Proceedings of the 2006 American Control Conference
Minneapolis, Minnesota, USA, June 14-16, 2006

Phantom: A Serializing Compiler for Multitasking Embedded Software

André C. Nácul and Tony Givargis
Center for Embedded Computer Systems – Department of Computer Science
University of California, Irvine – Irvine, CA 92697
{nacul, givargis}@uci.edu

***Abstract*— In an era of powerful general-purpose embedded compute platforms, the migration of system functionality from application specific integrated circuits (ASICs) to software has become a promising trend toward addressing the system complexity and shrinking time-to-market window challenges. Hence, in modern embedded systems, software development plays an increasingly vital role. On the other hand, the real-time concurrent programming model provides the high level abstractions necessary to effectively design complex software. Support for real-time concurrent programming is typically provided by a real-time operating system. We propose an alternate solution, the Phantom serializing compiler, to support the real-time concurrent programming model. The Phantom serializing compiler generates a single-threaded monolithic executable, from the multi-threaded application software, capable of directly executing on the underlying embedded compute platform. Further, the generated executable is tuned for maximum performance and efficiency, yielding an application-specific solution. In this paper, we give an overview of the Phantom serializing compiler and demonstrate its feasibility with some experimental results.**

I. Introduction

Design and development of modern embedded systems pose a number of unique engineering challenges. The shelf-time of consumer embedded devices is rapidly shrinking, forcing designers to develop products within a very short time frame. On the other hand, as a requirement for success, designers of embedded systems are increasingly forced to use innovation in product design to gain a competitive edge in the market [1]. Clearly, the complexity of embedded devices is rising at a rapid pace. The increased complexity of embedded devices is in part due to an increased user demand for features and functions, the need to support a large number of emerging communication protocols, the requirement to provide multimedia support, the necessity to address emerging security/privacy issues, and the general desire to implement innovative features and functions.

The hardware platform of today's embedded systems is frequently amortized over multiple products, intended to be highly programmable, and made reliant on one or more *embedded processor core(s)*. In recent years, a plethora of highly capable embedded processor cores have become available. Companies such as Tensilica [2], ARM [3], and Intel [4] each provide large families of highly tuned embedded processor cores. These embedded processor cores are shipped with optimizing compilers that support the sequential programming model of computation.

In order to cope with the complexity of systems, engineers are turning to software. In part, well established programming models, highly evolved tool chains, ease of software reuse, availability of open-source software, and expeditious nature of the *design-compile-execute* paradigm make developing a new feature in software more favorable [5]. It is assumed that, in coming years, more than 70% of product features are going to be supported by software [6].

Embedded software is characterized by a set of concurrent, deadline-driven, synchronized, and communicating tasks [7]. Hence, embedded software is best captured using the *real-time concurrent programming model*. The support for real-time concurrent programming is usually provided by a real-time operating system (RTOS). In general, an RTOS is a generic framework which can be used across a large number of processors and applications. An RTOS provides coarse grained timing support, and is loosely coupled to the running tasks. As a result, an RTOS is seldom optimized for any particular application. Additionally, the overhead of an RTOS prohibits its use in applications where the hardware platform is based on low-end microcontrollers.

In this work, we propose an alternative to an RTOS based on the idea of *serializing compilers*. A serializing compiler is an automated software synthesis methodology that can transform a multitasking application into an equivalent and optimized monolithic sequential code, to be compiled with the embedded processor's native optimizing compiler, effectively replacing the RTOS. The serializing compiler can analyze the tasks at compile time and generate a fine-tuned, application specific infrastructure to support multitasking, resulting in a more efficient executable than one that is intended to run on top of a generic RTOS. By having control over the application execution and context switches, the serializing compiler enables the fine grain control of task timing while enhancing overall performance.

