Lightweight Multitasking Support for Embedded Systems using the Phantom Serializing Compiler

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

ABSTRACT

Embedded software continues to play an ever increasing role in the design of complex embedded applications. In part, the elevated level of abstraction provided by a high-level programming paradigm immensely facilitates a short design cycle, fewer errors, portability, and reuse. Serializing compilers have been proposed as an alternative to traditional OS techniques, enabling a designer to develop multitasking applications without the need of OS support. In this work, we outline the inner workings of the Phantom serializing compiler and analyze the quality of the generated code with respect to memory and processing overheads. Our results show that such serializing compilers are extremely efficient, making them ideal to be used in design of highly parallel applications (e.g., multimedia, graphics, and signal processing applications).

1. INTRODUCTION

The functional complexity of embedded software continues to rise due to a number of factors such as sophisticated user interfaces, seamless operation across multiple communication protocols, need for security, and so on. Consequently, the development of embedded software poses a major design challenge. At the same time, the elevated level of abstraction provided by a high-level programming paradigm immensely facilitates a short design cycle, fewer errors, portability, and IP reuse. In particular, the concurrent programming paradigm is an ideal model of computation for design of embedded systems, which often encompass inherent concurrency.

Furthermore, embedded systems often have stringent performance requirements and, consequently, require a carefully selected and performance tuned embedded processor to meet specified design constraints. In recent years, a plethora of highly customized embedded processors have become available. As an example, Tensilica [11] provides a large family of highly customized application-specific embedded processors.

Such embedded processors ship with cross-compilers and the associated tool chain for application development. However, to support a multitasking application development environment, there is a need for an operating system (OS) layer that can support task creation, task synchronization, and task communication.

Such OS support is seldom available for each and every variant of the base embedded processor. In part, this is due to the lack of system memory and/or sufficient processor performance to afford the high performance penalty of having a full-fledged OS (e.g., in the case of micro-controllers such as the Microchip PIC [7] and the Philips 8051 [9]). Additionally, manually porting and verifying an OS to every embedded processor available is a high-cost job, in terms of time and money.

To fill the gap in realizing a multitasking application targeted at a particular embedded processor, researchers have proposed *Phantom* [8]. *Phantom* provides a fully automated source-to-source translator, taking a multitasking C program extended with POSIX as input and generating an equivalent, embedded processor independent, single-threaded ANSI C program, to be compiled using the embedded processor-specific tool chain. The output of *Phantom* is a highly tuned, correct (i.e., by construction) ANSI C program that embodies the application-specific embedded scheduler and dynamic multitasking infrastructure along with the user code.

In this work, we outline the inner workings of the *Phantom* serializing compiler. Specifically, we provide details on the architecture of *Phantom* generated code, such as memory layout, code organization, and the scheduler. Moreover, we analyze the quality of the *Phantom* generated code with respect to memory and processing overheads. Through our experiments, we show that such serializing compilers are extremely efficient, making them ideal to be used in design of highly parallel applications (e.g., multimedia, graphics, and signal processing applications).

Phantom is a new approach in addressing the challenge of multitasking support for embedded applications. We are unaware of any related work that use a similar compiler-based technique as that used in *Phantom*. However, we can identify three different approaches that address some of the design challenges solved with *Phantom*, namely a Virtual Machine (VM) based technique, template-based OS generation techniques, and static scheduling techniques. In the VM approach, portability is achieved, but with the overhead imposed by the VM layer. Moreover, the VM has to be ported to each new platform. To improve on these, solutions like JITs[1] and customized VMs for embedded platforms[13] have been proposed. In the template based OS generation, a custom OS is generated from a generic library of templates [4][5][12]. However, no single generic OS template can be used in the variety of embedded processors available. Finally, the static scheduling techniques [2][3][6] solve the static, a priori known, tasks class of problems, without addressing the dynamic multitasking issues.

The remainder of this work is organized as follows. In Section 2, we briefly describe *Phantom*, the source to source translator. In section 3, we discuss the architecture of the code generated by *Phantom*. In Section 4, we describe our experiments and give insight into the performance of code generated with *Phantom*. Finally, in Section 5, we state our conclusions.

