

DARK: Designing A High Performance Micro-kernel for Power Electronics Controllers

Kuljeet Singh and Stephen H. Edwards

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061 USA

Abstract- Dataflow is a style of software architecture ideally suited for embedded control. The Dataflow Architecture Real-time Kernel (DARK) is a lightweight RTOS specifically designed to support such designs in power electronics control. DARK's high performance design is presented along with performance comparisons against a commercial real-time operating system.

I. INTRODUCTION

Choosing appropriate software architecture for embedded control applications can improve quality while reducing cost. Dataflow is an architectural style that naturally supports component oriented reusability in embedded software designs. Dataflow applications consist of concurrently executing processes that only interact with each other by asynchronous messages sent through data channels.

In this paper, we present an overview of an operating system kernel called DARK (Dataflow Architecture Real-time Kernel) that supports the construction of dataflow software.

The following section gives a brief introduction to dataflow architecture. Section III describes the design and implementation of DARK. Section IV contains performance comparison of DARK against MicroC/OS-II, a commercial RTOS [1]. We conclude with a short note on our experiences and future work to be done on DARK.

II. DATAFLOW ARCHITECTURE

Dataflow [2] is software architecture, in which a system is composed of nodes and arcs, as shown in Fig. 1. Nodes represent processes, whereas arcs represent data connections between them.

This architecture is being used to design software for plug and play power electronics building blocks (PEBBs) [3]. The dataflow processes map to Elementary Control Objects (ECOs), which are functionally self-contained and concurrently executing entities. A firing rule specifies the input channels on which the ECO should wait for inputs, before starting execution. The ECOs are connected to each other through data channels.

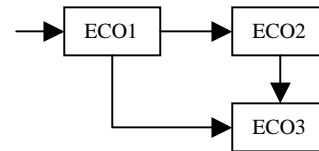


Fig. 1. A dataflow graph

Dataflow applications require a runtime supervisor or an operating system, which initializes the system using dataflow graph at startup. When, the system is running, it should manage and monitor the execution of ECOs. The commercially available real-time operating systems (RTOSs) are unsuitable for this purpose because of some inherent differences between dataflow applications and regular applications.

Unlike a regular application, a dataflow application has lot of context switches, due to its component-based design. The ECO is defined as an entity, which takes inputs, process them and outputs the results, if any. This leads to a high frequency of inter-process communication in the application. In addition, the scheduling requirements of the ECO-dataflow architecture are different from regular applications. The scheduling of an ECO takes place when its firing rule is satisfied. These applications also demand dynamic ECO priorities that depend on the firing rule on which the ECO is being fired or executed.

We propose DARK as an operating system kernel ideally suited for these applications. DARK uses the underlying hardware of power-electronics controllers to drastically reduce the context-switching time, has dynamic priorities and scheduling mechanism that depends on the firing rules. It also provides typed data channels for efficient and reliable inter-process communication.

III. DARK- DESIGN AND IMPLEMENTATION

In a dataflow graph (DFG), the ECOs and data-channels are represented as nodes and arcs, respectively (Fig. 1). DARK uses a DFG descriptor file to initialize itself at startup. This file is provided by the application programmer and it contains the information about the nodes (ECOs) and arcs (data-channels) connecting them.

The application programmer also provides the implementation of the ECOs. The ECOs are implemented as C functions.

A. Kernel Structure

DARK maps each ECO in the DFG to a thread. A thread is an entity capable of executing concurrently with other threads and has its own runtime stack. The user provides the stack size, thread priority, ECO function with which the thread is associated and the thread-firing rule in the DFG descriptor file. The stack size required for a thread depends on the number of local variables used and the nested function calls in the ECO function. The priority given by the user is the initial priority that is assigned to the thread when the kernel starts. The kernel uses the firing rule associated with a thread for scheduling purposes. Each thread has a Thread Control Block (TCB) associated with it (Fig. 2), in which it stores all the control information.

```
typedef struct
{
    volatile ECO_Data p;
    ECO                eco;
    Context            thread_env;
    Thread_State      thread_state;
    Firing_Rule       firing_rule;
    Firing_Mask       in_ports_ready;
    int               wakeup_time;
    unsigned int*     stack_pointer;
    unsigned int      stack_size;
    bool              in_OS_call;
} TCB;
```

Fig. 2. Thread control block

All of the information in the TCB is initialized during the startup using the data supplied by the user. The Context structure is used for saving/restoring the runtime environment of the thread during the context switching. The variable `in_ports_ready` is used for scheduling purposes, whereas `wake_up_time` is used for time management. They will be described in the subsequent sections.

