Abstract

In this work we approach the problem of monitoring and debugging real-time distributed systems by performing static analysis and transformations to eliminate obtrusion to the monitored system. Our work extends the CRL testbed compiler and run-time environment to support monitoring and logging for the purpose of post-mortem debugging. The main contribution of this work is the innovative use of compiler transformations and idle slots for monitoring and logging.

1 Introduction

Monitoring and debugging in a real-time distributed system is a non-trivial problem, which, despite the significant research in the area, still lacks a satisfactory solution. The difficulties arise from the fact that observing the behavior of a real-time process may alter it (the probe effect). Furthermore, a monitoring intrusion in one process may affect the entire system by altering the normal resource access pattern. Accounting for intrusion on the communication medium is especially challenging since it requires a global analysis of the distributed system.

2 Motivation

The motivation for this work is two-fold. First, we are inspired by recent research on real-time languages and environments ([1], [2], [3], [8], [10], [14]). There has been a lot of effort both to provide language support for real-time computing and to design methods for monitoring of distributed real-time systems, resulting in numerous proposed monitoring schemes ([28]).

Secondly, we are motivated by our view of the development process of real-time systems. For complex distributed real-time systems the developer needs sophisticated tools which provide for automated analysis of the system, both during design and development as well as during performance evaluation and testing ([6], [27]). Without such tools it is nearly impossible to develop highly dependable systems, not even considering cost and efficiency of the development process ([9]). Our practical approach to non-intrusive monitoring provides a simple and viable solution to the difficulties inherent to this problem.

3 Compiler and transformation engine

In this section we discuss two components of a development environment and a real-time language under construction in the Dependable Real-time Systems Laboratory (formerly the Real-Time Computing Laboratory) at New Jersey Institute of Technology [27]. The compiler and transformation engine form the heart of this development environment, which is based on a new high level real-time language, CRL [26], designed for building complex applications with a variety of (possibly conflicting) requirements. CRL is an object-oriented language which provides a rich set of timing constraints low or high granularity (from statement level to system level). CRL incorporates natural parallelism via threads and inter-process communication via messages and remote procedure (method) calls.

The CRL compiler is a front end for our environment. During compilation, standard static analysis is performed: control flow and call graphs are extracted and exported for further use along with the symbol table and other information. The timing properties of basic blocks are determined (worst case execution times) and combined according to the control flow and call graphs to determine the WCET of methods ([15], [16], [17], [1], [7], [12]). This is possible since CRL permits only bounded control structures. The compiler translates the CRL code into an intermediate representation, currently a restricted and structured subset of C++, which is used for further analysis and transformations. The intermediate code consists of almost-instruction-level statements, which are trivial to analyze, assuming there exist an instruction time-map for the target architecture. This assumption is not excessively restrictive, since even with
small variations, experiments have shown that within a basic block, instruction execution times tend to average out.

The analysis and transformation engine uses the intermediate code and control structures extracted during compilation to perform schedulability analysis ([19], [20]) and code transformations. We are interested in two types of transformations. First, we consider optimizing transformations, for example removing invariant form a loop. Such transformations always improve the average execution time and do not extend the WCET, if we consider the process in isolation. In a system with multiple communicating processes, however, this kind of optimizations may perturb resource access patterns which may lead to unpredictable delays and consequently to extending the WCET. To avoid this effect, we replace "gained" time by idle instructions (no-ops), which guarantees that optimizing transformations will be time-wise safe in a multiprocessor environment.

A different kind of transformation may also lead to the insertions of such idle slots in the program code. Consider conditional balancing and conditional linking ([24], [25]) - two ways to improve the schedulability analysis of the code. In conditional balancing we work with the time-wise "shorter" branch of a conditional and attempt to transform it into a timing replica of the "longer" branch. We insert idle instructions before any resource access to match a longer computation, or after a resource access to match a longer computation at the resource’s (remote object) site. If a resource access (via RPC) is unmatched, we insert a call the to same resource, under a different version (method) to essentially make the resource access profile of both branches identical. The result of these transformations is that the two branches of the modified conditional look identical to the schedulability analyzer and only one of the paths will contribute to the number of possible executions being considered.

Another source of idle spots could trivially be the programmer, who may explicitly specify idle time slots for synchronization or other purposes. The collection of various idle slots could be beneficial to the overall performance of the system in a number of ways. In [29] and [30] the authors present a method for safely removing idle slots by performing multi-process analysis. In this work we are interested in using the idle slots for monitoring functions.