2. THE PHANTOM APPROACH

2.1 Introduction

Input to *Phantom* is a multitasking program P_{input}, written in C. The multitasking is supported through the native *Phantom* API,

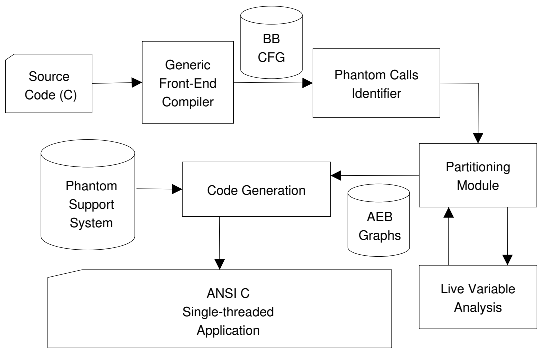


Figure 1: Phantom Compiler Architecture

```
typedef struct {
  int id;
  pthread_mutex_t *lock;
  pthread_mutex_t *unlock;
}game_t;
int winner;
void *game(void *arg) {  /* THREAD */
  game_t g = (game_t *)arg;
  int num;

  while(1) {
    pthread_mutex_lock(g->lock);
    if(winner) {
      pthread_mutex_unlock(g->unlock);
      return NULL;
    }
    else {
      num = rand();
      if(num == g->id)
        winner = g->id;
      pthread_mutex_unlock(g->unlock);
    }
  }
}
```

```
int main(int argc, char **argv) {
  pthread_t t1, t2;
  int r;
  struct game_t g1, g2;
  pthread_mutex_t m1, m2;

  pthread_mutex_init(&m1, NULL);
  pthread_mutex_lock(&m1);
  pthread_mutex_init(&m2, NULL);
  pthread_mutex_lock(&m2);
  g1.id = 1;
  g2.id = 2;
  g1.lock = g2.unlock = &m1;
  g2.lock = g1.unlock = &m2;
  winner = 0;
  pthread_create(&t1, NULL, game, &g1);
  pthread_create(&t2, NULL, game, &g2);
  pthread_mutex_unlock(&m1);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  printf("Winner is %d\n", winner);

}
```

Figure 2: Code Example

which is a subset of the standard POSIX interface[10]. These primitives provide functions for task creation and management (e.g., task_create, task_join, etc.) as well as a set of synchronization variables (e.g., mutex_t, sema_t, etc.). 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 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 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. 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 C data structures and synchronization APIs are included from the *Phantom* system support library, resulting in the final ANSI C single-threaded code.

In the current version, *Phantom* is able to handle soft, firm, and event-driven real-time applications. All the modules pictured in Figure 1 are implemented and can be used in the automatic code generation process. Next, we briefly present the major components of *Phantom*. For a more complete description of these components, including discussions about code partitioning, please refer to [8].

2.2 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 shown in Figure 2, implementing a game between two tasks that are picking up random numbers, and the corresponding CFG of function game of the example, pictured in Figure 3. Figure 3(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 3(b), and subsequently divides the function code into a number of AEBs,

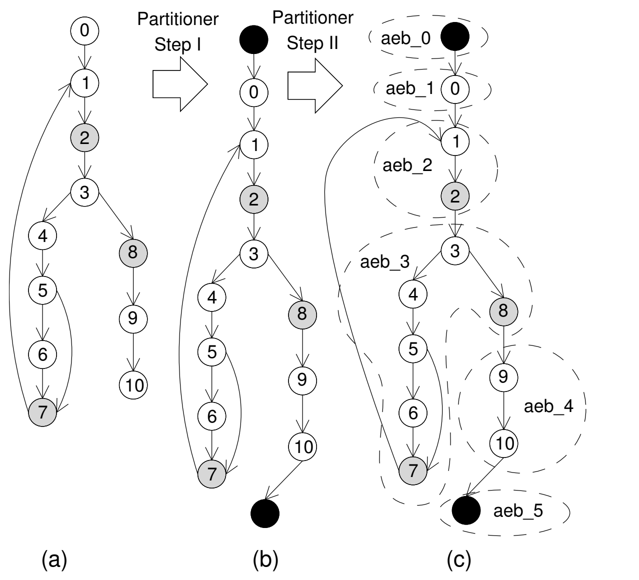


Figure 3: CFG Transformations for Function *game*

as shown in Figure 3(c), in a process we call *phantomization*.

Figure 3(c) shows the AEB graph of function game 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. A detailed description of the code layout and the scheduler implementation is in Section 3.