DARK maps each arc in the DFG to a typed data channel. The data channels are implemented as circular buffers. They have a Queue Control Block (QCB) associated with them to store the control information (Fig. 3).

```
typedef struct
{
    int                DC_id;
    Type_Tag           type;
    short int          element_size;
    Array_Descriptor   array_dimensions;
    Overflow_Style     overflow_style;
    int               front;
    int               rear;
    int               size_in_bytes;
    volatile int       size_in_elts;
    volatile int       num_entries;
    bool              blocked;
    Process*          source_thread;
    Process*          sink_thread;
    char              buffer[1];
} QCB;
```

Fig. 3. Queue control block

Each data channel has an id associated with it. The `type` field stores the type of elements that can be stored in the data channel, for type checking purposes. Although the type checking improves reliability of the system, it also introduces a small overhead, so the user is allowed to turn it off. DARK supports data-channels of all primitive data types along with complex data types like multi-dimensional arrays and strings. If a data-channel is full, the writing ECO may wait, overwrite the newest element or overwrite the oldest element. The `overflow_style` states the action to be taken in these cases.

The space for the element storage is allocated as a chunk of memory adjoining the QCB block, so that it can be accessed by using the variable `buffer` as a pointer (Fig. 4).



Fig. 4. QCB space allocation

DARK provides a simple API to the user for interacting with data channels. The user reads from or writes to a data-channel by calling a function of the form: `<operation>_<type>_DC (DC_ptr, data)`. Here the operation can be read or write and the type is the type of data to be read or written. `DC_ptr` is the pointer to the data channel obtained from the TCB. The separate functions for each data-type, aid type checking. Internally, these functions are implemented as macros, which call a single function that reads and writes raw bytes. Other functions for special operations like obtaining the status of data-channels or flushing all entries of the queue are also provided.

B. Scheduling and thread management

Unlike commercial RTOSs, DARK uses firing rules for scheduling. Thus, the user is not required to manually check for data in the incoming channels inside the ECO code and the thread starts executing only when the data is ready. The thread scheduling depends on its priority and the firing rule associated with it. A firing rule contains one or more records consisting of `firing_mask` and `Fired_ECO_priority` as the fields (Fig. 5). Firing mask is a binary number that specifies the input data channels on which the thread should wait for data, before being fired. For example, the firing mask 00000111 indicates that the thread (ECO) is ready to fire when it has data on channels 1, 2 and 3. An ECO can have more than one firing masks associated with it. As an example, it can take one action when it has data in three incoming channels, while it can take other action when it has data in only two incoming channels. The firing masks are arranged in the firing rule according to their priority, with the highest-priority firing mask at the beginning. The `Fired_ECO_priority` is the new priority that is assigned to the thread, when it is fired as a result of the corresponding firing mask. The current status of the input channels is maintained in the TCB field `in_ports_ready`.

time. When a thread is added to the waiting queue, a kernel variable `actions_pending` is set, which indicates that the scheduler should check the waiting queue.

Whenever the DARK scheduler is called, it compares the thread wakeup time with the system time. If a thread is ready, it is added to the ready queue. Only the first thread in the queue needs to be checked by the scheduler, because the waiting queue is arranged in ascending order.

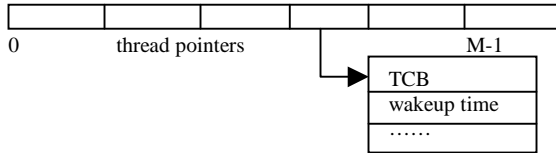


Fig. 8. A waiting queue

E. Interrupt handling

The C interrupt handler provided by the compiler is used in most of the commercial RTOSs. It uses a general approach to handle interrupts, in which all the registers are saved/restored while handling the interrupts. This method proves costly in system like a power electronics controller, where interrupts are coming consistently.

The compiler for Analog Devices SHARC microprocessor provides a second option of using the alternate register set for interrupt handling. As the C runtime uses only the primary register set, this option works well for the commercial RTOSs. But DARK cannot use this option because it uses both the alternate and primary register sets.

DARK uses an alternative approach to handle external interrupts, which is not only as reliable as the first option, but also provides performance comparable to the second option.

DARK has an event code associated with each action that can be taken on receiving an interrupt. Whenever the DSP receives an external interrupt, the FPGA writes an event code to a memory-mapped location and generates an external interrupt the microprocessor pin. The kernel on receiving this interrupt invokes a simple interrupt service routine (ISR) written in assembly. The ISR adds the event code to a circular event queue and sets the kernel variable `actions_pending`, before returning to the code that was being executed. As the ISR code uses only five registers, there is no need to save and restore all the registers. After that, it is the responsibility of the scheduler to taking actions on the basis of the event codes.

