Synthesis of Time-Constrained Multitasking Embedded Software

ANDRÉ C. NÁCUL and TONY GIVARGIS
University of California, Irvine

In modern embedded systems, software development plays a vital role. Many key functions are being migrated to software, aiming at a shorter time to market and easier upgrades. Multitasking is increasingly common in embedded software, and many of these tasks incorporate real-time constraints. Although multitasking simplifies coding, it demands an operating system and imposes significant overhead on the system. The use of serializing compilers, such as the Phantom compiler, allows the synthesis of a monolithic code from a multitasking C application, eliminating the need for an operating system. In this article, we introduce the synthesis of multitasking applications that execute in a timely manner. We incorporate the notion of timing constraints into the Phantom compiler, and show that our approach is effective in meeting such constraints, allowing fine-grained concurrency among the tasks. As an additional case study, we present the implementation of a software-based modem and show that real-time applications such as the modem have guaranteed performance in the serialized code generated by the Phantom compiler.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Design

Additional Key Words and Phrases: Code serialization, multitasking, real-time embedded software, software synthesis

1. INTRODUCTION

The complexity of embedded system designs is rising steadily and software importance is growing significantly. Embedded devices incorporate multiple functionalities, complex user interfaces, different protocols, and security mechanisms. Many of these functionalities are implemented in software, which is more flexible when compared to ASICs, provides higher reuse, and yields shorter time to market.

Multitasking greatly simplifies the design of today's complex embedded systems. Embedded systems are inherently concurrent, and the concurrent

This work was supported by NSF award CCR-0205712 and Capes Foundation award 1054/01.5.
Authors' address: Department of Computer Science—Center for Embedded Computer Systems, 444 Computer Science, University of California, Irvine, Irvine, CA 92697; email: {nacul, givargis}@uci.edu.

programming paradigm is a natural choice for embedded software design. Such concurrency support is usually provided by an operating system (OS) layer, and comes at the cost of memory and processing cycles to control the schedule and execution of the tasks. A multitasking OS, however, is not available for every processor type and its derivatives. The limited processing capability of some processors, specially microcontrollers, restricts the possibilities of multitasking. Moreover, porting operating systems to every variant of a processor is an extenuous job.

Equally important in embedded software is adherence to timing constraints. In embedded real-time systems, where correct timely execution is as important as correct computation, multitasking is widely used. In such systems, it is important to provide guarantees about the timing behavior of the software. The quality of the timing guarantees is what differentiates a system that can be used in a hard real-time environment from a soft realtime environment.

To address the need for multitasking and timing constraints support in embedded software, we have developed the *Phantom* serializing compiler. With Phantom, we provide a fully automated tool to generate a single threaded, ANSI C program (i.e., the output) from a C/POSIX program (i.e., the input). The input is a C program, extended with POSIX to specify multitasking primitives. The output is strict ANSI C code, without any POSIX or other multitasking references. Phantom synthesized code serializes the execution of a multitask application and embeds the scheduler and multitasking control into a single monolithic program. The generated code is highly tuned for the input application, and, since it is ANSI C, can be compiled with the processor's native toolchain. Therefore, Phantom facilitates the multitasking programming model at a lower cost and higher efficiency, providing a way to support concurrency for resource constrained microcontrollers and for systems where an operating system port is not available or possible. Due to the generic characteristics of the input code, any multitasking C code can be transformed by the Phantom compiler.

In our previous work, we have introduced the Phantom compiler, discussing the partitioning problem [Nacul and Givargis 2004] and detailing the structure of the synthesized code [Nacul and Givargis 2005a]. In the previous versions of the compiler, there was no support for automatically specifying timing constraints, and therefore influence the timing behavior of the generated code precisely. This article, introduces our framework to support timing constraints, and incorporates timing guarantees in the Phantom synthesized code.

Timing constraints support in the Phantom compiler requires analysis of the source code and estimates of the execution time of each scheduled block when generating the serialized, monolithic program. In this work, timing estimates are based on application profiling. Therefore, our approach is mainly targeted at soft real-time applications, because profiling does not provide estimates that are accurate enough for hard real-time systems. Nevertheless, our methodology can be applied to hard real-time systems as well, replacing the profiling phase by user-supplied WCET estimates for each segment of the application. As discussed later in the article, WCET estimates are simpler in the Phantom compiler. Specifically, in order to allow scheduling and context switching, the

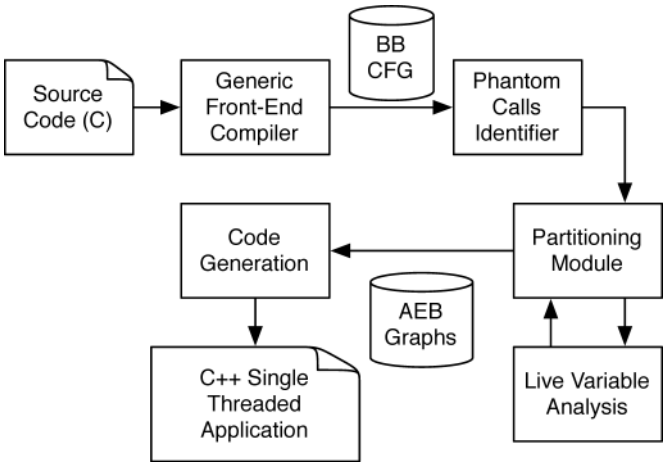


Fig. 1. Phantom compiler architecture.

original code is partitioned into smaller, simpler blocks of execution. Therefore, WCET estimation is a less complex process.

This article is organized as follows. Section 2 introduces the phantom compiler. Section 3 discusses the partitioning in phantom and how code partitioning can be used to control timing behavior of the application. In Section 4, we present the framework used to generate applications that meet timing constraints with the Phantom compiler. Experimental results are discussed in Section 5. Related approaches are introduced in Section 6, and we conclude in Section 7.

2. THE PHANTOM COMPILER

This section presents an overview of the Phantom compiler and its code synthesis process. For a complete and detailed description, refer to the previously published work Nacul and Givargis [2004]; Nacul and Givargis [2005a].

Input to Phantom is a multitasking program P_{input}, written in C. The multitasking is supported through the native Phantom API, which is a subset of the standard POSIX interface (Go online to the POSIX Open Group Web site `http://www.opengroup.org`). The POSIX primitives provide functions for task creation and management as well as a set of synchronization variables. Output of Phantom is a single-threaded strict ANSI C program P_{output} that is equivalent in functionality to P_{input}. More specifically, P_{output} does not require any OS support and can be compiled by any ANSI C compiler into a self-sufficient binary for a target embedded processor.

Figure 1 shows the block diagram of Phantom. The multitasking C application is compiled with a generic front-end compiler to obtain the basic block (BB) control flow graph (CFG) representation. This intermediate BB representation is annotated, identifying Phantom primitives. The resulting structure is used by a partitioning module to generate nonpreemptive blocks of code, which are called *atomic execution blocks* (AEBs), to be executed by the scheduler. Every task in the original code is partitioned into many AEBs, generating an AEB

graph. Then a live variable analysis is performed on the AEB graphs and the result is fed back to the partitioning module to refine the partitions until acceptable preemption, timing, and latency are achieved. When a final partition is achieved, the scheduling and context switching information is included for each of the resulting AEBs in the AEB graphs. The resulting AEB graphs are then passed to the code generator to output the corresponding ANSI C code for each AEB node, resulting in the final ANSI C single-threaded code.

When compared to an RTOS kernel, the serialized code is very different, and optimized in a number of ways. First, the serialized code has a more efficient context switching, since it is possible to know which variables are live at the context switch point, optimizing the data that needs to be saved. The code for saving and restoring context is simplified, and embedded in the serialized code for each context switching point. Additionally, the serialized code contains the information of the next task to be executed, simplifying the scheduling decision. Nevertheless, some drawbacks of multitasking are also present in the serialized code. Whenever there is a context switch, a new task starts executing, and cache, TLB and other architecture elements are affected. This is an inherent characteristic of multitasking, and is not changed by Phantom. However, the serialized code can be more efficient than the traditional RTOS. If tasks are implemented as separate processes, which require more complex context information, the serialized code presents a significant improvement.