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 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 3(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 alternate 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 run-time, 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 is blocked waiting 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 3. 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.

2.3 Partitioning

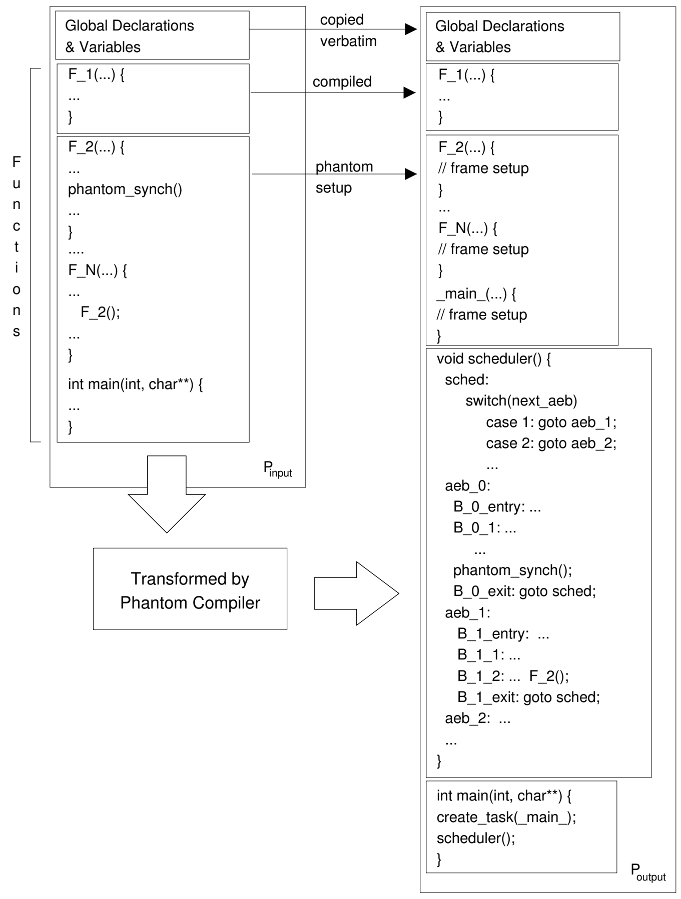
The partitioning of the code into AEBs is key to implementing multitasking at a high abstraction level. Recall that boundaries of AEB represent the points where tasks might be preempted or resumed for execution. Some partitions are unavoidable and must be performed for correctness, specifically, when a task invokes a synchronization operation, or when a task creates another task.

However, partitioning beyond what is needed for correctness impacts timing. In general, partitioning will determine the granularity level of the scheduling (i.e., the time quantum), as well as the task latency. A complete discussion on partitioning, with a deeper analysis of different partitioning schemes and algorithms for exploring different code partitions, is described in a separate work [8].

2.4 Experiments with Phantom

The *Phantom* approach has been successfully applied to a number of applications developed for testing the translation flow [8]. In summary, *Phantom* outperforms standard UNIX POSIX implementations, being 2 to 3 times faster in execution time [8]. 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.

With regard to different partitioning schemes, it was shown that there is a clear correlation between partition and performance metrics, like latency and multitasking overhead [8]. Different code partitioning can reduce application latency by as much as two orders of magnitude, at the expense of an increase of the multitasking overhead by a factor of 120 [8]. The effect of partitioning is highly dependent on the application structure itself.



```
F_1(...) {
...
}
F_2(...) {
...
phantom_synch()
...
}
....
F_N(...) {
...
  F_2();
...
}
int main(int, char**) {
...
}
```

```
F_1(...) {
...
}
F_2(...) {
// frame setup
}
...
F_N(...) {
// frame setup
}
_main_(...) {
// frame setup
}
void scheduler() {
  sched:
      switch(next_aeb)
          case 1: goto aeb_1;
          case 2: goto aeb_2;
          ...
  aeb_0:
    B_0_entry: ...
    B_0_1: ...
       ...
    phantom_synch();
    B_0_exit: goto sched;
  aeb_1:
    B_1_entry:  ...
    B_1_1: ...
    B_1_2: ...  F_2();
    B_1_exit: goto sched;
  aeb_2:  ...
  ...
}
int main(int, char**) {
create_task(_main_);
scheduler();
}
```

