

# Evaluating the Performance Overhead of Object-Oriented Techniques in Embedded Software Design

Sumithra Bhakthavatsalam and Stephen H. Edwards

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

*Abstract*—This research involves experimental comparisons of the performance and memory overhead of an OO embedded real-time kernel with those of the non-OO version of the kernel. In this paper we present a discussion of the techniques used to produce an efficient OO design, the design and implementation of the OO kernel and the results of the experiments carried out to compare the performances of the OO and the non-OO kernels.

## I. INTRODUCTION

Object-oriented programming offers code reusability, easy maintenance of software and the enforcement of data encapsulation and modularity. These benefits are congruent with the goals of PEBB (Power Electronics Building Block) systems, which are meant to introduce modularity, standardization and reusability in power electronics systems [2]. However, the performance issues that OO programming introduces make it prohibitive especially in the context of real-time systems, where efficiency is a serious concern.

DARK++ is an OO redesign of an embedded real-time kernel, DARK (Dataflow Architecture Real-time Kernel) written in C. The kernel and a representative set of power electronics control applications are being implemented in C++. Experimental evaluation of the performance of these applications is being performed in order to assess the efficiency of the OO kernel vis-à-vis that of the non-OO kernel.

The reader is advised to refer to [1] for a discussion of the performance issues in OO systems and the design and functionality of some important classes in the system.

DARK++ is designed based on Dataflow Architecture that supports asynchronous inter-component communication [2]. Dataflow Architecture is characterized by a large number of components in order to facilitate modularization, high frequency of inter-component communication due to the large number of components, and special scheduling requirements due to the data-driven nature of dataflow applications. These characteristics necessitate special attention to the design of a dataflow system; in particular, the components in the system must be high-speed components, the inter-component communication must be efficient, and context switching must be efficient.

Over and above the dataflow considerations, the OO performance issues also need to be considered in order to achieve a high-speed kernel.

Our kernel, just like its non-OO counterpart, provides four versions – *preemptive multithreaded*, *non-preemptive multithreaded*, *single-threaded dynamically scheduled*, and *single-threaded statically scheduled*. The kernel is configurable so that one can choose to run an application with any of these versions of the kernel.

Since a PEBB system could comprise multiple processors, we could have components controlled by different processors communicating with each other. This makes it necessary to handle intra-processor as well as inter-processor communications. For this purpose, we have two types of data channels; while *Message Queues* are used to connect components controlled by the same processor, *Mailboxes* are used for inter-processor communications. A mailbox may be viewed as a special case of a message queue. It is nothing but a message queue of unit size since inter-processor communications take place one unit of data at a time.

This paper contains the results of the performance experiments carried out on DARK++ and DARK and presents a comparison of the two. We also present here, a description of the efficient OO design of the DARK++ system and some important high-speed kernel features (that are not necessarily specific to an OO kernel). The description of these design features will help understand the effect of each of them on the performance of the system, which is reported later in the paper.

The following section discusses techniques that have been employed to overcome the OO performance issues. It describes in particular, how these have been employed in the DARK++ system. Section III presents a discussion of the most important kernel features like thread management, high-speed context switching, interrupt handling and mutual exclusion. These are implemented identical to the way they have been implemented in the DARK. In Section IV we provide a discussion of the four configurable versions of the kernel and their features and a few implementation details that would help us understand how the different options impact performance. Section V contains a comparison of DARK and DARK++ performance. We present the DFGs (Dataflow Graphs) of two control applications – the Open-loop three-phase Inverter, and the Closed-loop three-phase Inverter, and report the performance data for these two

applications that were run on the different versions of the kernel, with message queues and with mailboxes. Further, we compare the performance of our kernel with that of the DARK and present an analysis of the data gathered in terms of the fraction of time spent on each of the different tasks – (scheduling, context switching, process execution, data channel operations and ready queue operations), the factors that contribute to overhead, etc. The paper concludes with the most important lessons learnt so far from the research and a brief note on future work.

## II. TECHNIQUES FOR EFFICIENT OO DESIGN

The critical factors that cause performance overhead in an OO system are heap-allocated memory, dynamic binding and method call overhead. In this section we describe OO design techniques we have adopted, that address the high performance objective for our system.

Object-oriented systems are characterized by extensive use of dynamic memory allocation, dynamic creation and destruction of objects, and use of pointers. These lead to overhead in memory management due to the repeated dereferencing of pointers, pointer chasing and cache inefficiencies. Therefore in systems where performance is critical, dynamic creation and destruction of objects is best avoided. In the DARK++ system, all objects are created statically, before the scheduler begins execution; i.e., all objects required by the system are created during start-up and never during the execution of the scheduler. Hence overhead due to dynamic memory allocation is avoided.