2.1 Scheduling and Synchronization

We define the basic unit of execution, scheduled by the scheduler, an atomic execution block (AEB). An AEB is a block of code that is executed in its entirety prior to scheduling the next AEB. A task T_i is partitioned into an AEB graph whose nodes are AEBs and edges represent control flow. Consider the example Control flow graph (CFG) shown in Figure 2. Figure 2(a) shows the output of the compiler front-end that is fed to the partitioning module, annotated with the Phantom primitives. The partitioner adds two control basic blocks, *setup* and *cleanup*, as shown in Figure 2(b), and subsequently divides the function code into a number of AEBs, as shown in Figure 2(c), in a process we call *phantomization*.

Figure 2(c) shows the AEB graph of the original CFG as being composed of AEBs *aeb_0*, *aeb_1*, *aeb_2*, *aeb_3*, *aeb_4*, and *aeb_5*. We note that an AEB node may be composed of one or more basic blocks. The termination of an AEB region transfers the control back to the scheduler. The scheduler, then, has a chance to activate the next AEB, from either the same task or from another task that is ready to run.[1]

It may happen that a function f in the original input code is phantomized (i.e., partitioned) into more than one AEB, each one of them being implemented as a separate region of code. In that case, there is a need for a mechanism to

[1]An AEB is a structure similar to a hyperblock in trace scheduling [Fisher et al. 2002]. An AEB, however, is able to cross loop boundaries, which is a limitation of hyperblock structures. Furthermore, the borders of AEB are determined by synchronization and timing constrains, instead of only the control flow as in hyperblocks.

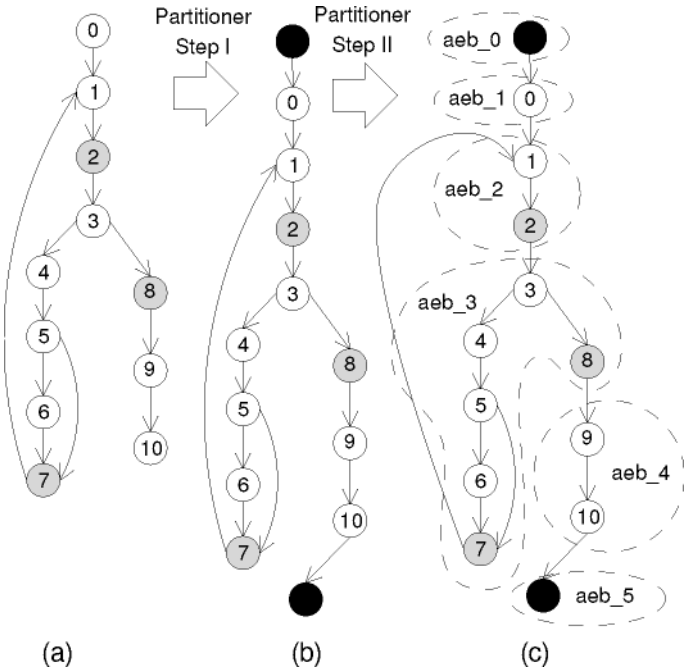


Fig. 2. Example of CFG transformations.

save the variables that are live on transition from one AEB to the other, so that the transfer of one AEB to another is transparent to the task. Also, every task must maintain its own copy of local variables during the execution of f as part of its context. Phantom solves this issue by storing the values of local variables of f in a structure inside the task context, emulating the concept of a *function frame*. The frame of a phantomized function f is created in the f_{setup} block, and cleaned up in the last AEB of f. These operations are included by the partitioner for every function that needs to be *phantomized*. They are represented by the dark nodes in Figure 2(b).

During runtime, there is a need to maintain, among others, a reference to the next AEB node that is to be executed some time in the future, called `next_aeb`, in the context information for each task that has been created. When a task is created, the context is allocated, the `next_aeb` field is initialized to the entry AEB of the task, and the task context is pushed onto the queue of existing tasks to be processed by the embedded scheduler.

The embedded scheduler is responsible for selecting and executing the next task, by activating the corresponding AEB of the task to be executed. The `next_aeb` reference of a task T_i is used to resume the execution of T_i by jumping to the region of code corresponding to the next AEB of T_i. At termination, every AEB updates the `next_aeb` of the currently running task to refer to the successor AEB according to the tasks's AEB graph.

The scheduling algorithm in Phantom is a priority-based scheme, as defined by POSIX. The way priorities are assigned to tasks, as they are created, can

enforce alternative scheduling schemes, such as round-robin, in the case of all tasks having equal priority, or earliest deadline first (EDF), in the case of tasks having priority equal to the inverse of their deadline. Additionally, priorities can also be changed at runtime, so that scheduling algorithms based on dynamic priorities can be implemented.

Phantom implements the basic semaphore (`sema_t` in POSIX) synchronization primitive, upon which any other synchronization construct can be built. To implement semaphores, there is a need to add to a task T_i's context an additional field called `status`. `Status` is one of *blocked* or *runnable* and is set appropriately when a task operates on a semaphore.

A semaphore operation, as well as a task creation and joining, is what is called a *synchronization point*. Synchronization points are identified by a gray node in Figure 2. At every synchronization point a modification in the state of at least one task in the system might happen. Either the current task is blocked, if a semaphore is not available, or a higher-priority task is released on a semaphore *signal*, for example. Therefore, a function is always phantomized when synchronization points are encountered, and a call to a synchronization function is always the last statement in its AEB. At this point, the scheduler must regain control and remove the current task from execution in case it became blocked or is preempted by a higher priority task.

3. CODE PARTITIONING

The Phantom partitioning is central to the correctness and the performance of the generated code [Nacul and Givargis 2004]. Boundaries of AEB represent the points where tasks might be preempted or resumed for execution. Every application has to be partitioned, so that context switching, synchronization, and scheduling are possible. Partitioning at synchronization points is mandatory, and is required to maintain correctness. Partitioning beyond synchronization points impacts the timing response of the code. In general, partitioning will determine the granularity level of the scheduling (i.e., the time quantum), as well as the task latency, event response time, and the multitasking overhead.

The multitasking overhead accounts for the time spent executing code that is not directly related to the original application. Instead, the code is executed to control task scheduling and iteractions. Typically, the multitasking overhead is due to runtime scheduling decisions, semaphores and mutexes checks, and interrupt handling. Ideally, one wants to minimize the multitasking overhead imposed on the application.

The response time can be characterized as the maximum amount of time between two scheduler activations. Response time is an important measurement in real-time applications. It estimates the maximum amount of time until an event, such as a communication from another task or an external input, is serviced in the system. In a system with cyclic tasks and different task priorities, the response time determines the wait time of the tasks until they are granted access to the processor and put into running state. In a system like Phantom, where tasks are preempted only at specific points in the code, a smaller response time has an impact on the overall system performance. The smaller the

```
 1 void task() {
 2
 3 int a, b;
 4
 5 a=10;
 6 b=0;
 7
 8 while(a>b) {
 9   b=rand();
10   print(b);
11 }
12 print(a);
13 ...
}
```

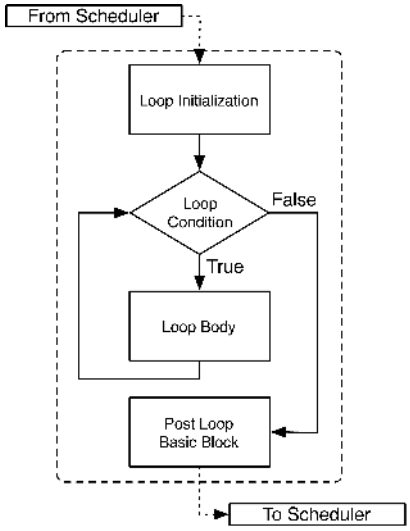
Fig. 3. Sample code segment.

AEBs, the more frequently the scheduler will be activated and events can be checked, therefore resulting in a smaller response time. However, every scheduler activation increases the total execution time of the multitasking code, as a result of the added overhead.

The timing behavior, and consequently the response time of the Phantom code, is determined by the partitioning process. On one end, there is the so-called *cooperative schedule*, where the code is partitioned only at the points mandatory for synthesizing a functionally correct application. On the other end, it is possible to generate a partition where every basic block is one AEB by itself, and every basic block transition is interlocked by a scheduler invocation. While this is the most responsive system possible, it carries a lot of overhead due to the large number of context switches.