Traditional compilers are efficient in generating code for sequential applications. They can use platform specific resources, generate an optimized instructions, and explore pipelines and memory hierarchies. More recently, with VLIW and simultaneous multithreading (SMT) architectures, compilers have started to support the generation of concurrent code [8] [9]. However, compilers have a limited understanding of tasks, and completely lack support for timing constraints of real-time applications. Our serializing compiler technology strengthens existing compilers, making them timing and task-aware.

The Phantom Compiler [10] [11], our implementation of a serializing compiler, provides a fully automated mechanism to synthesize a single threaded, ANSI C/C++ program from a multithreaded C/C++ (extended with POSIX [12]) program. The Phantom generated code is highly tuned for the input application.

This paper is organized as follows. Section II discusses related approaches to the serializing compiler. Section III presents the Phantom Compiler, while Section IV introduces our solution to support timing constraints. Experimental results are presented in Section V. We conclude the paper in Section VI.

II. 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. The performance of the virtual machine is the main drawback of this approach, which also

1-4244-0210-7/06/$20.00 ©2006 IEEE

requires that the virtual machine is ported to the target platform. Some of the performance issues are addressed with JIT [13] and customized embedded virtual machines [14].

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. On template-based code generation, some approaches generate a simulated RTOS from a system-level language description as input [15] [16] [17] [18]. Other works actually synthesize a customized RTOS code [19].

Another related line of research presents the generation of statically scheduled code [20] [21] [22]. A good survey on the topic is presented by Edwards [23]. 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 is that dynamic multitasking is not possible.

In the application domain of control-dominated embedded systems, POLIS [24] is another framework developed to synthesize embedded software, but restricted to reactive systems. Polychronopoulos et al. [25] explores the concept of auto-scheduling, which is a dynamic scheduling technique for parallel tasks. In auto-scheduling, task granularity can be controlled during the synthesis process [26].

The approach that is closest to the Phantom compiler is presented by Dean [27]. In his work, he proposes Software Thread Integration (STI), 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.

III. 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 [11] [10].

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 [12]. These 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 is 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

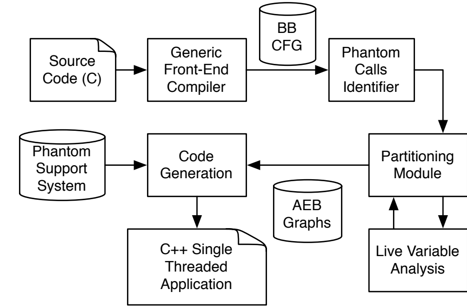


Fig. 1. Phantom Compiler Architecture

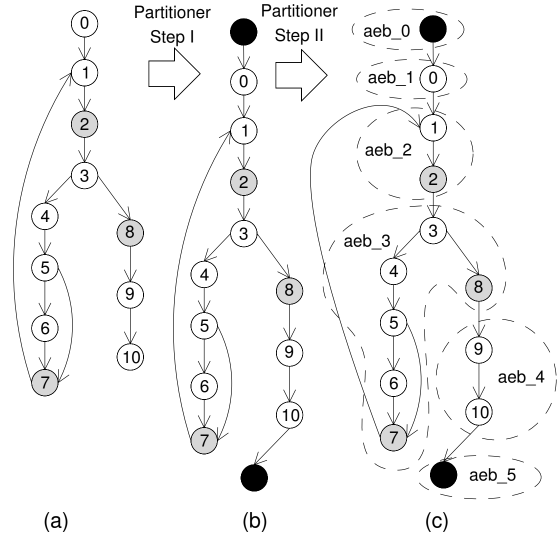


Fig. 2. Example of CFG Transformations

to generate non-preemptive 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 AEBs, generating an AEB Graph. The timing partitioning module analyzes the AEBs for timing constraints, and refines the partitions of the AEBs that violate any constraint. The resulting AEB graphs are then passed to the code generator to output the corresponding ANSI C++ code for each AEB node. In addition, the embedded scheduler, along with other multitasking data structures and synchronization APIs are included from the *Phantom* system support library, resulting in the final ANSI C++ single-threaded code.