Dynamic Binding is another characteristic of OO systems that poses a threat to performance. With dynamic binding, decisions about the association of a method call with the actual location of the entry point of the corresponding machine code are deferred until run-time. Since the resolution happens at run-time rather than at compile time, one pays for it when a system is executing; in other words, it slows down the performance of the system. Virtual methods in OO programs are implemented by dynamic binding and hence degrade performance. In the DARK++ system, virtual methods are avoided where possible. Only the `Implementation` method in the `ECO1` class [1] is virtual since the structure of the ECO class hierarchy dictates the use of a virtual method. The overhead due to this virtual method, however, plays a role only in the first switching cycle of the scheduler when the `Implementation` method is explicitly called. In further cycles, control goes to the ECO through a `longjmp` into the implementation code, and hence there is no additional overhead due to the virtual method.

---

<sup>1</sup> Elementary Control Object - a functional entity in the PEBB system that can be used independent of the system details, in a variety of systems. It serves as a modular, standardized and reusable component. ECOs are modeled as processes in the DARK++ system. The kernel schedules and runs them.

Another important factor that causes performance overhead in OO systems is method calls. Method calls introduce performance concerns due to the time spent in setting up the stack frame and transferring control to the called methods. The OO paradigm inherently means more method calls. Classes typically have a greater number of smaller methods. Since most behaviors use a greater number of method calls arranged in deeper call trees, this pattern leads to additional overhead in comparison to non-OO languages. All methods in the DARK++ system are inlined to nullify the effect of method calls. Although the “inline” keyword gives no guarantee about the method actually being inlined, but rather, serves as a hint to the compiler, this is the best that we can achieve while still retaining a structured and neat design and implementation. Some frequently used blocks of code that are macros in DARK are retained as macros in DARK++ too. These features lead to significant performance gains.

Inlining has the obvious disadvantage of introducing a large space requirement. In the context of embedded systems since memory has to be used judiciously, we need to take consideration of this fact. Therefore, methods that are called as part of start-up, before the scheduler actually begins execution, are not inlined. This leads to memory savings where possible.

Besides the discussed measures taken for improving the efficiency of the DARK++ system, some other features such as effective use of (user-defined) templates leads to an appreciable performance, since template instantiations are performed during compile time.

We have discussed some important OO design tricks for boosting efficiency. Besides these, there are also some other important features meant for performance gain. In particular, some critical parts of the DARK++ kernel such as those for context switching and interrupt handling are written in assembly. The context switching implemented in assembly employs the dual-register set hardware provided by the DSP and contributes to a significant performance boost. This is explained in Section III (B). These features are the same as in DARK.

## III. KERNEL FEATURES

This section briefly discusses the design and the salient implementation details of the various kernel features. These are implemented identical to the way they are implemented in the non-OO kernel.

### A. Thread Management

The processes in the control application that are scheduled and run by the kernel are the ECOs (Elementary Control Objects) in the PEBB system [2].

An ECO can be viewed as a process that executes its `Implementation` code provided by the ECO designer. The various possible states of these processes are: *ready*, *blocked*, *wait\_for\_fire*, *timed\_wait*, *timed\_wait\_for\_fire* and *dead*. When the kernel starts, each thread is in the

*wait\_for\_fire* state. A process is in *ready* state once its required input data channels have data tokens in them. The process is *blocked* when it tries to read from an empty data channel or write to a full data channel.

After every *read* operation on a data channel, the status of the source ECO (ECO that writes to this data channel) is checked. If the source ECO is found to be blocked, then it is unblocked. Similarly, after every *write* operation, the mask of its sink ECO (ECO that reads from this data channel) is updated; i.e., the bit corresponding to the data channel in question is set. Thus, while a *read* operation could unblock a process blocked on a data-channel, a *write* operation could fire it.

The *wait\_to\_fire* function can be used to fire the ECO again. If the ECO is not ready for firing, it goes into the *wait\_for\_fire* state. The user can also delay the execution of the ECO for a pre-determined time, which puts the ECO into *timed\_wait* state. The *timed\_wait\_for\_fire* state is a combination of *wait\_for\_fire* and *timed\_wait*. An ECO in this state can be fired if a firing mask becomes true or if the time period elapses. The ECO goes into the *dead* state once it finishes execution. Figure 1 is the thread state diagram.

