Achieving Predictable Execution in COTS-based Embedded Systems

Stanley Bak†, Rodolfo Pellizzoni‡, Emiliano Betti†, Gang Yao†, John Criswell†, Marco Caccamo†, Russel Kegley♯

University of Illinois at Urbana-Champaign, USA†, University of Waterloo, Canada‡, Lockheed Martin Corp., USA♯

Abstract—Building safety-critical real-time systems out of inexpensive, non-real-time, Commercial Off-the-Shelf (COTS) components is challenging. Although COTS components generally offer high performance, they can occasionally incur significant timing spikes. To prevent this, we propose controlling the operating point of shared resources, for example main memory, to maintain it below its saturation limit. This is necessary because the low-level arbiters of these shared resources are not typically designed to provide real-time guarantees. Here, we discuss a novel system execution model, the PRedictable Execution Model (PREM), which, in contrast to the standard COTS execution model, coschedules, at a high level, components in the system which may access main memory, such as CPUs and I/O peripherals. To enforce predictable, system-wide execution, we argue that real-time embedded applications should be compiled according to a new set of rules dictated by PREM. To experimentally validate the proposed theory, we developed a COTS-based PREM testbed and modified the LLVM Compiler Infrastructure to produce PREM-compatible executables.

I. PRedictable Execution Model (PREM)

Building computer systems out of commercial off-the-shelf (COTS) components, as opposed to custom-designed parts, typically improves time-to-market, reduces system cost, while providing generally better performance. For real-time systems, however, one hurdle in the way of using COTS is transient timing spikes which may occur when there is contention for shared resources. The low-level arbiter of shared resources in a COTS system typically does not have a mechanism to deal with the timeliness aspects of incoming requests, which may end up delaying more critical tasks, causing an unintended and undesirable priority inversion.

The PRedictable Execution Model (PREM) [1], in contrast to the standard COTS execution model, coschedules at a high level all active components in the system, such as CPU cores and I/O peripherals. Briefly, the key idea is to control when active components access shared resources so that contention for accessing shared resources is implicitly resolved by the high-level coscheduler without relying on low-level, non-real-time arbiters. Here, we specifically focus our attention on contention at the level of the interconnect and main memory.

A. Scheduling Memory Access

In order to schedule access to the main memory at a high level, we propose the PREM execution model, where tasks running on the CPU are logically divided into two types of intervals: compatible intervals and predictable intervals. The compatible intervals are compiled and executed without further modifications; they are backwards compatible, but they should nonetheless be minimized in order to provide a good level of resource (main memory) utilization. Predictable intervals, on the other hand, are split into two phases: a memory phase and an execution phase. During the memory phase, the task can access main memory, typically loading the cache with memory which will be accessed during the rest of the interval. During an execution phase, no further main memory access is performed. Each interval executes for a fixed amount of time equal to its worst-case execution time, which simplifies system-wide scheduling.

With PREM, peripheral access to main memory is also controlled. Peripherals will only access main memory when a task is in its execution phase. In this way, only one active component at a time accesses main memory, avoiding the effects of the non-real-time interconnect and memory arbiters. Figure 1 shows an example of a single predictable interval.

A more complex scenario is shown in Figure 2, where two tasks (τ1 and τ2) run together with two related peripheral I/O flows (τI/O 1 and τI/O 2). In this case, the input and output for each task is done in the adjacent I/O periods (double
buffering), and I/O driver execution at the start and end of each real-time job is done using compatible intervals. Again, only one component accesses memory at any time.

B. Practical Aspects and Implementation

In order to perform PREM scheduling, practical solutions are necessary to do both the division of intervals of execution into memory and execution phases, as well as ways to control when peripherals access main memory. Furthermore, a scheduling coordinator is necessary to ensure the various components follow the system-wide schedule. We developed solutions for these concerns, and evaluated our prototype system on several benchmarks.

From the task division aspect, we defined preprocessor macros which are used by a programmer to prefetch, into cache, data structures used during a predictable interval. These macros would also send messages to the global scheduler component, and included a loop to enforce a constant execution time for predictable intervals (equal to the worst-case execution time). This simplified the process of having a global schedule without affecting hard real-time guarantees (since the system is always provisioned for the worst-case execution time). We also created an LLVM [3] compiler pass needed to prefetch the code and stack used during a predictable interval. As part of future work, we are planning to further extend our compiler modifications in order to locate the accessed components follow the system-wide schedule. We developed solutions for these concerns, and evaluated our prototype system on several benchmarks.

From the task division aspect, we defined preprocessor macros which are used by a programmer to prefetch, into cache, data structures used during a predictable interval. These macros would also send messages to the global scheduler component, and included a loop to enforce a constant execution time for predictable intervals (equal to the worst-case execution time). This simplified the process of having a global schedule without affecting hard real-time guarantees (since the system is always provisioned for the worst-case execution time). We also created an LLVM [3] compiler pass needed to prefetch the code and stack used during a predictable interval. As part of future work, we are planning to further extend our compiler modifications in order to locate the accessed data structures in a predictable interval automatically, instead of relying on the programmer.

For the peripherals, we created a real-time bridge which, transparent to user-level applications, buffered a peripheral’s input and output until the global scheduler permitted access to main memory. This is necessary since, if we stopped the communication on the bus without hardware modification, the internal buffers of the peripheral could fill up and lead to data loss. Our real-time bridge included a large external memory which could store the data for longer periods of time while main memory access was restricted. Although this is custom hardware added into the system, off-the-shelf peripherals do not need to be modified to use the real-time bridge. This was demonstrated using an unmodified TEMAC Ethernet component on a Xilinx ML505 FPGA. Software running on the main system saw a standard network interface card.

Finally, to coordinate the system we created a new a system-aware hardware component. This peripheral scheduler, receives short messages from the CPU over the PCIe bus, and controls when peripherals can access the bus. In our implementation this was also done using a Xilinx ML505 FPGA. The preprocessor macros would send start and end interval messages to the peripheral scheduler. The peripheral scheduler would, at the appropriate times, allow the real-time bridges to transfer data to or from main memory using a dedicated wire.

A system-level view of the architecture is shown in Figure 3. Here, the peripheral scheduler is directly connected through external wires to each of the real-time bridges. It is also on the PCIe bus, which allows it to send and receive messages from the task running on the CPU. These messages are sent when the task executes the PREM preprocessor macros.

To test our prototype, several benchmarks have been compiled and executed according to PREM. These benchmarks demonstrated both that PREM could practically be used to increase system predictability, and that PREM did not usually introduce significant overhead. Tasks where PREM did not perform well were ones where the exact amount of data which needed to be prefetched during the memory interval could not be determined at runtime (for example, programs which had a largely-varying output size). In real-time systems, however, we believe these programs are rare since their execution times will also likely have a corresponding largely-varying range. Additional details about our experimental setup and the benchmarks are available in the PREM conference paper [1].

REFERENCES