Reconfigurable Software Development

Bradford K. Clark
Missile Software Branch, Naval Weapons Center
China Lake, Ca. 93555

Abstract

Reconfigurable Software is software for Real-Time embedded systems that is characterized as portable, modular and schedulable. Portability is achieved by use of a High Order Language and machine independent time management. Modularity is achieved using Abstract Data Types. The Rate Monotonic Scheduling theory is used for analysis of schedulability. Ada is the most suitable High Order Language to meet the characteristics of Reconfigurable Software.

Introduction

A major problem associated with virtually all embedded systems is achieving acceptable performance at affordable prices. Computer hardware and software is a major contributor to the price associated with many embedded systems. The key element in achieving affordable systems is to exploit commonality in hardware and software during acquisition and throughout the life cycle of the embedded system.

The cost of computer hardware is reduced by developing an architectural design that supports reconfigurability. By using a reconfigurable architecture, common hardware is used on a number of different embedded systems. Similarly, a suitable reconfigurable architecture supports and helps the introduction of new technology. This results in cheaper product improvements throughout the life cycle of the embedded system. The software costs are similarly reduced by transporting the software between multiple platforms, developing higher quality software that is easier to maintain, and producing software that will meet its real-time requirements.

Real-time software is software that is not only logically correct but must complete computation by a specified time[1]. The software consists of units of computation called tasks. Tasks are schedulable if all their deadlines are met—that is, if every periodic task finishes its execution before the end of its period[2]. The time requirements for software are expressed as a frequency that translates into a quantum of time called a frame.

While reducing or maintaining software cost is important, at issue is the risk involved in making such a commitment. The imposition of a time correctness increases the complexity of the software and for performance reasons makes the software specific to the platform. Faster processors do not always solve software performance problems because of the blocking that occurs when waiting for data or at a module-to-module synchronization point. Finally, the platform on which the application executes has size and weight constraints that impose restrictions on memory or processor selection.

This paper offers a concept called Reconfigurable Software to save on software costs. Reconfigurable Software is software that is portable, modular, and analyzable for schedulability (Figure 1). Portability is accomplished by developing software with a machine-independent language and by expressing the real-time schedulability requirements in a machine-independent form. Modularity is achieved by implementing the software from a design paradigm using abstract data types. Schedulability is determined by using Rate Monotonic Analysis. The

U.S. Government work not protected by U.S. copyright.
conclusion of this paper addresses the disadvantages of using this software.

Figure 1. Reconfigurable Software

Gaining Portability

Three areas contribute to the portability of software. The first is using a machine-independent language called a High Order Language (HOL). An HOL is portable because its programming constructs make no assumptions about the hardware upon which it will execute. Control loops, branching decisions, data representation, data storage, and error handling are implemented with English-like commands. Subprograms and variables are expressed with English terms. The portability of an HOL is achieved by using a compiler to translate a source code program into a target's microprocessor machine code. A possible limiting factor in this compiler translation for different machines is the proliferation of different language extensions offered by compiler vendors with the intent of making their implementation of the language for a target processor very fast. Another hindrance to portability is compiler optimizations for specific coding styles. These extensions and special optimizations result in a non-portable HOL and should be hidden (discussed later).

A second practice that contributes to portability is the isolation of machine interfaces from the rest of the software system. Examples of these interfaces are memory addresses, I/O ports, and bus interface units. When directly interfacing to the hardware, the interface must be encapsulated and hidden from the rest of the software. Just as the HOL hides the machine so must the programmer hide the machine from the application. The encapsulating module exports an interface for use by other software modules in communicating with the hardware. This encapsulation mechanism will make it possible to change the actual hardware without changing all of the software modules that use the interface.