### B. Context Switching

Many digital signal processors used in embedded control systems, ADSP 21160 being one, have two sets of registers for increased performance – the *primary* set and the *alternate* set. DARK++, which runs on ADSP 21160, uses the primary register set for the kernel and the alternate register set for the process threads. Due to the use of two independent sets of registers for the kernel and the process threads, all that is required during a transfer of control between the two is flipping of a bit in a control register, which denotes the current *mode* (indicating whether the currently running thread uses primary set/secondary set), and saving/restoring some key status registers. DARK++ uses customized *setjmp* and *longjmp* assembly language procedures that selectively save/restore just these required registers.

Since most context switches in dataflow applications occur between the scheduler and executing threads, minimizing the cost of such switches increases the performance significantly. The use of the dual-register-set architecture in DARK++ for high-speed context switching between the scheduler and application threads has been found to reduce the switching time by 80%.

### C. Interrupt Handling

There are many RTOSes (Real-time Operating Systems) that support interrupt handling through the use of compiler-provided mechanisms, using C functions that can be used as interrupt routines. This method involves a substantial overhead in context switching, since all registers are saved and restored while handling interrupts. The C/C++ compiler provided by Analog Devices for its SHARC DSPs (Digital Signal Processors) supports this approach, and in addition,

also provides the option of using the alternate register set for interrupt handling (since the C/C++ runtime uses only the primary register set). However, DARK++ cannot use this option, since it uses both the alternate and primary register sets, as explained in Section B.

DARK++ uses an alternative approach for handling external interrupts. This method provides performance comparable to that of using the alternate register set for interrupt handling. Here, rather than placing actions directly in the interrupt handler itself, DARK++ uses a minimal footprint handler that simply logs incoming events into the *event queue*, which is managed by the DARK++ scheduler. The interrupt handler runs in the currently active register set and only needs to save and restore a couple of registers. It logs a 32-bit code representing the interrupt that was received, into the event queue (a circular buffer of incoming events) and then returns control to the kernel. The status of the event queue is reflected by a variable, *actions\_pending*.

DARK++ also supports clock interrupts and non-maskable interrupts (NMI). The clock interrupt ISR is written in assembly and simply increments the kernel data member, *current\_time* that is used for time management. Only a few registers required for incrementing a variable are saved and restored in this ISR. NMI is used for emergency condition notification and requires a time critical response. In most cases, it results in a call to the application's emergency shutdown procedure, bypassing all other kernel as well as application code.

### D. Mutual Exclusion

Since threads have no shared memory and communicate only through data channels, most mutual exclusion problems do not arise in DARK++.

### Volatile Declarations

In the embedded system context, it is imperative to have an efficient and optimized executable. With the optimizer enabled, typically several variables are cached in registers to make data fetches efficient.

As explained in Section B, DARK++ uses the dual-register-set hardware for efficient context switching. This means that the kernel and the process threads work with distinct registers. Consequently, the two threads use different registers to cache the same data and this could lead to inconsistencies. It should be noted here that if all methods that manipulate data members private to their class were called through normal method call, then this problem would not arise, as these data members would not be cached in their callers. However, in the interest of our high performance objective, methods are inlined, and this causes even the private data to be cached in registers, as part of the caller's thread. Therefore all the data that can potentially be accessed by both the kernel and the process threads have been

identified and declared *volatile* to ensure that they always get accessed from the memory instead of from registers.

The *non-preemptive* version of DARK++ does not invoke the scheduler on every OS call but instead, runs each thread until the thread suspends itself waiting for input data. Thus

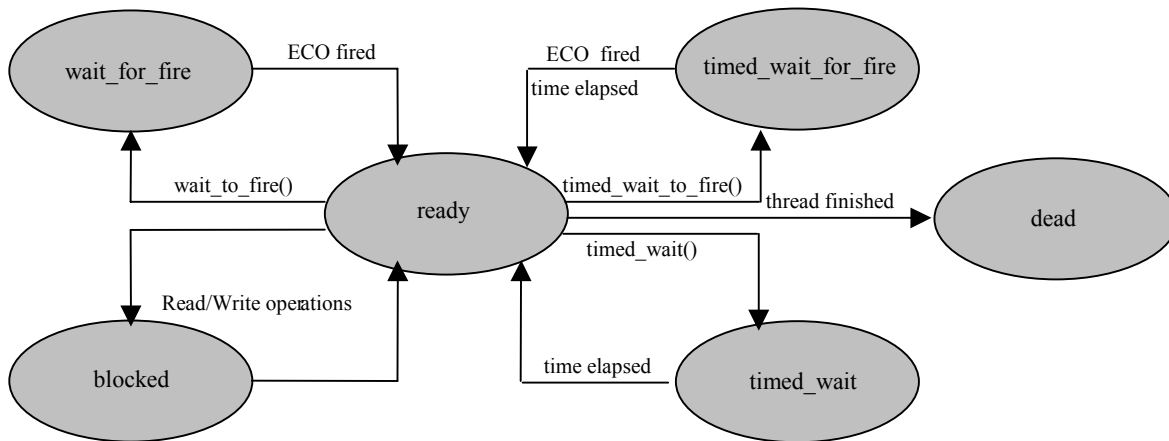


Figure 1. Thread State Diagram [3]