In between these two extremes, there are lots of partitioning schemes. Each scheme partitions the application in different places, resulting in different timing behavior, AEB sizes, number of context switches, and so on. It is desirable to obtain the partition that meets the required constraints while, at the same time, minimizing the multitasking overhead imposed on the application. For AEBs with a straight sequence of code, that is, with no loops, this is not difficult to do. If an AEB a_i is too large, that is, if its execution time does not meet the timing constraints, it is always possible to partition a_i into a_{i1} and a_{i2}, therefore reducing the size of the original a_i. In such case, there is an increase by one in the number of context switches on every execution of a_i, which is acceptable to meet the timing constraints.

Nonetheless, partitioning AEBs which contain loops is more difficult. Meeting timing constraints in such cases demands more complex partitioning and code analysis. Assume the sample code segment for a task shown in Figure 3, which contains a loop (lines 8–10) that executes an undetermined number of times. Figure 4 shows one partitioning scheme possible with Phantom, where a loop is entirely contained within an AEB that is activated by the scheduler. A general diagram is shown in Figure 4(a), with dashed boxes illustrating the AEB borders, and dotted lines illustrating scheduler invocation and return.



(a) Partitioning diagram

```
1 aeb1:
2    a=10;
3    b=0;
4 loop:
5   if(a>b) {
6     b=rand();
7     print(b);
8     goto loop;
9   }
10  print(a);
11  next_aeb=2;
12  return;
13 aeb2:
   ...
```

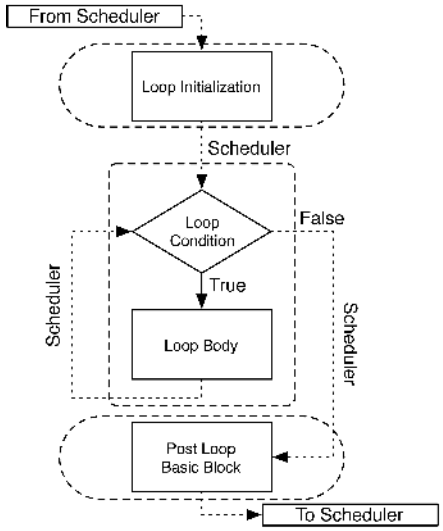
(b) Sample code segment

Fig. 4. Single AEB with loop.

Continuous boxes represent actual application code. Figure 4(b) shows the code of the AEB that is generated in this case by the Phantom compiler. The AEB `aeb1` is invoked by the scheduler and begins execution in lines 2 and 3, with the loop initialization code. The loop condition and the loop body are shown in lines 5 and 6–9, respectively. Note that line 8 effectively implements the loop body iteration. After the loop completes, it is followed by the postloop basic block of line 10 within the same AEB. Finally, control is returned to the scheduler in line 12, after saving the next AEB to be executed by this task in line 11. Note that lines 11 and 12 are additional code synthesized by the compiler to control multitasking.

The AEB shown in Figure 4 can execute for a long time, namely, until `b` is randomly assigned a value larger than `a`. While the AEB executes, all other tasks are waiting, as is the scheduler. Therefore, events cannot be checked, and timely execution of other tasks is not guaranteed. Nevertheless, the multitasking overhead is small, since the scheduler is activated only after the AEB (and consequently the loop) completes.

The available alternative is to place the loop body in an AEB by itself. In this case, the scheduler runs between every loop iteration, and loop repetition is effectively implemented by the scheduler. Such partition is depicted in Figure 5. Here, the loop initialization routine is in a separate AEB (`aeb1` in Figure 5(b)) than the loop condition and body (`aeb2`), and yet a separate AEB holds the postloop basic blocks (`aeb3`). Note that the loop condition (line 7 in Figure 5(b)) is checked on every loop iteration. Also note that, once the loop body (lines 8–9) is executed, the AEB returns to the scheduler (line 10) without modifying the next AEB of the current task. Therefore, when the task regains the processor, `aeb2` will be executed again, effectively performing the



(a) Partitioning diagram

```
 1 aeb1:
 2   a=10;
 3   b=0;
 4   next_aeb=2;
 5   return;
 6 aeb2:
 7   if(a>b) {
 8     b=rand();
 9     print(b);
10     return;
11   }
12   next_aeb=3;
13   return;
14 aeb3:
15   print(a);
16   next_aeb=4;
17   return;
18 aeb4:
   ...
```

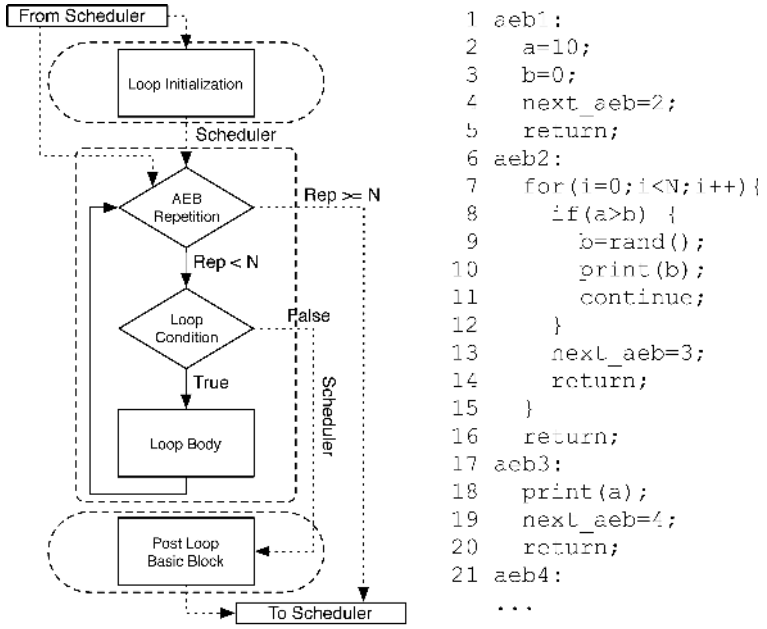
(b) Sample code segment.

Fig. 5. Loop in multiple AEBs.

loop repetition. If the loop condition fails, the next AEB is updated (line 12) and the loop body is no longer repeated. In this case, the AEB that contains the basic blocks following the loop (line 14) will be scheduled.

With these current partitioning schemes, either the loop can be entirely contained in the AEB, being executed completely in one scheduler activation (shown in Figure 4), or the loop body is partitioned into AEBs, which are activated by the scheduler as many times as necessary, depending on the loop iteration conditions (shown in Figure 5). In the latter case, the response time of the application is smaller, since the scheduler is invoked after every loop iteration. Possibly the time between scheduler invocations is significantly smaller than the timing constraints. However, the overhead imposed by such invocations is large, specially if the loop iterates many times.

It is possible that an AEB may contain a loop whose loop body is short and significantly smaller than the timing constraint. However, the execution of the complete loop without preemption can violate the constraints, or starve other tasks in case of infinite loops. Therefore, it is possible for the loop body to iterate N times before being preempted by the scheduler, while meeting the timing constraints and not increasing the multitasking overhead excessively. In order to accomplish such a solution, there is a need to modify the partitioning algorithm of Phantom [Nacul and Givargis 2004]. In the modified version, the loop body is enclosed within an external `for` loop, which repeats the loop body execution N times before preempting the loop and returning to the scheduler. Later the scheduler activates the task again, the loop body is resumed and allowed to execute another N times, if necessary. Figure 6 shows the new partition and scheduler iterations.



(a) Partitioning diagram (b) Sample code segment

Fig. 6. Preempting loops after N iterations.

Figure 6(b) shows the code synthesized by Phantom in the new partitioning approach. Note the external loop in lines 7–15 enclosing the original loop, lines 8–12. It is important to emphasize that the original loop condition is always tested in every loop iteration. Whenever the condition is false, the loop (and the AEB) terminates, returning control to the scheduler (lines 13–14) with the appropriate indication of the next AEB to be executed (line 13). Therefore, regardless of the value of N, the execution of the modified code will always be correct. Additionally, the proposed loop enclosing technique can be used in loops with nontrivial indexing, because the original loop condition is tested in every iteration.

Using the partitioning approach depicted in Figure 6, it is possible to control the execution time of an AEB more precisely, with a finer granularity. With such partition, there is a balance between multitasking overhead and timely execution of tasks. In order to implement it, one needs to determine the value of N, representing the number of consecutive loop iterations of the AEB before it is preempted. In the following section, we introduce our framework to analyze and generate code that meets timing constraints based on Phantom.