A. 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. Figure 2(a) shows the CFG output by the compiler front-end, annotated with the Phantom primitives. The partitioner adds two control basic blocks, *setup* and *cleanup* (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*.

The termination of an AEB transfers the control back to the scheduler. The scheduler, then, activates the next AEB, from either the same task or from another task ready to run. The scheduling algorithm in *Phantom* is priority based. The way priorities are assigned to tasks,

as they are created, can enforce alternate scheduling schemes, such as round-robin or earliest deadline first (EDF). Additionally, priorities can also be changed at run-time, so that scheduling algorithms based on dynamic priorities can be implemented.

Every function f that is phantomized generates AEBs that are implemented as a separate region of code. Therefore, Phantom needs to provide a mechanism to store f's live variables at the transition between AEBs, when a context switch happens. Also, every task must maintain its own copy of local variables during the execution of f on its context. *Phantom* solves this issue by creating an object for each phantomized function, whose attributes are the local variables of f, emulating the concept of a *function frame*. The frame of a function f is created in the f_{setup} block, and cleaned up in the last AEB of f. 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 to be executed when the task regains the processor, called `next_aeb`, in the context information for each task that has been created. When a task is created, the context object 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.

Phantom implements the semaphore synchronization primitive, upon which other synchronization constructs 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 either *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 a synchronization point (gray nodes in Figure 2). At every synchronization point, a modification in the state of at least one task in the system can 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.

IV. Code Partitioning

Partitioning is central to the correctness and the performance of the generated code [11]. Boundaries of AEB represent the points where task preemption occurs. Every application is 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 of scheduling (i.e., the time quantum), as well as the task latency, 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 interactions. Typically, the multitasking overhead is due to run-time scheduling decisions, semaphores and mutexes checks, and interrupt handling. Ideally, one wants to minimize the multitasking overhead of the application.