In addition to the general purpose interrupts explained above, DARK also supports clock interrupts and non-maskable interrupts (NMI). The clock interrupt ISR is also implemented in assembly and it increments the kernel variable `current_time`. This variable is used for time management. The NMI interrupt arises in emergency conditions and requires time critical response. In most conditions, it results in a call to the emergency shutdown procedure.

F. Mutual exclusion

The semaphore-based approach is the most common method for avoiding shared data problems. Though it is powerful and adopted by most RTOSs, it leads in a heavy cost in terms of performance.

Another approach adopted by RTOSs is to disable interrupts when entering a critical section and re-enable them on exiting the section. This option is unsuitable for large critical sections because the probability of missing critical interrupts is more.

DARK does not have a shared data problem because of the separation between operations of receiving and handling the interrupts. The interrupt receiver or the ISR just adds an event code to the event queue. The scheduler has the sole responsibility of handling the interrupts. Also, due to the non-preemptive behavior of DARK, the shared data problems between two ECOs never arise.

III. PERFORMANCE EVALUATION

An open-loop application, for PEBB based 3-phase inverter, was used for performance evaluation and comparison. This application had 7 ECOs (Fig. 9) and it was executed on a SHARC 21062 microprocessor using VisualDSP emulator provided by Analog Devices.

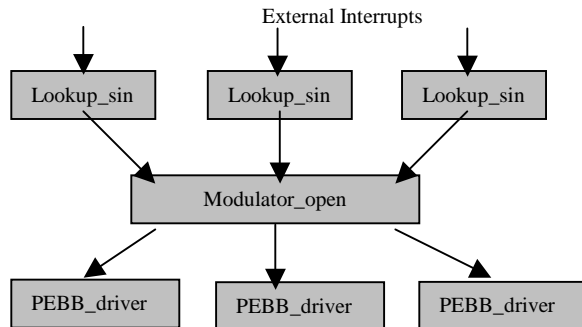


Fig. 9. Open loop application used for performance evaluation

DARK's performance was compared with MicroC/OS-II [1], a commercial RTOS for embedded systems. The open loop application was executed both on the DARK and MicroC/OS-II platforms for the comparison.

Part 1 of the Fig. 10 shows the instruction cycles taken by DARK and MicroC/OS to complete one switching cycle of the application. Part 2 shows the time taken by both kernels for start-up. As MicroC/OS-II does not provide the flexibility of using a dataflow graph as input, it takes less time for startup. DARK parses a dataflow graph at startup and automatically generates the threads and data-channels needed by the application.

In Fig. 11, the instruction cycles taken for read and write operations are shown. Part 1 shows the cycles taken for a write, while part 2 shows the cycles taken for a read operation. In the case of DARK, a part of the scheduling (operations like checking firing rules) is done during read and write operations. So, it takes more cycles as expected. The

time taken for DARK write operation is shown for the worst case, when an ECO is found ready to be fired and added to the ready queue.

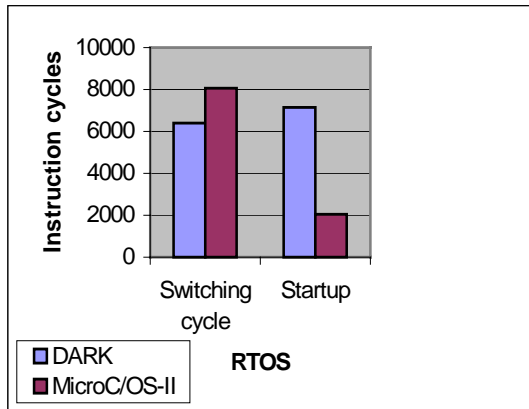


Fig. 10. Comparison of time taken for one switching cycle and startup time

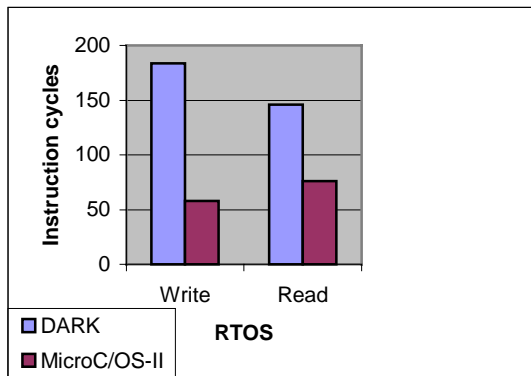


Fig. 11. Comparison of data channel read/write operations

Fig. 12 shows the comparison of the context switching time taken by DARK and MicroC/OS. Part 1 shows the ECO to kernel and kernel to ECO context switching time, and part 2 shows ECO to ECO context switching time.