4. TIMING ANALYSIS

The synthesis of code that adheres to specified timing constraints, such as maximum response time of a task, requires an analysis of the application code, and in case of the Phantom compiler, appropriate partitioning. Because an AEB executes atomically, that is, there is no preemption during an AEB execution,

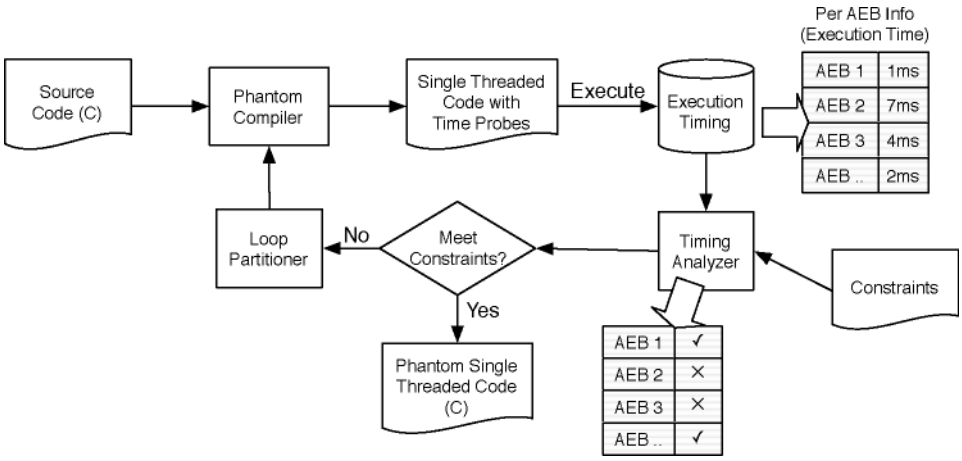


Fig. 7. Timing analysis framework.

reaching the right AEB size for all AEBs is crucial to obtain the desired timing behavior of an application. In this section, we present the timing analysis framework developed to analyze AEBs and generate the appropriate code partition for a given set of timing constraints.

Our partitioning exploration tool tries to reach a code partitioning that meets all timings constrains. For that, it needs estimates of the execution time of each AEB generated in the partition. Our exploration process gradually refines AEBs, starting from the cooperative scheduler, where partitions are larger, but context switching overhead is minimal. In each step, profiling is used to estimate the execution time of each AEB. For those AEBs that do not meet the constraints, the partitioning algorithm is run again, and a new set of AEB is generated from the original. Profiling is obtained for the new AEBs, and the process is repeated until an acceptable partition is obtained for each AEB. When all the AEBs meet the timing constrains, no more partitioning is necessary, and the final code is generated.

Our timing analysis framework is shown in Figure 7. The original C application, extended with POSIX, is compiled by Phantom and partitioned with the cooperative scheduling model, that is, only the partitions mandatory for correct multithreading. The code generation process of Phantom instruments the code with timing probes, which will generate profiling information for each AEB executed. The Phantomized code is executed and the generated profile is analyzed in the Timing Analyzer tool.

The Timing Analyzer checks for the constraints specified by the application designer, and outputs a list of the AEBs that do not meet the timing constraints. Each of these AEBs is processed by the Loop Partitioner, which searches for loops in the AEB and appropriately partitions the AEB into multiple AEBs with modified, and correct, new versions of the loop. The modified loops will have the structure discussed previously, as shown in Figure 6.

The new partition is processed again by the Phantom compiler, which synthesizes the corresponding C code for the new AEBs. The process is repeated

until all the AEBs meet the timing constraints. When all constraints are met, the Phantom compiler synthesizes the final version of the code, without the timing probes.

4.1 AEB Refinement

When refining the AEB partitioning, we focus on the loops contained in the AEB. Loop handling is the most complex structure to manipulate in partition. When refining, loops are restructured as discussed in Section 3.

The goal in the AEB partitioning refinement algorithm is to determine which loops should be preempted while being executed. Additionally, our algorithm also determines at which point during execution such preemption has to take place. In other words, the algorithm decides how many loop iterations can be performed before the AEB is preempted and execution returns to the scheduler, so that timing constraints such as response time are met. In order to do that, we will modify the structure of the AEB loops (as shown in Figure 6), so that the loop has points where preemption is possible. The loop body will iterate as many times as possible within an AEB. Therefore, timing constraints are met while multitasking overheads are kept to a minimum.

For AEBs that contain single, nonnested loops, the algorithm determines the value N of consecutive loop body iterations that can be performed without missing any timing constraint. N can be derived from the execution time of the loop body and the required response time. For AEBs with nested loops, our loop partitioning algorithm works from the outermost to the innermost loops in the AEB. Intuitively, this is the method that minimizes the increase in context switches and multitasking overhead. When an inner loop is preempted, all its enclosing outer loops are also preempted, resulting in a higher multitasking overhead. Therefore, we try to minimize the number of context switches, and consequently the multitasking overhead, by preempting as few loops as possible.

When multiple, independent loops are available in the same AEB, our algorithm selects the one with the longest running time to partition first. Intuitively, preempting the longest loops is more likely to reduce the response time quicker than partitioning the shorter loops.

After every modification in the task partitioning, the Phantom compiler synthesizes the instrumented code again. The new code is executed and the profiling information is checked against the constraints. This process repeats until the constraints are met. Once an AEB meets the required timing constraints, the preemption algorithm will analyze the partitioned loop bodies, determining the number of iterations N possible for each loop body before a scheduler invocation. Although this is a greedy algorithm, it is the most likely to generate the application with the minimum number of context switches while preempting the specified loops.

4.2 Estimation of AEB Execution Times

Estimating the execution time of an AEB in different partitions is fundamental to obtaining partitions that meet the required timing constraints. In our current framework, we use profiling to obtain execution time of each AEB. While

profiling is not the most accurate tool for hard real-time systems, it is a good approximation in soft real-time systems.

In case of hard real-time systems, our proposed partitioning and scheduling still applies. However, in this case profiling is not the most appropriate tool for estimating execution time of AEBs, since it contains an inherent imprecision. Alternatively, the designer can provide estimates for WCET of each AEB, using any method adequate to the application, replacing the profiling estimates. Our flow still applies, only using the provided WCET estimates. It is not the purpose of this work to provide tight bounded estimates for WCET. Various contributions have been made on WCET estimation, such as the works of Ermedahl et al. [2005] and Theiling et al. [2000]. Kirner and Puschner [2005] presented a study of different WCET analysis frameworks. If desired, WCET estimates from external tools can be supplied to the partitioning module of the Phantom compiler, which can perform partitioning based on these accurate estimates.

Nevertheless, the estimation of execution time of an AEB is likely to be simpler than the estimation of WCET of the whole application. One reason is that an AEB has much less code to be analyzed than the whole application. Moreover, the AEB code contains a well defined and simpler control flow, since there are restrictions to the structure of the code allowed inside an AEB. Finally, even if an unbounded loop is present (one of the most complex structures to handle in WCET estimation), our loop transformation techniques can manipulate the loop and transform it into a bounded loop that executes for a fixed number of times each time it is activated. While our loop transformation technique won't facilitate estimating WCET of the whole application, it absolutely helps in estimating other important real-time characteristics of an application, such as the maximum jitter, maximum response time, and other similar statistics.

5. RESULTS

We have implemented the described algorithm for loop partitioning and preemption. The timing analysis framework described on Figure 7 was also implemented, so that the exploration of different partitions can be done automatically for a given set of constraints. Additionally, we have implemented a tool that can explore all the possible loop partitions of an application, so that it is possible to compare the performance of our timing framework to the exhaustive approach of loop partitioning. All tools related to Phantom, as well as the benchmarks used in the experiments, are available in the Phantom Web site [Nacul and Givargis 2005b]

Eight application benchmarks were used to test the performance of our proposed framework for timing constraints. The applications were designed with POSIX threads, and included multithreaded versions of traditional algorithms such as *quick sort*, *consumer producer*, *matrix multiplication*, and *dct*. The benchmarks also included a *virtual machine* simulator, a *watch*, and *deep_stack*, a recursive application that exhausts the system stack. Finally, the benchmarks included a multithreaded software modem, which characterizes a real-time application. The modem is capable of V.21 and V.23 protocols, and therefore can transmit up to 1200 b/s. Table I summarizes the benchmarks.