#### IV. CONFIGURABLE OPTIONS

The DARK++ kernel can be configured to yield four distinct versions. These are obtained by selectively retaining/removing certain kernel features. Removal of features leads to an increase in performance with a concomitant reduction in run-time flexibility. The application designer can select the most appropriate DARK++ version for a given application's requirements. Table 1 lists the features in different versions.

The *full-featured* version of DARK++ has nothing disabled, and is a multi-threaded preemptive kernel. This version of the kernel schedules threads dynamically based on their firing rules and priorities. After every OS call (*read* and *write* operations), the scheduler is invoked to check for higher- and equal-priority threads. A context-switch takes place if and only if a higher- or equal- priority thread is ready. Preemption by an equal-priority thread ensures fair scheduling.

we avoid the overhead of calling the scheduler after every *read* and *write* operation by sacrificing immediate response to higher-priority threads.

The other two versions of DARK++ are single-threaded and avoid the time spent in context switching, thereby giving a significant performance boost to the application. The single-threaded approach is ideal for monotonic applications that execute sequentially, but unsuitable for applications that are highly dynamic. While the dynamically scheduled single-threaded version of DARK++ uses firing rules and priorities to select a process for execution, the statically scheduled single-threaded version uses a precomputed firing order for threads, eliminating all use of priorities and firing rules, and is therefore the fastest.

Data Channels are of two types – message queues and mailboxes. A mailbox is an inter-processor data channel and is nothing but a special case of message queue where the size of the queue is one. This obviates the need for queue arithmetic and is therefore more efficient.

Versions	Features		
	Preemptive	Multithreaded	Dynamically Scheduled
Full-featured DARK++	X	X	X
Non-preemptive DARK++		X	X
Single-threaded dynamically scheduled DARK++			X
Single-threaded statically scheduled DARK++			

Table 1. Configurable options in DARK++ [3]

## V. PERFORMANCE EVALUATION

This section discusses the results obtained during the performance evaluation experiments of the kernel. The experiments were conducted on Analog Devices-SHARC 21160 digital signal processor. The Analog Devices VisualDSP++ simulator was used to run the experiments and collect profiling information. Experiments were conducted on two dataflow applications that are used in the power electronics control domain. The first application is an open loop control for a three-phase inverter, depicted in Figure 2. This application consisting of seven ECOs, looks up and derives appropriate duty cycle commands to send to each phase leg of the power stage being controlled. The DFG for the second application represents a feedback-based closed loop control for a three-phase inverter consisting of nine ECOs and is shown in Figure 3. It is a slightly more advanced control scheme for the same power stage hardware.

We present the results obtained by comparing the performance of the different versions of DARK++ on these two control applications. Figures 4 and 5 show the data for the open-loop application conducted using message queues and mailboxes respectively. Figures 6 and 7 show the corresponding data for the closed-loop application. The