The third consideration in making software portable is in the real-time requirements of the application. The current practice is to use count-down timers to signal via an interrupt the start of a new scheduling frame. While count-down timers are good as watchdog timers, timers should not be explicitly relied upon by the application as a source of time. If the count-down timer is varied to change the periodicity of one software module, all of the software modules will have their periodicity changed. The real-time requirements of each module should be encapsulated within that module. This requires an machine independent form of expressing time. Epoch time and elapsed time are sufficient abstractions of time to provide periodic execution but without reference to a machine-specific timekeeping mechanism. Epoch time is time of day and elapsed time is time since a starting point. Periodicity using epoch time is shown in Figure 2.

```plaintext
loop
    next frame ← time now + period
    ...
    suspend until next frame
end loop
```

Figure 2. Encapsulated Timekeeping

The software module in Figure 2 is in an infinite loop but does not run continuously. A determination of the starting time of the next frame is done before any work. During the work, the process may be interrupted any number of times, but it eventually arrives at the suspension point. It then suspends for any time remaining before the beginning of its next frame.
Modular Design Approach

The modular design paradigm for Reconfigurable Software is based on abstract data types (ADT) and encapsulation. An ADT is an abstraction of a significant detail in the solution space implemented as a software module. The abstraction focuses on the outside view of a module separating its behavior from its implementation. This consists of grouping data and the operations that manipulate that data into a single software module[3]. The outside view, or interface, is constructed with programmer-defined data types that represent the data abstraction and with subprograms that provide the abstraction of operations on the data. The programmer-defined data types restrict the range of values and convey the meaning of the data to be manipulated (Figure 3). The interface exports operations and data for use by other ADTs in the software system.

A software system built with modules can be composed of a combination of dynamic and static ADTs. Dynamic ADTs are units of logical or physical concurrency and execute at a periodic rate or by stimulation from an external event. Static ADTs execute only when invoked by dynamic ADTs. Dynamic ADTs are the modules that are examined for schedulability.

While ADTs export an interface for other software modules to use, they encapsulate the algorithms manipulating the data. This permits changing the underlying implementation mechanism while minimizing the impact on the rest of the software system. For instance, if the ADT manages a hardware device and that device with its controlling algorithms change, software modules that use the interface will not have to change their method of calling the device handler. Figure 3 is a simple interface to a serial port driver.

A goal of modular design methods is high module cohesion and loose inter-module coupling. ADTs contribute to high cohesion by localizing data and operations. Inter-module data sharing determines the degree of inter-module coupling. Asynchronous data transfer can be used to reduce coupling. This asynchronism is realized by the implementation of buffer ADTs placed between data consuming ADTs.

The use of a modular design approach has many benefits. The use of abstract data types eliminates global data. The data and its associated operations are encapsulated within the ADT. High module cohesion simplifies error correction. Errors are localized to the ADT and their correction has a minimum impact on the software system. High module cohesion makes the system tolerant of change. The underlying implementation of an algorithm or device driver is hidden and can be changed providing the ADT interface does not change. A modular software design based on ADTs can assist with portability and analysis of schedulability (Figure 1).

Real-Time Scheduling

The operations of an ADT represent units of execution. To guarantee schedulability of the software system, these units of execution must be scheduled to run on the processor in a specified order. Traditionally, the order of execution has been programmed into the software. Another approach is to formulate rules to be followed in deciding the order of execution and then provide a software module in the runtime, called a dispatcher, that makes scheduling decisions during the execution of the software system. This latter approach is more flexible for systems that undergo modifications.

An optimal software scheduling algorithm for single processor systems based on the rate monotonic assignment of priorities was ad-
This algorithm fixes priorities for a known set of tasks (units of concurrent execution) and requires that lower priority tasks be preempted from the processor if a higher priority task is ready to run. This scheduling algorithm also predicts the least upper bound of processor utilization. This prediction permits analysis of timing correctness and processor utilization for real-time applications.