Figure 4: Code Layout of Input and Output Programs

3. GENERATED CODE ARCHITECTURE

3.1 Code Layout

The code layout of the input program P_{input}, once processed by a C pre-processor, is conceptually organized in two sections, as shown in Figure 4. The first section contains all global declarations and variables, while the second section contains a set of functions. One of these functions is the `main` function, i.e., the entry point of the application. The *Phantom* output program P_{output} is organized in five sections, as shown in Figure 4. The first section contains global declarations and variables. The second section contains a set of functions that are not phantomized. The third section contains a set of functions, each corresponding to one phantomized function of P_{input}. The fourth section contains a single function, called `scheduler`, which contains the code for all the phantomized functions, as well as the scheduling algorithm. Finally, the fifth section contains the `main` function of P_{output}.

All the functions of P_{input} are analyzed and classified in two groups: the *phantomized* and *non-phantomized* functions. A function is phantomized if (i) it is the entry point of a task, (ii) contains a synchronization primitive(s), or (iii) calls a phantomized function. Note that, since `main` is the entry point of the first task that is created by default, it is automatically phantomized.

The second section of P_{output} contains all non phantomized routines, copied over from P_{input}. The third section contains the setup functions, each corresponding to a phantomized function of P_{input}. A setup function is responsible for allocating the frame structure of each phantomized function. The frame and task context memory layout is described in a later subsection.

The next section of P_{output} contains the phantomized functions, along with the scheduler. All of these (i.e., the phantomized functions and scheduler) are embodied into a single C function of P_{output}, namely `scheduler`. Recall that a phantomized function is partitioned into a set of AEBs, $aeb_0, aeb_1, \ldots, aeb_n$. An AEB aeb_i is in turn composed of one or more basic blocks $B_{i,enter}, B_{i,2}, B_{i,3}, \ldots, B_{i,exit}$. By definition, execution of AEB aeb_i starts at the entry basic block of $B_{i,enter}$ and ends at the exit basic block $B_{i,exit}$. The exit basic block $B_{i,exit}$ of AEB aeb_i transfers control to a special basic block `sched` that serves as the entry point of the scheduling algorithm. The `scheduler` function contains all these basic blocks, starting with basic block `sched`, in low-level C, using C labels to denote basic block boundaries and C goto statements as a branching mechanism. The scheduling algorithm is described in a later subsection.

Finally, the fifth section of P_{output}, contains an implementation of the main function, which creates a single task, corresponding to the main entry point of P_{input}, and calls the `scheduler` function to invoke the scheduling algorithm.

3.2 Memory Layout

Each time a task is created, memory is allocated to store its context. At any given time, a special global variable, named `current`, is made to point to the context of the running task by the scheduler. Moreover, a queue of running tasks, named `tasks`, is maintained, according to the priorities of each task, by the scheduler, as described in the following subsection. The context of a task is further defined in Figure 5.

```
struct context_t {
  id        // unique id
  status    // runnable or blocked
  priority  // priority level
  next_aeb  // next aeb to execute
  stack     // stack for function frames
  waiting   // task waiting to join
  ret_val   // exit value of this task
}
```

Figure 5: The Task Context Data Structure

Most of the fields of this structure were discussed earlier. Here, we focus on the `stack` field of a context. The purpose of the stack is to store the task-local data of each phantomized function. Moreover, the choice of a stack is to allow for recursion and nested function calls. The collection of all this data for a phantomized function f is called f's *frame*, and is structured as shown in Figure 6. The frame of each phantomized function includes function arguments and local variables which are live at the boundary of its AEBs. The code in all basic blocks of f's AEBs access the most recent instance of f's frame.

```
struct f_frame_t {
  arg_0   // first arg. of phantomized function
  arg_1   // second arg. of phantomized function
  ...
  arg_N   // last arg. of phantomized function
  local_0 // live variable
  local_1 // live variable
  ...
  ret_aeb // next AEB of calling function
}
```

Figure 6: The Frame Data Structure

The stack is managed by the setup and the cleanup AEBs of phantomized functions. Specifically, when a function g of the current task calls a phantomized function f, the setup function f_{setup} is invoked. Then, f_{setup} pushes f's frame onto the stack of the current task, copies f's arguments to the frame, saves the return AEB of the calling function g, and makes the current task's next AEB point to the entry AEB of f. The structure of the setup function is shown in Figure 7.

```
void f_setup(arg_0, ... , arg_N) {
   f_frame_t *frame
   frame = &current->stack.buffer[current->stack.free]
   current->stack.top = current->stack.free
   current->stack.free += sizeof(f_frame_t)
   frame->arg_0 = arg_0
   ...
   frame->arg_N = arg_N
   frame->ret_aeb = current->next_aeb
   current->next_aeb = f_aeb_0
}
```