An important timing characteristic in real-time applications is the *response time*. It can be defined as the maximum amount of time between two scheduler activations, and is used to estimate the maximum period of time until an event is serviced in the system. With Phantom, where tasks are preempted only at specific points in the code, a smaller response time has an impact on the overall system performance. Smaller response times mean more frequent scheduler activations and event checking in the system, and consequently require smaller AEBs to be generated. However, every scheduler activation increases the total execution time of the multitasking code, as a result of the added overhead.

```
 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

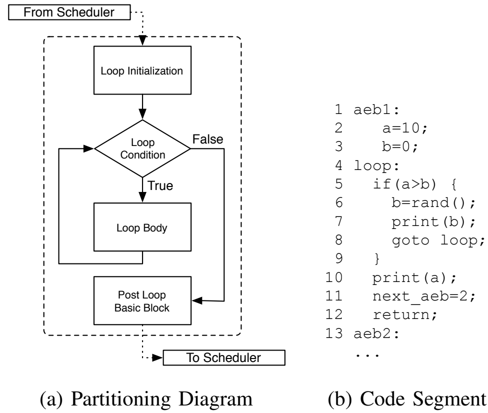
The timing behavior, and consequently the responsiveness of the Phantom code, is determined by the partitioning process, since tasks can only be preempted at the border of AEBs. On one end, there is the so-called *cooperative schedule*, where the code is partitioned only at the points mandatory for synthesizing 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.

Between these extremes, there are lots of partitioning schemes. Different partitions result 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 minimizing the multitasking overhead of the application.

For AEBs with straight sequence of code, i.e., no loops, this is not difficult to do. If an AEB a_i is too large, i.e., does not meet the timing constraints, it is always possible to partition a_i into a_{i1} and a_{i2}, reducing the size of the original a_i. Here, 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.

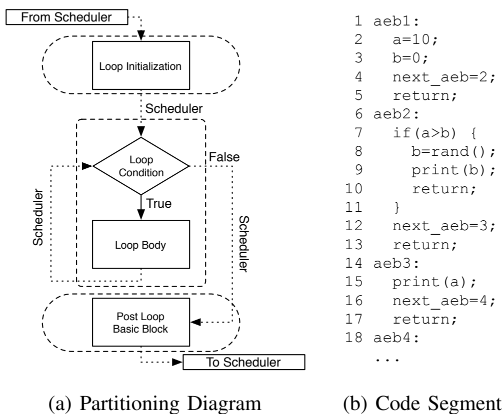
Partitioning an AEB with loops, however, is not as trivial. Particularly, unbounded loops are critical, as the one shown in Figure 3, lines 8-10. In general, three partitioning schemes are possible. Firstly, the loop can be entirely contained inside the AEB, including loop body and control into the same AEB, shown in Figure 4. Alternatively, the partitioner can separate the loop back-edge to be (logically) executed by the scheduler, forcing a context switch at every loop iteration, as illustrated in Figure 5. Finally, the partitioner is able to organize the loop body and back-edge into the same AEB, adding extra control instructions to allow a context switch during the loop execution (Figure 6).

AEBs that contain unbounded loops, as in Figure 4, can execute for a long time. 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



(a) Partitioning Diagram (b) Code Segment

Fig. 4. Loop inside AEB



(a) Partitioning Diagram (b) Code Segment

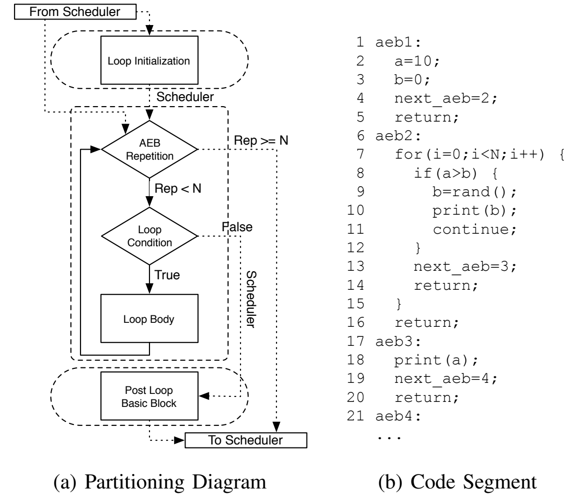
Fig. 5. Loop in Multiple AEBs

only after the AEB (and consequently the loop) completes.

The separation of the loop back-edge, enforcing a context switch at every iteration, reduces the response time of the application. 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 8-10) does not modify the `next_aeb` of the current task. Therefore, when the task regains the processor, `aeb2` will be executed again, effectively traversing the loop back-edge via the scheduler. Here, the time between scheduler invocations is likely smaller than the timing constraints. However, the overhead is large, specially if the loop iterates many times, each of them executing a short loop body.

Therefore, it is possible for the loop body to iterate N times before being preempted. It can meet the timing constraints, not increasing the multitasking overhead excessively. This is the solution pictured in Figure 6. The loop body is enclosed within an external `for` loop, which repeats the AEB execution N times before 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.

Using the 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 before the AEB is preempted. The execution time of a loop body can vary between iterations, and it is not usually possible to have all iterations to execute for exactly the same time. The value