4 Automated instrumentation

Recognizing the need for compiler/tool support for development, streamlining and testing, we propose an automated and interactive approach to instrumentation and monitoring. The general problem in most cases is what to monitor. Both extremes – monitoring too little or too much – are undesirable and thus, the environment must provide a compromising solution. In our approach, the user is allowed to state explicitly what structures require monitoring via pragmas and annotations. These requirements may be approved, (partially) rejected or enhanced by the static analysis. The user must be kept aware of any changes of the monitoring requirements he or she stated in the code and may interact, providing additional insight for optimal selection of monitored variables.

To facilitate the validation and automation of monitoring instrumentation we rely on well-known analysis techniques. We select a set of variables to be monitored by performing standard data-flow analysis. Starting with the set of live variables, we prioritize, using use-def sets, by the occurrence of variable’s last change, the most recently changed having the highest priority. With a set of prioritized variables for each idle slot, we must perform monitoring timing analysis to verify if we can handle all selected for monitoring variables and, if not, to determine exactly how much we can monitor. After a safe set of variables is established we insert an appropriate monitoring stub for the current idle slot. The stub is handled by the kernel during run-time, but it has a statically known execution time, which depends only on the amount of information to be monitored (for a particular target platform), i.e. on the selected set of variables.

5 Non-intrusive Monitoring

Once we have placed the monitoring stubs in the transformed program, as result of the static analysis, we must provide run-time support for the monitoring activities. We do this in the execution environment for the CRL language, which is part of the development suite of tools. The run time environment consists of a kernel and a network simulator which interact with each other and also with information extracted at earlier stages (process-to-processor assignment, control flow, symbol table, etc.).

While the kernel is currently physically implemented as a single process, it maintains the abstraction of distributed operation and can be easily split up (by replicating it on each processor), should our platform become physically distributed. The kernel is executing a continuous loop; every iteration, it checks an event list, selects the next event, and performs the appropriate action. Events include: scheduling a thread, executing a call to a method, sending a message to a remote object (making a call to a method of an object currently residing on a different processor), and updating object queues. Every entry in the event table has a time-stamp to determine when the kernel should react to that event, and every object has a queue to serialize access to all methods exported by that object. The order in that
queue depends on the scheduling criteria used and the arrival order of messages.

Synchronized clocks are emulated (maintained by the kernel) for the entire network. Should the implementation migrate to a physically-distributed system, we would recommend use of GPS or standard time sources for synchronization, as advocated in [6]. The time is measured in abstract real-time units. All events are stamped with time of occurrence.

The kernel responds to an event by initiating the required activity; for example, by activating thread execution or initiating the execution of methods. Thus, calls (except some calls to local methods or system libraries) are directed as requests or as events to the kernel. The kernel actually makes the call by executing the callee method. While there are some implementation challenges ([27]) with this approach, we choose it because it facilitates preemption at natural points in the code (call sites and method boundaries), which enables the kernel to take control and potentially schedule another method for execution.

Monitoring stubs on the other hand are not treated as preemption points. The kernel gets control briefly, records the monitoring information with a time-stamp to a monitoring buffer and returns to executing the method at the next instruction. To avoid intrusion on network connections between nodes we are currently considering two models for transferring the monitoring buffer to the monitoring node (the node where the monitoring process resides). In the first, the buffer is gradually sent to the monitoring node during normal communications, as monitoring information piggy-backs on regular messages to fill up the fixed-size packages. In the second approach, we reserve bandwidth exclusively for monitoring communications. This is similar to a hardware approach in which the monitoring node is connected to all remaining nodes via a secondary monitoring network to avoid intrusion on the interconnection network.

6 Current status and future work

The CRL support environment is currently under development in the Dependable Real-Time Systems Lab at NJIT. The compiler, linker and the runtime are largely operational, though we are in the process of providing a generalized symbol table and general timing constraint support. A number of transformations, such as speculative execution, have been supported and work is on the way to support more. Basic timing and schedulability analysis tools are in place. The work on the assignment tool for CRL is in its early stages, though there are other assignment and allocation tools in operation (developed for other Lab projects). The tools can be demonstrated, with care, to interested parties.

We are currently implementing run-time support for monitoring, hoping to be able to experiment with different models of sending monitoring information over a standard communication network while eliminating or minimizing intrusion. We also plan to experimentally determine the efficiency of monitoring using idle slots. As we expand our platform, we plan to research the tradeoffs of compiler support versus run-time support for monitoring.

7 Acknowledgements

We would like to thank Ananth Ganesh, Robert Kates, Jeff Venetta and many cis611 (Real-Time Systems) students for their contributions to the CRL environment and tools. We are indebted to the Office of Naval Research (grant N00014-92-J-1367), the National Science Foundation (grant CCR9402827), and AT&T for providing generous support for this project. Last but not least, we would like to express our sincere thanks to Prof. Thomas Marlowe for his vision and dedication to the CRL project.

References