Figure 7: Code Structure of Setup Functions

Conversely, when a called function f complete its execution, the cleanup AEB aeb_{exit} of f performs the following. First, it restore the current task's next AEB to point to the next AEB of the calling function g, which was stored in the frame of f by the f's setup function. Then, it pops the frame of the current task's stack, as shown in Figure 8.

```
f_aeb_exit: {
   f_frame_t *frame
   frame = &current->stack.buffer[current->stack.top]
   current->next_aeb = frame->ret_aeb
   current->stack.free = current->stack.top
   current->stack.top -= sizeof(f_frame_t)
}
```

Figure 8: Code Structure of Cleanup AEB

3.3 Scheduler

The scheduler's code is included in the same C function containing the phantomized functions, called `scheduler`. The scheduling algorithm makes use of a priority queue that stores all the runnable tasks. The priority queue guarantees that the highest priority task is always the first task in the queue. In case of a priority tie among two or more tasks, the scheduler implements a round-robin scheme among them, so that all equal-priority tasks fairly share the processor. When a task is selected by the scheduler for execution, the global `current` pointer is updated accordingly.

Each AEB returns the execution to the scheduler upon termination. This is accomplished through a jump to the first basic block of the scheduler. Once the scheduler determines the next task T_i to be executed, it uses T_i's `next_aeb` reference to transfer control back to the next AEB. The transfer of control from the scheduler to the next AEB of the running task is implemented using a switch statement containing `goto`'s to all AEB's of the application. When the AEB completes execution, control is returned to the scheduler, which then pushes the current task's context back to the queue of runnable tasks if the task is not blocked or terminated. An overview of the scheduler is depicted in Figure 9.

```
queue_t *tasks
context_t *current
void scheduler() {
   while(tasks->size > 0) {
      sched: {
         if(current->status == RUNNABLE)
            tasks->push(current)
         current = tasks->pop()
         switch(current->next_aeb) {
            case 1: goto aeb_0
            case 2: goto aeb_1
            ...
         }
      }
   }
  // code for all the AEBs follows
}
```

Figure 9: Code Structure of Scheduling Function

An optimization in the scheduling algorithm allows a task to execute more than one AEB each time it is selected from the priority queue. We call this a *short context switch*. With the short context switch, it is possible to save the overhead of pushing/popping a new task from the priority queue with a bypass. A full context switch is executed every so often, alternating short and full context switches with a pre-determined ratio. A full context switch ensures a fair sharing of the processor among equal-priority tasks.

In order to implement the short context switch, we add a counter to the scheduling algorithm, used to keep track of the number of consecutive short context switches performed. The counter is initialized to a value representing the ratio between short and full context switches. The value of the counter defines a *time quantum*, i.e., a number of consecutives AEBs of the same task to be executed before a full context switch. The counter is decremented at every short context switch, and a full context switch is executed once the counter reaches zero and expires. Obviously, a full context switch can happen before the counter expires, in the case that a task is blocked or terminates. Alternatively, a timer can be used in place of a counter, yielding a real *time-sharing* of the processor in the round-robin approach. Figure 10 shows the optimized scheduler algorithm, incorporating the short context switch optimization.

In *Phantom*, for efficiency reasons, a limited priority queue is implemented. A limited priority queue is one that allows a finite, and a priori known, number of priority levels (e.g., 32). However, this does not pose any limitations, since the number of priority levels, required by the application, can be provided to the *Phantom* serializing compiler. The implementation of the priority queue is as follows. A separate array-based queue is allocated for each priority level, which are accessed by the scheduler in order of highest to lowest priority. Manipulation of the array-based queues at each priority level is very efficient, and takes constant time. At any given

```
queue_t *tasks
context_t *current
void scheduler() {
   while(tasks->size > 0) {
      if(current->status == RUNNABLE)
         tasks->push(current)
      current = tasks->pop()
      cnt = RATIO;
      sched: {
        if(cnt-- && current->status == RUNNABLE)
            switch(current->next_aeb) {
               case 1: goto aeb_1
               case 2: goto aeb_2
               ...
            }
      }
   }
  // code for all the AEBs follows
}
```

Figure 10: Code Structure of Optimized Scheduling Function

point, a reference is maintained to the highest non-empty priority queue. Given this, the overall access to the queue of runnable tasks by the scheduler requires constant running time, regardless of the number of runnable tasks.