Table I. Application Benchmarks

Name	Description	Response Time	Exec. Time	MT Overhead
cons_prod	Classical consumer producer problem, 100 consumers and 100 producers. Buffer with 1000 entries.	12.5 μs	4.50 s	50.2%
dct	Multitask implementation of 8×8 dct. One task for each point in the result matrix.	31.2 μs	0.78 s	13.5%
deep_stack	Multiple recursive tasks. Tests the cost of recursive function calls in the Phantom system.	9.3 μs	2.02 s	23.4%
matrix_mul	Multitask implementation of matrix multiplication. Resulting matrix is 150×150 elements. One task per element in the result.	6.25 μs	1.59 s	27.3%
modem	Multitask software modem implementation supporting DTMF, V.21, and V23 protocols.	6.25 μs	4.00 s	27.7%
quick_sort	Multitask implementation of the traditional sorting algorithm.	1.88 μs	0.04 s	2.3%
vm	Multitask simulator for a simple processor.	31.2 μs	27.6 s	8.2%
watch	Time-keeper application, used to test timing behavior of the generated code.	50 μs	67 s	1.6%

All experiments were conducted on a Celeron 1.6 GHz, with 256 Mb of RAM, running Linux kernel 2.6.11. For code instrumentation and execution time measurement, we used the *time stamp register* (TSR) built into the Pentium processor. The TSR is incremented at every clock cycle and is accessible via an assembly instruction. It provides an accurate measurement of the number of cycles executed by the processor. The code instrumentation and measurement process followed the recommendations from Intel [Intel Corp. 1997], which accounts for the pipeline, cache, and instruction reordering effects of the processor. To avoid interference from other system processes in the measurements, the Phantom code was executed in single-user mode, and its priority was set to the maximum process priority.

5.1 General Experiments

We performed a set of experiments to evaluate the general performance of serialized code generated by the Phantom compiler. The results are briefly presented on Table II. For a longer discussion on these results, please refer to our previously published work [Nacul and Givargis 2004].

Here, we compared the performance of serialized code when compared to the traditional OS-based approach using pthreads libraries, which are compliant to the POSIX standard. Both versions were executed on the same platform, running Linux kernel 2.6.11. On average, multitasking with Phantom achieved a speedup of 2.07, with a maximum of 2.8, when compared to traditional POSIX-based threads running on an OS (see Figure 8). These results come specially

Table II. Performance Results

Application	POSIX	Phantom
consumer_producer	7.23 s	3.54 s
dct	1.02 s	0.49 s
deep_stack	2.05 s	0.84 s
matrix_mul	1.10 s	0.55 s
quick_sort	2.97 s	1.12 s
vm	2.83 s	5.35 s
watch	67.01 s	67.00 s

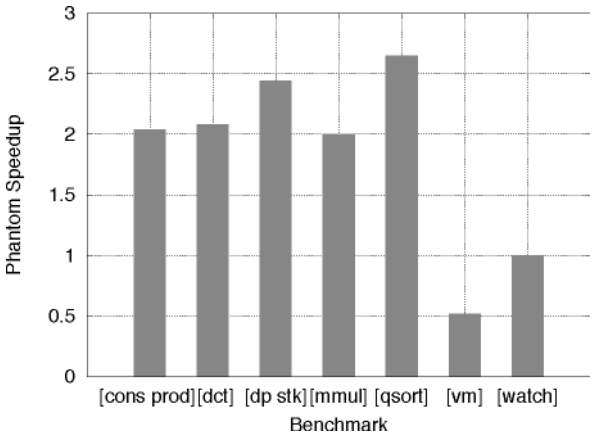


Fig. 8. Phantom speedup.

due to the lightweight implementation of Phantom [Nacul and Givargis 2005a], and as a consequence of being able to, at compile time, generate specific code for each different application.

Our previous experimental setups also evaluated the performance of the overhead equivalent to context switching in Phantom serialized code, as well as synchronization overheads, showing that serialization provides a very efficient multitasking abstraction [Nacul and Givargis 2005a].

5.2 Embedded Platform Experiments

We conducted a set of experiments in a real embedded platform. In particular, we used a Linksys WRT54G Wireless Router. The WRT54G contains a MIPS processor running at 216 MHz. It has 4 MB of FLASH memory for storing code and applications, and 16 MB of RAM, shared among all applications running on the hardware. The WRT54G firmware is an embedded version of Linux [BusyBox 2005]. Support for standard C functions, usually supplied by libC, were provided by μClib, a trimmed-down version of libC developed specially for embedded devices. Additionally, pthread libraries are available for the WRT54G, among other libraries, so that compiling and executing the applications using the traditional compile-execute paradigm with an operating system was possible.

We instrumented the WRT54G, and used external equipment to measure the execution time of each benchmark in the router. Using the high-frequency

Table III. Experimental Results

Application	Number of Threads	Context Switches	Execution Time (s)
client_server	5	2834	0.040
			0.122
consumer_producer	40	80201	0.286
			2.127
dct	512	2711	23.440
			22.150
deep_stack	100	15601	0.122
			0.122
matrix_mul	225	1158	0.286
			0.450
quick_sort	683	2392	0.021
			1.031
vm	500	2501	16.47
			14.38

DAS4020/12 PC sampling board [Measurement Computing, Inc. 2005] to measure data from the router, we were able to obtain execution time of the benchmarks running under the embedded OS and with Phantom. Specifically, we used a GPIO pin on the WRT54G to indicate start and completion of each individual benchmark. The DAS4020/12 board is a very fast, high-precision sampling board that can sample 24 digital input and output channels, four analog input signals, and two analog output signals at a maximum frequency of 20 MHz. We set the sampling rate to 50 kHz. Therefore, our measurements present a maximum timing jitter of 20 μs.

Due to the reduced memory in the WRT54G, we decreased the number of tasks in each of the benchmarks, while maintaining the main functionality. Table III summarizes the execution times of the application benchmarks, using Phantom and traditional pthread libraries. For each application, the top row presents results from the serialized version, while the bottom row shows the pthread-based approach. In most cases, the serialization of the application benchmarks with the Phantom compiler resulted in shorter execution times. On average, the serialized version of an application needed 30% less time to complete execution.

It can also be seen in Table III that serialization is not always the most effective solution. This is the case with *vm* and *dct*, where the execution performance of the serialized applications were worse than that of their traditional counterparts. In these examples, the iteration with the thread library was less than in other examples. Therefore, the advantages brought by serialization are amortized over the rest of the application. Since our prototype serializing compiler does not always generate optimal code, these two examples ended up taking longer to execute than the pthread-based one, mostly because of some deficiencies on the code generator stage of our prototype implementation.

5.3 Timing Analysis Experiments

For this section of experiments, the maximum response time was specified for every benchmark. Note that it is not our intention to show a direct relation

Table IV. Partitioning Exploration in Matrix Multiplication

Partition	# Ctx. Switches	Resp. Time	Exec. Time
1	112,803	398,000 μs	0.91 s
2	3,488,103	397,000 μs	4.22 s
3	473,403	2,600 μs	1.28 s
4	477,303	18.2 μs	1.29 s
5	3,832,053	5.3 μs	4.55 s
6	772,053	4.9 μs	1.59 s

between response time and multitasking overhead. Instead, our experiments intend to show that the Phantom compiler is successful in meeting certain timing constraints, and that our loop preemption scheme is able to reduce the multitasking overhead while maintaining a correct execution of the code.

The cooperative schedule is the partition that results in the minimal number of context switches, meaning there is least overhead imposed by the scheduler and multitasking control. Therefore, the cooperative scheduler is also the one that completes every benchmark the fastest. Meanwhile, the cooperative scheduler is also the least responsive system, due to its large partitions and consequently large AEB execution times.

Table I summarizes the performance of the benchmarks when synthesized with timing constraints, namely, a maximum response time. The constraints are input to the Phantom compiler, and are arbitrary in these examples. In the current prototype version of the Phantom compiler, timing constraints are specified as command-line options. Note that the response times used in the experiments were very small, significantly less than those traditionally guaranteed in standard operating systems.

Table I also shows the multitasking overhead of each application for these specific constraints. As it can be seen in Table I, the Phantom compiler can meet various timing constraints successfully. The multitasking overhead imposed on the system varies with different constraints, and is highly dependent on the application itself. Different synchronization points in different applications imply that some will have a larger overhead than others for the same timing constraint. This is in direct relation with the application code. For example, if there is a critical-section inside a loop that is executed a large number of times, that will result in a large number of context switches and high multitasking overhead, even if the critical section is very small compared to the timing constraints. Nevertheless, some additional partitioning might be necessary due to other parts of the application generating large AEBs in the initial, cooperative partition. This is the case with benchmark consumer-producer, and for this reason, the multitasking overhead is not always proportional to the timing constraint.