(a) Partitioning Diagram (b) Code Segment

Fig. 6. Preempting Loops After N Iterations

of N can be computed from an average execution time of the loop body, in case of soft time constraints. If worst case timing guarantees are necessary, such as in a hard real-time system, the value of N is determined by the Worst Case Execution Time of the loop body.

A. Timing Analysis Framework

The synthesis of code that adheres to specified timing constraints requires an analysis of the application code and appropriate partitioning. As there is no preemption during an AEB, reaching the right AEB size for all AEBs is key 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 given a set of timing constraints.

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. Phantom instruments the generated code with timing probes, to obtain profiling information for each AEB. 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 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.

When searching an AEB for loops to be partitioned , the algorithm will select the outermost loop of an AEB in case there are nested loops within one AEB . Otherwise, if an inner loop is selected, all the enclosing outer loops will be partitioned, in addition to the selected loop. Therefore, multitasking overhead will be excessively increased. If the new partition still does not meet the constraints, the next nesting level will be analyzed, an so the algorithm works inwards in the nested loop structure.

Note that since we rely on profiling information for partitioning, the approach is not applicable to hard real-time applications. However, it is possible to replace the profiling method for a static analysis

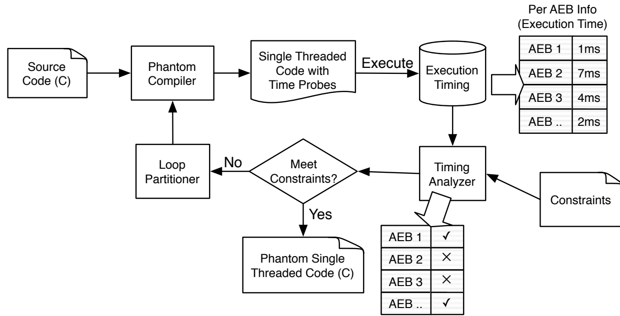
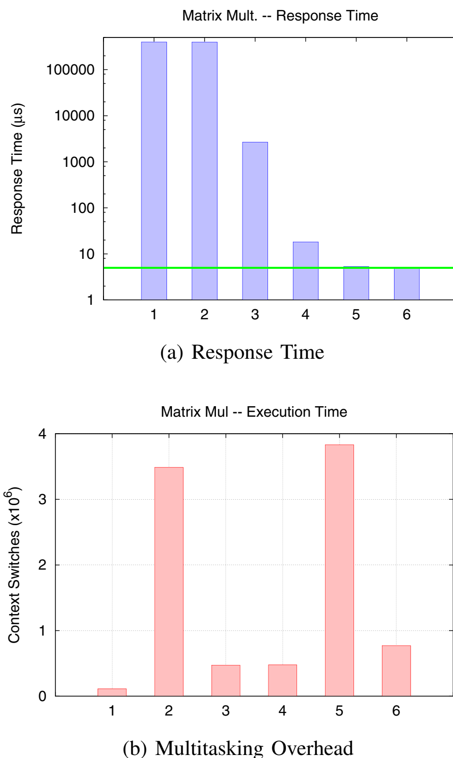


Fig. 7. Timing Analysis Framework



(a) Response Time

(b) Multitasking Overhead

Fig. 8. Matrix Multiplication Timing

considering the WCET of each basic block, which would provide much stricter guarantees on the execution of the synthesized code.

V. Results

Eight application benchmarks were implemented to test the overall performance of the code synthesized with the *Phantom* compiler, as well as its capability of meeting timing constraints. The applications were designed with POSIX threads, and include multithreaded versions of traditional algorithms such as *quick sort*, *consumer producer*, *matrix multiplication*, and *dct*. The benchmarks also include a *virtual machine* simulator, a *watch*, and *deep_stack*, a recursive application that exhausts the system stack. Table I summarizes the benchmarks.

We have previously reported on the overall performance of *Phantom* code when compared to traditional OS-supported multitasking [11]. In summary, *Phantom* outperforms standard Linux/POSIX implementations, being 2 to 3 times faster in execution time. On the average, multitasking with *Phantom* achieves a speed-up of 2.3, with a maximum of 2.9. In general, multitasking applications synthesized with *Phantom* show a much improved performance (i.e., low operating overhead). The reason is two fold. First, the generated application encompass a highly tuned multitasking framework that meets the application-specific needs. Second, the multitasking infrastructure itself is very compact and efficient, resulting in a much lighter overhead for context switching, task creation, and synchronization [10].

We have also experimented with the timing analysis framework presented in this work. Different response time were specified for each benchmark. Table I summarizes the performance of the benchmarks when synthesized with the timing constraints. Note that the response times specified are very small, significantly less than traditionally supported in standard operating systems. Table I shows the multitasking overhead of each application for these specific constraints. The multitasking overhead imposed on the system varies with different constraints, and is highly dependent on the application.