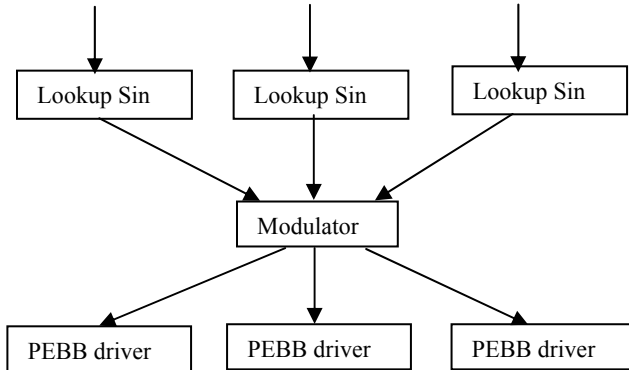


Figure 2. Open-loop three-phase inverter

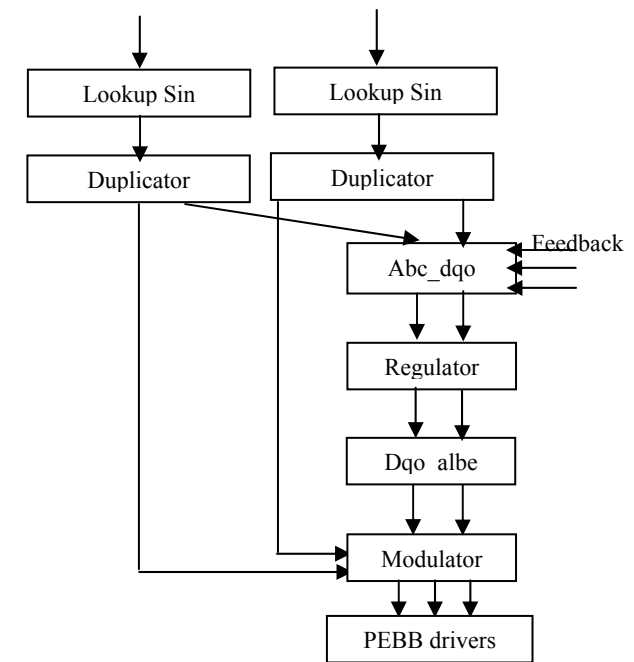


Figure 3. Closed-loop three-phase inverter

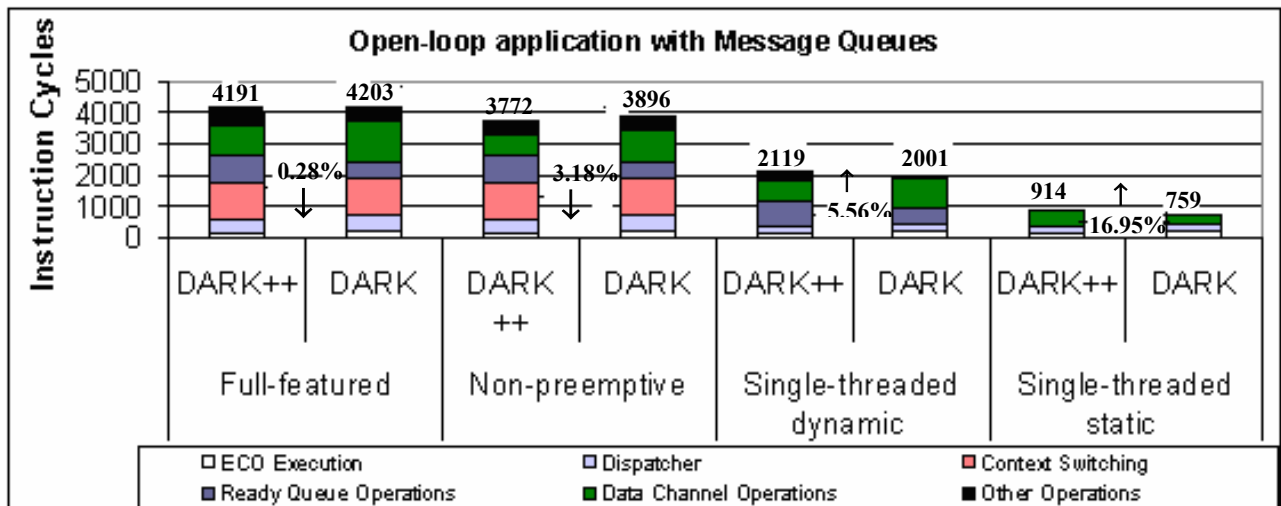


Figure 4. Performance Data for Open-loop three-phase inverter – with Message Queues

graphs show the total number of instruction cycles taken by DARK and DARK ++ for one switching period of the kernel for the mentioned applications. This is broken down into six categories – ECO execution, dispatcher, context switching, ready queue operations and other operations. We can compare the contributions of each of these factors to the execution time for the application in the case of DARK++ with those in the case of DARK. The total number of instruction cycles is shown at the top of each bar and the percentage increase or decrease in execution cycles for DARK++ relative to DARK is shown in between the two bars. The graphs contain data for all four versions of the kernel.