For the sake of discussion, we chose two of the most interesting benchmarks to present in detail. Table IV shows the result of our partition exploration framework for the *matrix multiplication* benchmark. This is a typical example of the effect of different partitions in application execution time and response time. In

Table V. Matrix Multiplication: Cooperative Scheduling

AEB	# Ctx. Switches	Resp. Time	Exec. Time
1	22,500	16.6 μs	374,000 μs
2	1	0.8 μs	0.8 μs
3	22,500	0.3 μs	7,400 μs
4	22,500	1.6 μs	36,700 μs
5	150	0.4 μs	62.2 μs
6	22,500	0.4 μs	9,300 μs
7	22,500	0.6 μs	13,700 μs
8	150	0.4 μs	58 μs
9	1	10,600 μs	10,600 μs
10	1	399,000 μs	399,000 μs
Total	112,803	399,000 μs	0.91 s

this example, a maximum response time of 6.25 μs, or 10,000 processor cycles, was specified as the timing constraint. Partition 1, the cooperative scheduling, was the fastest to complete, in 0.91 s. Table V shows the AEBs resulting from the cooperative scheduling. The reaction time was extremely high, about 400 ms, while the number of context switches was small, about 112,000. As our timing analysis and loop partitioning framework runs, large AEBs are partitioned into smaller, more responsive blocks. Loops are also restructured, so that it is possible to preempt an AEB while executing loops.

In the *matrix multiplication* benchmark, multiple large loops were partitioned to achieve partition 2. The number of context switches increased significantly; however, there was very little impact on the response time. This was due to the fact that the longest AEB contains nested loops, whiles were not partitioned in the first step. From partition 2 to partition 3, the nested loops were divided into smaller AEBs, while other loops that were partitioned in the previous partition were configured to iterate more than once inside the AEB. As a result, both response time and number of context switches were reduced. Consequently, execution time also decreased.

By applying our methodology, we were able to quickly achieve a partition that performed very close to the specified constraint. The resulting partition, shown in Table VI had a response time of 4.9 μs and executed in 1.58 s, performing about 770,000 context switches. Although execution time was relatively higher than the cooperative scheduler (73%), there was a penalty to pay when reducing the response time of the system by five orders of magnitude. The final partition was obtained by our algorithm in six iterations. Note in Table VI that AEBs 1, 9, and 10 were partitioned into smaller AEBs, some more than once.

The final partition on *matrix multiplication* was obtained by partitioning and allowing preemption of AEBs whose execution time was beyond the timing constraint. Additionally, instead of context switching after every loop iteration, we allowed smaller loops to iterate consecutive times before being preempted by the scheduler. A solution which had no preemption in the loops, that is, where every loop iteration was followed by a context switch, resulted in an execution

Table VI. Matrix Multiplication: Final Partition

AEB	# Ctx. Switches	Resp. Time	Exec. Time
1.1	22,500	0.40 μs	9,100 μs
1.2	315,000	1.60 μs	503,000 μs
2	1	1.11 μs	1.1 μs
3	22,500	0.33 μs	7,500 μs
4	22,500	1.74 μs	39,200 μs
5	150	0.50 μs	76.3 μs
6	22,500	0.41 μs	9,250 μs
7	22,500	0.62 μs	13,900 μs
8	150	0.39 μs	59.4 μs
9.1.1.1	1	4.93 μs	4.9 μs
9.1.1.2.1	2,250	2.87 μs	6,500 μs
9.1.1.2.2	150	0.39 μs	59.2 μs
9.1.2	2,250	2.90 μs	6,500 μ
9.2	150	0.40 μs	59.9 μs
10.1.1.1	1	1.43 μs	1.4 μs
10.1.1.2.1	315,000	1.73 μs	545,000 μs
10.1.1.2.2	22,500	0.50 μs	11,300 μs
10.1.2	150	0.41 μs	61.8 μs
10.2.1	1,650	1.07 μs	1,700 μs
10.2.2	150	0.38 μs	58.2 μs
Total	772,053	4.9 μs	1.58 s

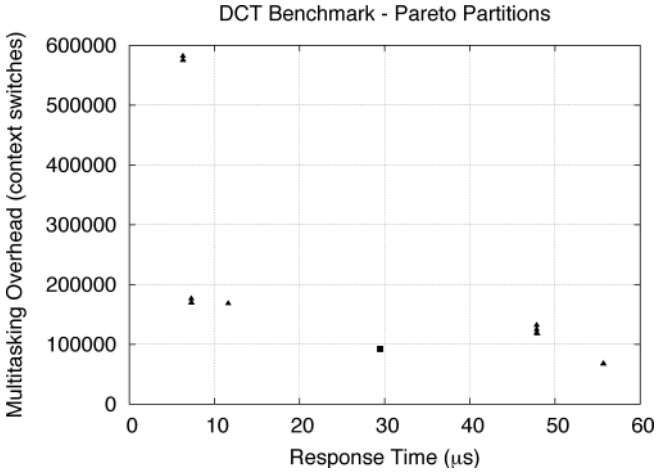


Fig. 9. DCT benchmark, pareto-optimal partitions.

time of 4.5 s (partition 5 in Table IV). It performed an order of magnitude more context switches and had approximately the same response time of the final partition obtained with this approach.

Figure 9 plots the response time and the multitasking overhead for the *dct* benchmark using the previous partitioning approach of Phantom. The triangular points on the plot are the Pareto-optimal partitions when loops are not allowed to preempt. The square point shows the solution achieved with loop preemption for the same set of constraints, which for this example was

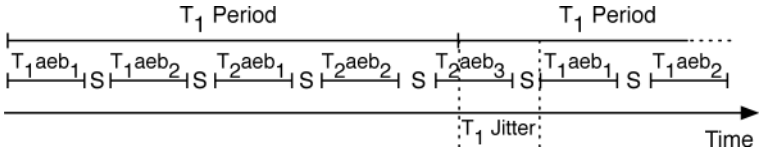


Fig. 10. Jitter sources in the phantom compiler.

set at 31.2 ms. Note that there is a large penalty in performance to reduce the response time below 50 ms. Using the timing analysis and new loop partitioning presented in this work, it is possible to reduce the response time without such a large impact in performance. In fact, our new partitioning can produce better response time with smaller multitasking overhead than the previous approach due to the better handling of loops in AEBs. Figure 9 shows the partition obtained with the new partitioning has a response time of 29.4 ms, below the constrain of 31.2 ms. If loops could not be preempted during execution, the best solution to this set of constraints would result in a response time as low as 12 ms, forcing almost twice as many context switches as the presented solution. The final execution time of this partition is only 5% worse than the cooperative scheduler, and 8% better than the previously possible partition. The proposed solution incurs a small overhead, nevertheless still providing a good performance and, most important, meets the timing constraint.

5.4 Real-Time Considerations

By using the Phantom Compiler to serialize the execution of multitasking code, it is possible to generate predictable code and provide guarantees regarding the timing behavior of the application. As an example, we included a modem implementation in our set of benchmarks. A software modem is a real-time application, in the sense that it must read data from and write data to the telephone system with a certain frequency, otherwise an error occurs in the transmission. In case of voice modems, which use the 4 kHz voice channel of the telephone bandwidth, they must sample the line at 8 kHz, or once every 125 μs, regardless of the transmission speed in bits per second (b/s).

The modem interface to the telephone line is composed of two tasks, the sender and the receiver tasks. Additionally, the software modem also includes support for DTMF as another task, so that dialing is possible. In the current software modem, the sender modulates the transmission with FSK, and both V.21 and V.23 protocols are implemented. The receiver demodulates the data, and even in the presence of noise, reconstructs the bit sequence.

While Phantom can provide timing guarantees for real-time applications like the software modem, tasks will still experience *jitter* during execution. In fact, eliminating jitter completely is a complex task in real-time systems. However, the maximum jitter of a set of Phantom tasks can be determined when the designer specifies a maximum response time to the Phantom Compiler. Figure 10 depicts the different components of the serialized Phantom code that influence the jitter in periodic task activations.

Consider two periodic tasks T_1 and T_2, such that T_1 has a higher execution priority than T_2. If T_1 has to be executed with a frequency f (8000 Hz in the case of the modem), the tasks have to be partitioned such that the activation frequency is met. T_2 executes whenever T_1 is not running, and the scheduler is invoked periodically to check on timers and verify the availability of other tasks.

