessimistic approach (as adopted by us) assumes that all other operations in the application also need to be modified to remain consistent with the modified operation. A better approach would store a table of version numbers and corresponding dependent objects in the object control block. Now, changing the version of one operation would require that the table in the object's control block be examined and all dependent objects also be changed to the corresponding version number. This approach offers better control over the version change process, but at the expense of the increased overhead in performing VCA. Such an approach would also require the version change process to be performed as a transaction so that all affected objects have a consistent view of version numbers.

The main advantage to the method we use is that it is extremely simple and has very low overhead. All that is involved in enacting a version change is changing one integer on each node in the configuration. The programmer need not specify any additional information for the algorithm to function.

5.3 Consistency of Enactments

One of the main problems associated with performing dynamic adaptations is ensuring that adaptation enactments are atomic. An enactment should affect all components of the target software at the same time. The basis for our arguments on the atomicity of DRA and VCA lies in the fact that adaptations will be enacted at most once during an application cycle and such enactments will occur during the Slop.
time on the nodes. During the Slope time of a node all application objects on the node are quiescent and so enactment can proceed. At the beginning of the next cycle the adaptations have been enacted and all objects have a consistent view of the system.

6 Final Comments

In this paper we outlined a system that enables the programming of embedded real-time applications that are statically and dynamically adaptable. We described the need for adaptation support and presented the mechanisms that facilitate the programming of adaptations. Finally, we presented some algorithms that have been used in robotics applications to ensure that deadlines continue to be met in spite of changes in the execution environment.

References