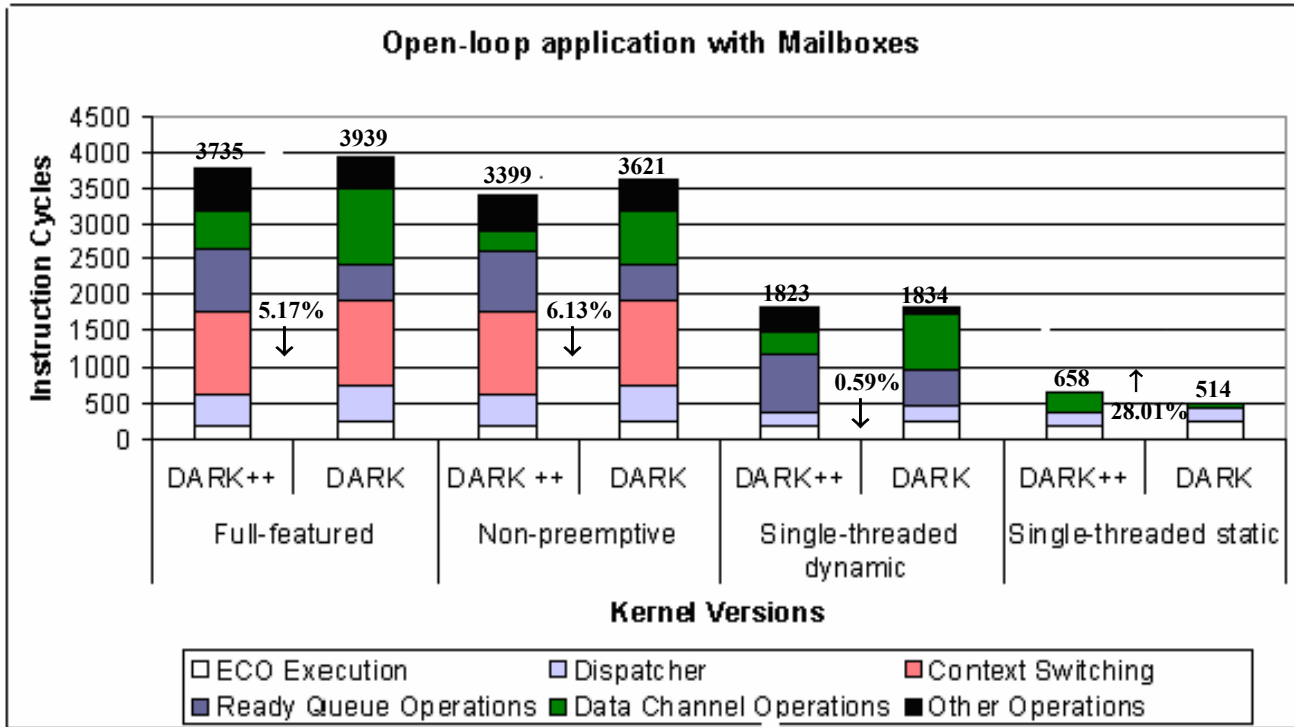


Figure 5. Performance Data for Open-loop three-phase inverter – with Mailboxes

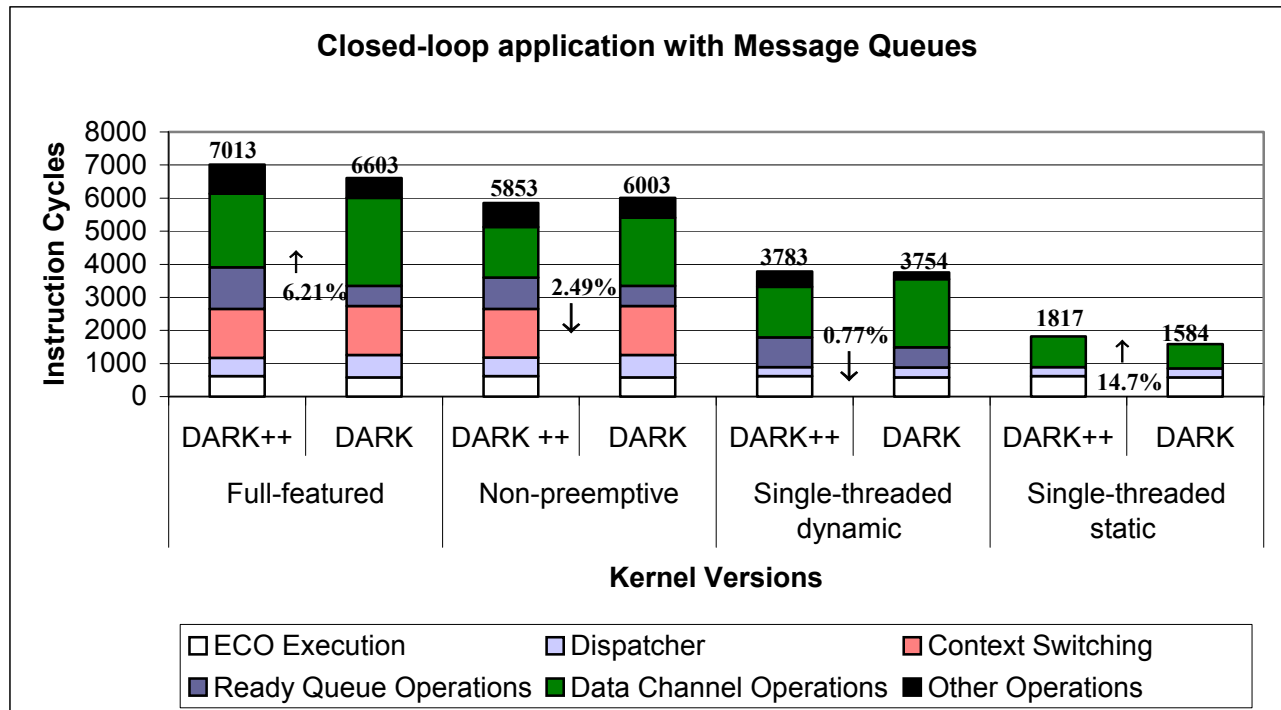


Figure 6. Performance Data for Closed-loop three-phase inverter – with Message Queues

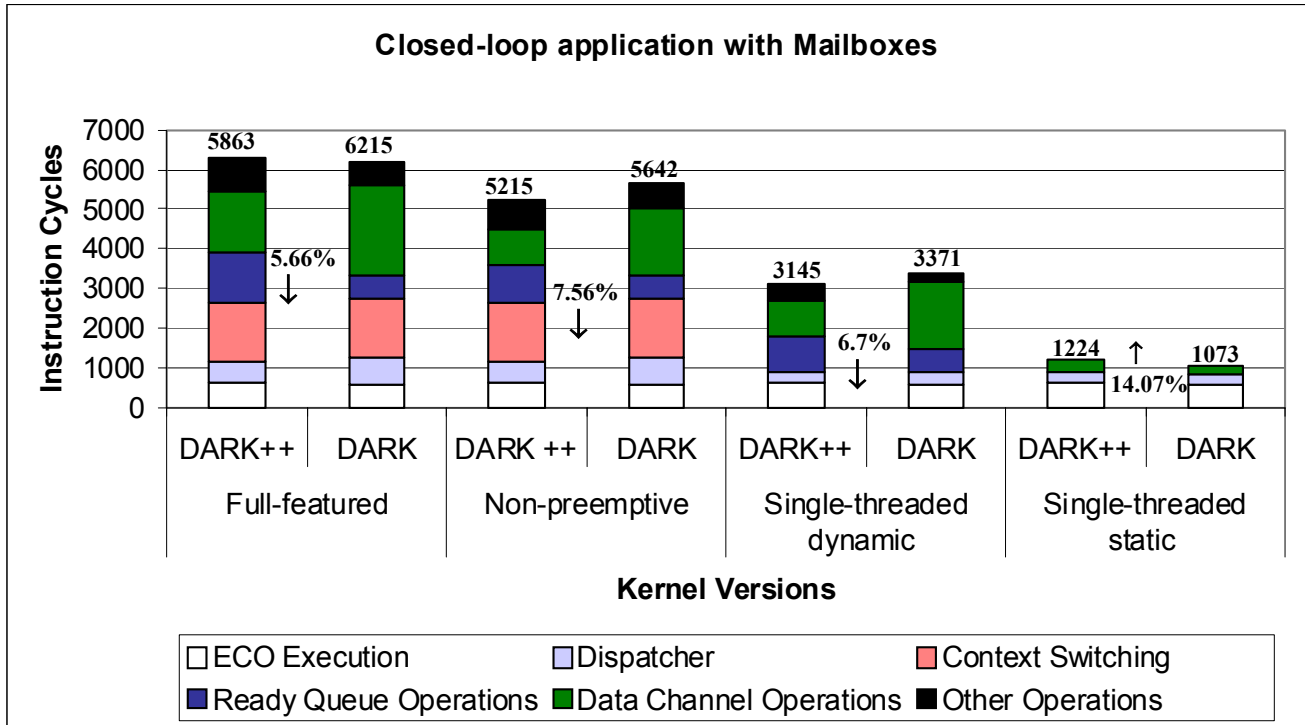


Figure 7. Performance Data for Closed-loop three-phase inverter – with Mailboxes

The above results indicate that DARK++ has performance comparable to that of DARK. The full-featured version of the kernel, in particular, outperforms that of DARK for the open-loop application and for the closed-loop application running on mailbox data channels.