When T_1's period expires, T_1 has to be scheduled on the processor. However, that won't happen until the scheduler runs, when the AEB being executed at that moment terminates. In the case where an AEB was recently scheduled for execution, T_1 will have to wait its completion before one of its own AEBs can be scheduled. Therefore, there will be a *jitter* in the scheduling of T_1 that, in the worst case, is equal to the execution time of the longest AEB in the system.

With the partitioning methodology presented in this work, the duration of the longest AEB is controllable at a fine level of granularity. Knowing the application constraints and the maximum acceptable jitter for each task, the designer can specify the timing parameters to the partitioner and guarantee that the timing specified will be met in the serialized code.

In the modem application, for example, the period of the sender and receiver tasks is 125 μs. Assuming a 5% jitter is acceptable, then the maximum execution time of an AEB is 6.25 μs. With this value as input, the Phantom compiler will partition the code such that no AEB executes for more than the constraint of 6.25 μs. As shown in Table I, this constraint was in fact given to the Phantom compiler for the serialization of the software modem. The generated code meets the constraint with a 27% overhead for scheduling and multitasking control.

6. RELATED WORK

Some of the features provided by the Phantom compiler are partially achieved by other approaches as well. In terms of code portability, Phantom generates a strict ANSI C code, which can be compiled with any standard compiler toolchain. The concept of Virtual Machines addresses this issue by providing an abstract machine that is simulated in every platform. Examples are Sun's Java [Gosling et al. 1996] and Microsoft's C# [Microsoft Corporation 2003]. The performance of the virtual machine is the main drawback of this approach, which also requires that the virtual machine is ported to the target platform. Some of the performance issues are addressed with JIT [Aycock 2003] and customized embedded virtual machines [Verdiere et al. 2002].

In the template-based RTOS generation techniques, a reference RTOS is used as a template in generating customized derivatives of the RTOS for particular embedded processor cores. This class of techniques mainly relies on inclusion or exclusion of RTOS features depending on application requirements and embedded processor core resource availabilities. The disadvantage of this class of techniques is that no single generic RTOS template can be used in the variety of embedded processor cores available. Instead, for optimal performance, a rather customized RTOS template must be made available for each line or family of embedded processor. In addition, for each specific embedded processor within a family, an architecture model must be provided to the generator engine.

In one example, Gerstlauer et al. [2003] have used the SpecC language, a system-level language, as an input to a refinement tool. The refinement tool partitions the SpecC input into application code and RTOS partitions. Each RTOS partition is subsequently refined to a final implementation. The mechanism used in this refinement is based on matching needed RTOS functionality against a library of RTOS functions. In a similar approach, Vercauteren et al. [1996] have proposed a method based on an API providing RTOS primitives to the application programmer. This RTOS template is used to realize the subset of the API that is actually used in the application program. Finally, Gauthier et al. [2001] have proposed an environment for RTOS generation similar to the previous approaches. Here a library of RTOS components that are parameterized is used to synthesize the target RTOS given a system level description of the application program.

System level modeling languages are also used to simulate the RTOS behavior in a larger design. The work presented by Moigne et al. [2004] used SystemC [Initiative 2005] to model the RTOS timing and functional behaviors. In their work, the authors did not present a synthesis of the RTOS, but only modeled the implementation of a generic RTOS based on templates for communication, synchronization, and scheduling. Nevertheless, the need for synthesis is clear if the simulated system is going to be synthesized as a prototype or final product.

The actual synthesis of embedded software from a system level design language was presented by Besana and Borgatti [2003]. Their work introduced a framework to estimate performance and synthesize application code against a predefined RTOS, namely, microC/OS [Labrosse 2002]. Although providing a complete flow, their proposal was restricted to specific system architectures and imposed a fixed RTOS choice. Nevertheless, it showed the importance of having the software and RTOS considered early in the design and specification process. A very similar framework was presented by Herrera et al. [2003], where SystemC was used to specify an embedded system and synthesize software and RTOS. Here, software was targeted at a specific RTOS, eCos [RedHat Inc 2005]. In this approach, SystemC primitives are replaced by the equivalent implementation in the eCos RTOS API.

Another related line of research presents the generation of statically scheduled code. In the static scheduling-based techniques, it is assumed that the application program consists of a static and a priori known set of tasks. Given this assumption, it is possible to compute a static execution schedule, in other words, an interleaved execution order, and generate an equivalent monolithic program. The advantage of this class of approaches is that the generated program is application-specific and thus highly efficient. The disadvantage of this class of techniques is that dynamic multitasking is not possible.

A very good general survey on generating sequential code for a static set of tasks was done by Edwards [2003]. In a more specific example, Lin [1998] has proposed a technique that takes as input an extended C code that includes primitives for intertask communication based on channels, as well as primitives for specifying tasks, and generates ANSI C code. The mechanism here is to

model the static set of tasks using a Petri net [Reisig 1992] and generate code simulating a correct execution order of the Petri net. Similar techniques have also been proposed by Cortadella et al. [2000]. One important aspect to note in both Lin's and Cortadella's approaches is that the generated code could still be multitasking, requiring the existence of an RTOS layer that can schedule and manage the generated tasks.

In a work by Cortadella et al. [2002], later extended by Hsiung et al. [2002], the concept of quasistatic scheduling was introduced. Both approaches are restricted to reactive systems only. In Cortadella's version of quasistatic scheduling, it is not possible to handle realtime constraints of embedded software. That restriction was partially addressed by Hsiung et al. [2002] with time-extended quasistatic scheduling. The application domain, however, was still for reactive embedded systems.

The generation of statically scheduled code is a strict requirement of synchronous languages like Esterel [Berry and Gonthier 1992]. Concurrency is built in the semantics of synchronous languages, and the need to execute it in sequential processors is the main challenge driving the compiler development, as the one proposed by Edwards [1999, 2002]. Synchronous languages, however, have a different execution model and sematics than traditional imperative languages like C. Furthermore, dynamic behavior is very restricted in these models, and there is no explicit primitive for thread creation.

In the application domain of control-dominated embedded systems, POLIS [Balarin et al. 1999] is another framework developed to synthesize embedded software. The solution implemented in POLIS is able to generate specialized code for the application, given that it is in the restricted domain of reactive systems. These are systems commonly found in industrial and automotive applications. The generated code is based on the concurrent finite state machine (CFSM) model developed by the authors. In POLIS, a custom scheduler is generated for the *static* set of tasks. Here, as with the other approaches presented, no dynamic behavior is possible.

In the topic of custom scheduler generation, the work of Polychronopoulos et al. [Schouten 1995] explored the concept of auto-scheduling, which is a dynamic scheduling technique for parallel tasks. Autoscheduling works on the nanoThreads concept. NanoThreads are used to extract parallelism on highly parallel descriptions. Scheduling is also mixed, as with quasistatic scheduling, where part of the scheduling is performed at compile time and part at runtime. With nanoThreads and auto-scheduling, the granularity of tasks can be controlled during the synthesis process [Moreira et al. 1995]. With autoscheduling, code transformations are performed to include runtime support for multitasking [Schouten 1995].

Karkowski and Corporaal [1998] presented an environment where software is synthesized from an ANSI C program. Their main interest, though, was in multiprocessor architectures, and the framework is capable of extracting functional parallelism from the application, both fine- and coarse-grain parallelism. Their work did not synthesize an RTOS. Instead, they were concerned with generating ASICs and statically scheduled code for the specific platform available in their setup.

The methodology presented by Cortés et al. [2004, 2005] proposed quasistatic scheduling for systems with soft and hard real-time tasks. They assumed a fixed, non-preemptive (collaborative) set of tasks, and compute multiple possible schedules offline. At runtime, as tasks complete, the system decides on schedule changes to take advantage of a possible slack and increase system utilization and efficiency. They proposed heuristics to handle single [Cortés et al. 2004] and multiprocessor [Cortés et al. 2005] architectures. However, their initial assumption was that all information about tasks are known ahead of time, and that tasks are static and nonpreemptive. In the Phantom compiler, tasks are preemptive, and we proposed an algorithm to efficiently partition the tasks according to the system's timing constraints. Furthermore, Phantom is able to dynamically create and schedule tasks.

An additional difference between Phantom and the work of Cortés et al. [2004, 2005] regards the representation of tasks and their interactions. While Cortés et al. used a task graph with data dependencies between tasks represented by the graph edges, Phantom is based on the AEB graphs, one for each task, where control dependencies are represented in the graph edges. In Phantom, data dependencies are implicit in the synchronization routines, which are the basis for the construction of AEB graphs. Therefore, Phantom is able to take both control and data dependencies between tasks (and parts of tasks) into consideration when constructing the final schedule.