Figure 8 shows the iterations of the timing analysis framework with the matrix multiplication algorithm. A maximum response time of 6.25μs, was specified as the timing constraint. Figure 8(a) illustrates the reduction in the maximum response time of the application. After 6 iterations, the timing constraint was met (straight line in Figure 8(a)). Figure 8(b) shows the variation in the execution time associated to each partition generated. Partition 1, the cooperative scheduling, is the fastest to complete execution (0.91s), but results in a very high response time (400ms). As our timing analysis tool executes, large AEBs are divided into smaller ones. Loops are also restructured, so that it is possible to preempt an AEB while executing loops.

The final partition was achieved after 6 iterations. The resulting partition has a response time of 4.9μs and executes in 1.58s. Although the final execution time was 73% larger than the cooperative scheduler, the response time was reduced by 5 orders of magnitude.

Our benchmark applications also included a software modem example. A software modem is a real-time application, since 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 4KHz voice channel on regular telephone systems, they must sample the line at 8Khz, resulting in a period of 125μs. In order to allow a 5% variation in the period, the response time for the modem was set to 6.25μs. Phantom was able to synthesize the corresponding code, which included a 27% overhead.

VI. Conclusion

We provided an overview of the *Phantom* serializing compiler, which allows the generation of single threaded code from POSIX-based multitasking applications. The synthesized code tightly couples the tasks, and is highly customized for the input application, allowing for a more efficient multitasking execution when compared to the traditional RTOS-based approach. Additionally, the generated code is platform-independent and can be compiled to any target platform, enabling multitasking even for low-end microcontrollers.

This work introduced a framework to support real-time constraints in the code synthesis process. In the current version, we allowed tasks to specify the maximum response time expected from the system, and we were able to effectively synthesize code that meets such constraints. Our experiments show that a very fine granularity is possible in the final application, in opposition to the usually coarse grained concurrency supported in traditional approaches.

There is still a number of opportunities for improvement and further development in the Phantom serializing compiler. Our immediate plans include extending the real-time support in the system, with the incorporation of periodic tasks, deadlines, and different scheduling

TABLE I
APPLICATION BENCHMARKS

Name	Description	Response Time (input)	Execution Time	Multitasking Overhead
consumer_producer	Classical consumer producer problem, 100 consumers and 100 producers. Buffer with 1000 entries.	12.5 μs	4.50s	50.2%
dct	Multitask implementation of 8x8 dct. One task for each point in the result matrix.	31.2μs	0.78s	13.5%
deep_stack	Multiple recursive tasks. Tests the cost of recursive function calls in the *Phantom* system.	9.3 μs	2.02s	23.4%
matrix_mul	Multitask implementation of matrix multiplication. Resulting matrix is 150x150 elements. One task per element in the result.	6.25μs	1.59s	27.3%
quick_sort	Multitask implementation of the traditional sorting algorithm.	1.88μs	43.9ms	2.3%
vm	Multitask simulator for a simple processor.	31.2μs	27.6s	8.2%
watch	Time-keeper application, used to test timing behavior of the generated code.	50μs	67s	1.6%

algorithms in addition to the current priority-based scheduling. Additionally, we plan on generating code with different optimization objectives in mind, such as low energy and small memory footprint.