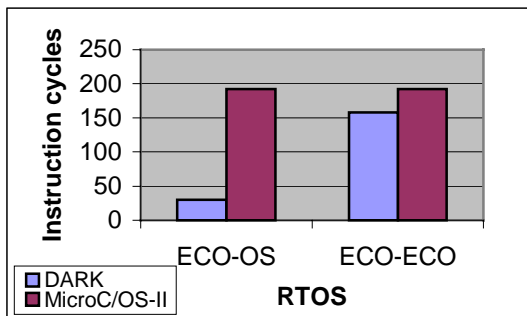


Fig. 12. Comparison of context switching time

Fig. 13 shows the total time taken by DARK for running the open-loop application for a single switching cycle divided into the major operations performed.

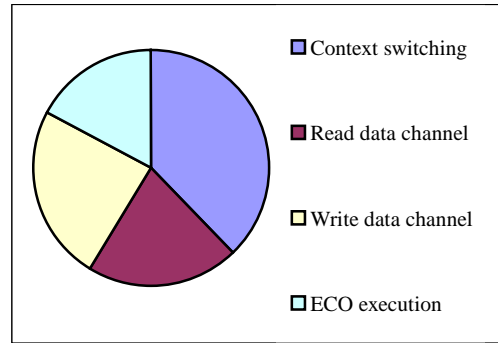


Fig. 13. Total time taken by DARK

IV. EXPERIENCES AND FUTURE WORK

DARK is still in its experimental stage and its performance is being optimized. There is a tremendous improvement in its performance over the last month, as shown in Fig 14. This was achieved by:

- (a) Converting the read and write functions into macros
- (b) Skipping the registers not used by C runtime while saving/restoring context
- (c) Turning the compiler optimizations on

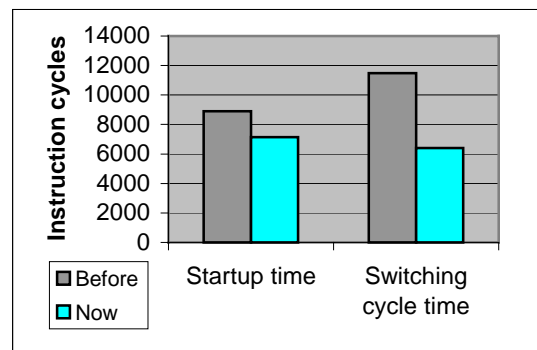


Fig. 14. Comparison between the last month's and present performance of DARK

In the future, the performance of the DARK will be further improved by code and data structure optimizations. The DARK will also be compared with VDK++, a commercial kernel by Analog Devices written in C++.

The design of a new version of DARK for distributed applications has been started. As the new PEBB architecture uses multiple boards, connected by a fiber-optic ring using PESNET [4] protocol, the development of the new version becomes important.

REFERENCES

- [1] Jean J. Labrosse, "MicroC/OS-II The Real-Time Kernel," *R & D Books*, October 1998.
- [2] Jinghong Guo, Stephen Edwards and Dushan Boroyevich, "Improved architecture of PEBB plug and play power electronics systems: Elementary control object (ECO) and dataflow," in *CPES 2001 Power Electronics Seminar and NSF/Industry Annual Review*, April, 2001.

- [3] D. Garlan and M. Shaw, "An introduction to software architecture," *Advances in Software Engineering and Knowledge Engineering*, vol. I, World Scientific Publishing Company, New Jersey, 1993.
- [4] Jerry Francis, Jinghong Guo, and Stephen Edwards, "Design of a Fault Tolerant Communication Protocol for Control of Distributed Power Electronics Systems," in *Proceedings of the 2002 CPES Annual Power Electronics Seminar*.
- [5] J.F. Bortolotti, P. Bernard, and E. Bauchet, "RTMK: A real-time micro kernel," *Dr. Dobbs's Journal*, May 1994.
- [6] K. Jeffay, D.L. Stone, and D. Poirier, "YARTOS: Kernel support for efficient, predictable real-time systems," in *Proc. Joint of Eighth IEEE Workshop on Real-Time Operating Systems and Software and IFAC/IFIP Workshop on Real-Time Programming*. Atlanta, GA. Vol. 7, No. 4, Fall 1991 pp. 8-13.