The approach that is closest to the Phantom compiler was presented by Dean [2004]. In his work, he proposed software thread integration (STI), or integrating multiple threads in a single execution flow. STI prioritizes the primary task, considered real-time, and statically schedules the secondary tasks in the available idle cycles. Timing is only guaranteed for the primary task, while secondary tasks run in a best-effort scheduling. In Phantom, we allow global timing constraints, which affects all the tasks.

Goel et al. [2002] presented a technique to increase the time accuracy of off-the-shelf operating systems in order to meet the timing constraints of real-time applications. In their approach, they used Linux as the multitasking support. By modifying the Linux scheduler and adding timer checks on every context switch, in a concept called *soft timers*, they were able to reduce jitter to as little as 5 μs. Their solution was targeted at traditional operating systems, but could be adapted to an environment like the one provided by the Phantom compiler.

7. CONCLUSION

This article presented a solution to the synthesis of multitasking code with timing constraints by the Phantom compiler. Our approach analyzes the application partition generated by Phantom and modifies it in order to meet timing constraints while minimizing multitasking control overhead. We showed the use of our technique with different benchmarks and various constraints. With the Phantom compiler, one can synthesize applications that are able to meet a smaller response time when compared to traditional operating systems.

We are currently working to incorporate more real-time issues into the Phantom compiler. We are addressing issues like application deadlines, task

priorities, and interrupt management, among others. We are also looking into incorporating architectural details to the Phantom compiler flow so that more precise timing and different code transformations are possible.

REFERENCES

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Comput. Surv. 35*, 2 (June), 97–113.

BALARIN, F., CHIODO, M., GIUSTO, P., HSIEH, H., JURECSKA, A., LAVAGNO, L., SANGIOVANNI-VINCENTELLI, A., SENTOVICH, E., AND SUZUKI, K. 1999. Synthesis of software programs for embedded control applications. *IEEE Trans. Comput. Aid. Des. Integrat. Circ. Sys. 18*, 6 (June), 834–849.

BERRY, G. AND GONTHIER, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programm. 19*, 4, 87–152.

BESANA, M. AND BORGATTI, M. 2003. Application mapping to a hardware platform through automated code generation targeting a RTOS: A design case study. In *Proceedings of Design, Automation & Test in Europe (DATE)*.

BUSYBOX. 2005. Go online to http://www.busybox.net.

CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., PASSERONE, C., AND WATANABE, Y. 2002. Quasi-static scheduling of independent tasks for reactive systems. In *Proceedings of the 23rd International Conference on Application and Theory of Petri Nets (Adelaid, Australia, June 24–28)*. Lecture Notes in Computer Science, vol. 2360. Springer, Berlin, Germany.

CORTADELLA, J., KONDRATYEV, A., MASSOT, M., MORAL, S., PASSERONE, C., WATANABE, Y., AND SANGIOVANNI-VINCENTELLI, A. 2000. Task generation and compile-time scheduling for mixed data-control embedded software. In *Proceedings of Design Automation Conference* (DAC).

CORTÉS, L. A., ELES, P., AND PENG, Z. 2004. Quasi-static scheduling for real-time systems with hard and soft tasks. In *Proceedings of DATE*. 1176–1181.

CORTÉS, L. A., ELES, P., AND PENG, Z. 2005. Quasi-static scheduling for multiprocessor real-time systems with hard and soft tasks. In *Proceedings of the International Conference on Real-Time and Embedded Computing Systems and Applications*. 422–428.

DEAN, A. 2004. Efficient real-time fine-grained concurrency on low-cost microcontrollers. *IEEE Micro 24*, 4 (July-Aug.), 10–22.

EDWARDS, S. 1999. Compiling esterel into sequential code. In *Proceedings of CODES*.

EDWARDS, S. 2002. An esterel compiler for large control-dominated systems. *IEEE Trans. Comput.-Aid. Des. Integrat. Circ. Syst. 21*, 2 (Feb.), 169–183.

EDWARDS, S. 2003. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans. Des. Automat. Electron. Syst. 8*, 2 (Apr.), 141–187.

ERMEDAHL, A., STAPPERT, F., AND ENGBLOM, J. 2005. Clustered worst-case execution-time calculation. *IEEE Trans. Comput. 54*, 9 (Sep.), 1104–1122.

FISHER, J., FARABOSCHI, P., AND YOUNG, C. 2002. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kauffman, San Francisco, CA.

GAUTHIER, L., YOO, S., AND JERRAYA, A. 2001. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. Comput.-Aid. Des. Integrat. Circ. Syst. 20*, 11 (Nov.), 1293–1301.

GERSTLAUER, A., YU, H., AND GAJSKI, D. 2003. RTOS modeling for system level design. In *Proceedings of Design Automation & Test in Europe* (DATE).

GOEL, A., ABENI, L., KRASIC, C., SNOW, J., AND WALPOLE, J. 2002. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

HERRERA, F., POSADAS, H., SÁNCHEZ, P., AND VILLAR, E. 2003. Systematic embedded software generation from SystemC. In *Proceedings of Design, Automation & Test in Europe (DATE)*.

HSIUNG, P.-A., LEE, T.-Y., AND SU, F.-S. 2002. Formal synthesis and code generation of real-time embedded software using time-extended quasi-static scheduling. In *Proceedings of the Asia-Pacific Software Engineering Conference*.

INTEL CORP. 1997. Using the RDTSC instruction for performance monitoring. Intel Application Notes. Intel, Santa Clara, CA.

KARKOWSKI, I. AND CORPORAAL, H. 1998. Exploiting fine- and coarse-grain parallelism in embedded programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

KIRNER, R. AND PUSCHNER, P. 2005. Classification of WCET analysis techniques. In *Proceedings of the International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC).

LABROSSE, J. 2002. *MicroC/OS-II: The Real Time Kernel*. CMP Books, San Francisco, CA.

LIN, B. 1998. Efficient compilation of process-based concurrent programs without run-time scheduling. In *Proceedings of DATE*.

MEASUREMENT COMPUTING, INC. 2005. PCI-DAS4020/12 Specifications, Rev 1.2. Available online at `http://www.mccdaq.com`.

MICROSOFT CORPORATION. 2003. The C# 2.0 Specification. Available online at `http://msdn.microsoft.com/vcsharp`.

MOIGNE, R. L., PASQUIER, O., AND CALVEZ, J.-P. 2004. A generic RTOS model for real-time systems simulation with SystemC. In *Proceedings of Design, Automation & Test in Europe* (DATE).

MOREIRA, J., SCHOUTEN, D., AND POLYCHRONOPOULOS, C. 1995. The performance impact of granularity control and functional parallelism. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*.

NACUL, A. AND GIVARGIS, T. 2004. Code partitioning for synthesis of embedded applications with Phantom. In *Proceedings of ICCAD*. 190–196.

NACUL, A. AND GIVARGIS, T. 2005a. Lightweight multitasking support for embedded systems using the Phantom serializing compiler. In *Proceedings of DATE*. 742–747.

NACUL, A. AND GIVARGIS, T. 2005b. Phantom compiler 0.8. Go online to `http://www.ics.uci.edu/~nacul/phantom`.

REDHAT INC. 2005. Embedded configurable operating system (ecos). Go online to `sources.redhat.com/ecos`.

REISIG, W. 1992. *A Primer in Petri Net Design*. Springer-Verlag, Berlin, Germany.

SCHOUTEN, D. 1995. Efficient scheduling of parallel tasks in a multiprogramming environment. Ph.D. dissertation. University of Illinois at Urbana-Champaign.

THE OPEN SYSTEMC INITIATIVE. 2005. Go online to `www.systemc.org`.

THEILING, H., FERDINAND, C., AND WILHELM, R. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Int. J. Time-Crit. Comput. Syst. 18*, 2 (May), 157–179.

VERCAUTEREN, S., LIN, B., AND MAN, H. D. 1996. A strategy for real-time kernel support in application-specific HW/SW embedded architectures. In *Proceedings of the Design Automation Conference* (DAC).

VERDIERE, V., CROS, S., FABRE, C., GUIDER, R., AND YOVINE, S. 2002. Speedup prediction for selective compilation of embedded Java programs. In *Proceedings of EMSOFT*.

Received February 2006; revised June 2006; accepted July 2006