Further into the *Phantom* compiler development, we intend to investigate the possibility of generating architecture-specific code, by integrating Architecture Description Languages into the set of inputs used by the compiler. We believe it is possible to optimize some parts of the code generation process if we target it to one determined architecture. We also intend to investigate the issues of determinism and predictability of the synthesized code. There are multiple levels to determinism, from guaranteeing deadline compliance to ensuring the exact sequence of instructions executed. We believe providing some level of determinism to the execution of the code would be valuable, specially in hard real-time systems.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation award number CCR-0205712 and by CAPES Foundation, Brazil, award number 1054/01-5.

REFERENCES

[1] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley and Son, 2001.
[2] Tensilica Inc., http://www.tensilica.com.
[3] ARM Inc., http://www.arm.com.
[4] Intel Inc., www.intel.com.
[5] E. Lee, “What’s Ahead for Embedded Software?” *IEEE Computer*, September 2000.
[6] The Boston Consulting Group, “The Growing Importance of Embedded Software,” www.bcg.com.
[7] W. Wolf, *Computers as Components: Principles of Embedded Computing Systems*. Kaufmann, 2001.
[8] L. Chen and Y. Wu, “Aggressive Compiler Optimization and Parallelization with Thread-Level Speculation,” in *International Conference on Parallel Processing*, Jun. 2003.
[9] G. Dimitriou and C. Polychronopoulos, “Loop Scheduling for Multithreaded Processors,” in *International Conference on Parallel Computing in Electrical Engineering*, Sept. 2004.
[10] A. Nacul and T. Givargis, “Lightweight Multitasking Support for Embedded Systems using the Phantom Serializing Compiler,” in *Proc. of DATE*, Feb. 2005.
[11] ——, “Code Partitioning for Synthesis of Embedded Applications with Phantom,” in *Proc. of ICCAD*, Nov. 2004.
[12] POSIX Open Group, http://www.opengroup.org.
[13] J. Aycock, “A Brief History of Just-In-Time,” *ACM Computing Surveys*, vol. 35, no. 2, Jun. 2003.
[14] V. Verdiere, S. Cros, C. Fabre, R. Guider, and S. Yovine, “Speedup Prediction for Selective Compilation of Embedded Java Programs,” in *Proc. of EMSOFT*, Oct. 2002.
[15] A. Gerstlauer, H. Yu, and D. Gajski, “RTOS Modeling for System Level Design,” in *Proc. of DATE*, Mar. 2003.
[16] S. Vercauteren, B. Lin, and H. D. Man, “A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures,” in *Proc. of DAC*, Jun. 1996.
[17] L. Gauthier, S. Yoo, and A. Jerraya, “Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software,” *IEEE Trans. on CAD*, vol. 20, no. 11, Nov. 2001.
[18] R. L. Moigne, O. Pasquier, and J.-P. Calvez, “A Generic RTOS Model for Real-Time Systems Simulation with SystemC,” in *Proc. of DATE*, Feb. 2004.
[19] F. Herrera, H. Posadas, P. Snchez, and E. Villar, “Systematic Embedded Software Generation from SystemC,” in *Proc. of DATE*, Mar. 2003.
[20] B. Lin, “Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling,” in *Proc. of DATE*, Feb. 1998.
[21] J. Cortadella, A. Kondratyev, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli, “Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software,” in *Proc. of DAC*, Jun. 2000.
[22] P.-A. Hsiung, T.-Y. Lee, and F.-S. Su, “Formal Synthesis and Code Generation of Real-Time Embedded Software using Time-Extended Quasi-Static Scheduling,” in *Proc. of Asia-Pacific Software Engineering Conference*, 2002.
[23] S. Edwards, “Tutorial: Compiling Concurrent Languages for Sequential Processors,” *ACM Trans. on Design Automation of Electronic Systems*, vol. 8, no. 2, Apr. 2003.
[24] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki, “Synthesis of Software Programs for Embedded Control Applications,” *IEEE Trans. on CAD*, vol. 18, no. 6, Jun. 1999.
[25] E. Polychronopoulos and T. Papatheodorou, “Scheduling User-Level Threads on Distributed Shared-Memory Multiprocessors,” in *EuroPar Conference*, 1999.
[26] J. Moreira, D. Schouten, and C. Polychronopoulos, “The Performance Impact of Granularity Control and Functional Parallelism,” in *Proc. of Workshop on Languages and Compilers for Parallel Computing*, 1995.
[27] A. Dean, “Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers,” *IEEE Micro*, vol. 24, no. 4, Jul-Aug 2004.