The algorithm put forth by [4] assumes a set of independently executing tasks. Most software systems have some degree of task interaction caused by the exchange of data. This interaction causes problems stemming from the preservation of data. The traditional solution of mutual exclusion causes problems in the form of priority inversion and deadlock. These problems were addressed by [2]. A set of protocols were defined that addressed mutual exclusion, bounded priority inversion, and guaranteed freedom from system deadlock even with chained resource requests. With these protocols, an analysis of the interacting task set can be made (Figure 4). A blocking factor, $B_i / T_i$, identifies the longest duration of blocking that can be experienced by task $i$.

$$\sum_{i=1}^{\infty} \frac{C_i}{T_i} + \max \left( \frac{B_i}{T_i} \right) \leq n \left( 2^n - 1 \right) = U$$

Figure 4. Analysis with blocking

The operations of an Abstract Data Type represent units of execution. These units have a period of time in which they start and must complete their execution. These units can be measured for computation times that can be used in scheduling analysis. The operations of the ADTs can each be characterized as $C_i / T_i$. The blocking time, $B_i$, is measured by inspection. The sources of the blocking time can be hardware interrupts, device response time, or operations exported by other ADTs. Loose coupling in the modular design will reduce this term.

Using Ada

The Ada programming language provides the best opportunity for implementing Reconfigurable Software. Because it is a standard, the risk of creating portable programs that can be moved between different compiler systems is reduced. Many Ada compiler vendors offer hooks into their runtime, which is outside the definition of the language. With careful application of ADTs in software development, these special features can be isolated, which will localize changes to the software when moving to a different system. Ada supports internal timekeeping with its package Calendar. The programmer need not care how the timekeeping mechanism is implemented as long as the required resolution of time can be obtained. If not, an ADT that handles time can be constructed. Ada offers a virtual memory view of the computing system. Specific memory or port references are buried in packages.

Dynamic ADTs are implemented in Ada using tasks encapsulated in package bodies. Tasks can make software modules dynamic with infinite loops and a suspension mechanism called a delay statement. The delay statement uses Ada's self-contained time facilities. Another method that makes tasks dynamic are by accepting interrupt requests from the hardware. Tasks can also be used in static ADTs to provide mutual exclusion properties. These tasks are implemented with infinite ADTs and only accept statements. Some compiler vendors have a rate-monotonic-based scheduler in their runtimes. This is a desirable feature that should not be hidden and that will be incorporated into the upcoming revision of Ada.

The package construct in Ada supports the construction of Abstract Data Types. Packages can export data types, error conditions, subprograms, and tasks. The package can be dynamic if it uses a dynamic task implementation in its body. A package can enforce mutual exclusion by using a static task. The data types can be controlled by making them private. The Ada compilation system relentlessly checks data type coherence between the package interface and the using software module.
Conclusion

The ability to predict real-time performance will help with developing software that can meet its real-time requirements. The modular design approach of Abstract Data Types will make the software easier to maintain. The portability of the software will encourage use on multiple platforms. All of these factors will contribute to lowering the cost of software development for embedded real-time systems.

The strength in making software reconfigurable is in the definition of standard interfaces among software system modules or ADTs. Changing the interface impacts every reference to that interface in the software system. Evolving work in system domain analysis will provide the knowledge to correctly define interfaces that are stable.

Asynchronous data transfer helps decouple software system modules but at the expense of data aging. For instance, the algorithms in an autopilot may not be able to tolerate old data that is not reflective of the present state of the moving body. This requirement dictates that synchronous data transfer be used and that the resulting blocking time be accounted for in the scheduling budget.

Reconfigurable software is inefficient because of the path that must be followed when moving data about the system. Uncontrolled sharing of data would solve this problem but could render the software unmaintainable. Decisions on performance improvements should examine the hardware as well as the software. The portable, modular features of Reconfigurable Software make this a realistic alternative to slicing up the software system because it cannot accommodate a change in the hardware.

The physical size and weight constraints of some platforms reduce the selection of hardware choices for constructing the embedded system. Libraries of Reconfigurable Software would provide the resources for hardware-software trade-off studies. Such a capability will give the system engineer the most realistic picture of the total computing system.

Reconfigurable Software is significant in that it combines the ideas of portability and modular design with a proven real-time scheduling algorithm. The application of a scheduling theory to a modular design approach is new. Performance prediction is an important factor in real-time development. The ADT design approach fits the Rate Monotonic Scheduling theory, which provides predictability and reduces complexity.

References