It may be observed that the context switching times are equal in DARK and DARK++ since both use the same custom *setjmp* and *longjmp* assembly functions to accomplish this. While the ECO execution and the scheduler execution times are comparable in the two cases, the ready queue operations have a significantly higher contribution to the entire execution time in the case of DARK++ than in DARK. This is because all accesses to any of the ready queue data members have been counted under this category and there are a number of calls to such operations – e.g., calls to a method that returns the number of items present in the ready queue. Such calls are made both by the scheduler, as well as from the data channel operations in the case of preemptive scheduling, to check for other high-priority ready processes that may be waiting. Typically, it has been observed that simple methods that return the value of a data member take between 7-9 instruction cycles. Hence even if such a method is inlined by the compiler, there is an overhead incurred by the frequent use of such methods.

The “other operations” category also takes more number of instruction cycles in the case of DARK++ than in DARK due to the same reason as mentioned above. There are some generic methods that are frequently used by

various callers to retrieve some data members and these introduce significant overhead.

It is worth noting that a great many operations that are specified as macros in DARK are class methods in DARK++, with the “inline” keyword. Therefore, while these operations are guaranteed to be preprocessed and efficient in DARK, many of them are not inlined by the compiler in DARK++. A better performance could have been achieved with DARK++ if it were possible to guarantee inlining of all methods. Also, the unpredictability of a method actually being inlined could lead to marginal irregularities in performance.

The single-threaded versions of the kernel need do no context switching and hence the number of instruction cycles for the category is zero. The single-threaded statically scheduled kernel makes use of a precomputed order to execute the processes sequentially and hence does not use the ready queue.

An important observation to make is that the single-threaded versions of DARK++ are slow. This is because, while the multithreaded kernel uses calls to *setjmp* to execute processes in all but the first switching cycle of the kernel, the single-threaded kernel always makes an explicit call to the ECO Implementation method and since this is a virtual method, there is a considerable overhead introduced due to the dynamic resolution to effect the call.

From the data gathered on these two applications and from the above discussion, we may conclude that careful design in OO paradigm can yield appreciable performance. It may be worth enumerating the main difficulties with the

OO approach. As we have seen, it very naturally imposes the need for more method calls. While one can choose to specify such methods with the “inline” keyword, since it relies on the discretion of the compiler, there may be inefficiencies.

Another related point is that, while it is often worthwhile to specify some frequently used (small) operations as macros in C, it may be inappropriate to do this in C++ (an OO language) where more often than not, we want operations as methods in a class and specifying these as macros might lead to a sloppy design. In DARK++, as stated earlier, we have specified a few generic operations used by the data channel *Read* and *Write* methods as macros. The question really is a tradeoff between elegance and performance.

It is best to avoid virtual methods as these rely on dynamic binding, which impact performance considerably. However, if the system being designed compels the use of virtual methods, one necessarily pays for the Vtable look-up and resolution during run-time.

There are some important points to remember while working on OO design for performance-critical systems. Use of dynamic memory allocation, perhaps by creating objects “on the fly” is a bad idea for a system where performance is critical. This should be avoided.

Templates are often handy and neat to use in the OO design and user-defined templates do not have any inherent performance concerns associated with them since template instantiations take place before run-time.

Where possible, virtual methods should be avoided as they impact performance tremendously. The performance numbers for the single-threaded statically scheduled version of DARK++ reflect this fact very clearly.

Although most programmers have a diffidence to use OO languages when considering performance of the system, a great deal of efficiency is achievable with careful design and implementation.

## VI. CONCLUSIONS AND FUTURE WORK

The experiments conducted thus far have shown encouraging results. In general, while using OO languages for performance-critical systems, dynamic memory allocation and dynamic binding should be used sparingly. While it may be hard to avoid method call overheads, it is best to try inlining the crucial methods.

So far we have gathered the performance data on two control applications – the open-loop three-phase inverter and the closed-loop three-phase inverter. In the future, we will conduct experiments on two more power control applications that have more number of ECOs and more inter-ECO communications. This will help us validate better, our current inferences. After this, the kernel will be used to run the applications on the actual hardware.

## ACKNOWLEDGMENT

This work was supported primarily by the Office of Naval Research under Award Number N00015-01-1-0954 and secondarily by the ERC Program of the National Science Foundation under Award Number EEC-9731677.

## REFERENCES

- [1] Sumithra Bhakthavatsalam and Stephen H. Edwards. "[Applying object-oriented techniques in embedded software design](#)," CPES 2002 Power Electronics Seminar and NSF/Industry Annual Review, April 2002.
- [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] Kuljeet Singh and Stephen H. Edwards. "[Design and Evaluation of an Embedded Real-time Micro-kernel](#)"