4. EXPERIMENTAL RESULTS

A set of synthetic benchmarks was implemented to evaluate the overhead imposed by the Phantom multitasking infrastructure. Various parameters of *Phantom* were evaluated, like context switching overhead, task creation cost, task joining cost, and mutex synchronization cost.

Cost was measured as the average number of instructions executed on the host processor for performing a particular operation (e.g., task creation, task joining, etc.) We compiled and executed the applications on a UltraSPARC-IIe workstation, 500 Mhz, with 256Mb of RAM running Solaris operating system. We used `cputrack` tool to obtain number of instructions and CPU cycles executed by a target program. (`cputrack` uses hardware counters to track CPU usage). All benchmarks were compiled with GCC v3.3. The time cost of each metric was calculated from the average CPI (cycles per instruction) of each benchmark, associated with the processor cycle time.

For each benchmark, designed to measure a particular metric, we first obtained a baseline execution count. The baseline execution count accounted for all the computation code less the *Phantom* generated multitasking infrastructure. Then, the multitasking infrastructure was enabled and instruction counts were re-evaluated. The difference between the baseline and the version with the multitasking infrastructure gave us a measure of the performance of *Phantom* for that metric. All experiments in this phase were performed using at most one task active and a single priority level. On average, *Phantom* multitasking infrastructure overhead is small, and has an impact of less than 1% in the execution time of the synthetic benchmarks. Our results are summarized in Table 1.

Next, we evaluated the impact of multiple task and multiple priorities in task context switch. In these experiments, we used a mixed scheduler, with a 10:1 ratio between short and full context switch. Figure 11 shows the results. Here, the horizontal axis of the plot depicts the number of runnable tasks in the system (i.e., one of 2, 10, 20, 50, 100, 500, and 1000 tasks). The vertical axis of the plot depicts the average number of instructions for performing a context switch.

We note from Figure 11 that the overhead of context switch is

Table 1: ***Phantom*** **Multitasking Performance Results**

Metric	No optimization (-O0)		With optimization (-O2)	
	Instructions	Time (μs)	Instructions	Time (μs)
full context switch	427	1.81	206	0.47
short context switch	82	0.35	37	0.08
mixed context switch (10:1)	124	0.52	58	0.13
task creation	1113	4.74	833	1.90
task join	506	2.15	227	0.52
mutex lock	68	0.29	40	0.09

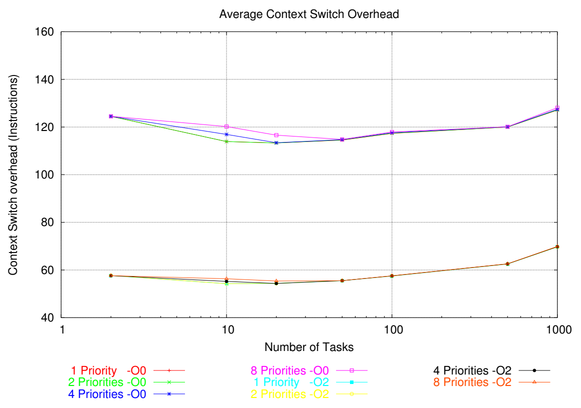


Figure 11: ***Phantom*** **Context Switch Cost in Instructions with Multiple Threads and Multiple Priorities**

small, *fairly constant*, and *independent* of the number of runnable tasks in the system. A similar result was obtained for task creation overhead. Contrary to intuition, there is initially a slight decrease in the context switch time when the number of tasks increase. With a small number of tasks, there are more reorganizations in the priority queue, since every context switch can possibly insert a task with a different priority in the queue. As the number of tasks increase, reorderings are less constant, since a task with the same priority is likely to be in the queue already. Therefore, context switch is slightly faster. Nevertheless, the impact of *Phantom* in the execution time of the benchmarks is typically less than 1%, for the applications tested. A similar trend is observed with respect to the number of priorities, i.e., increasing the number of priorities does not have a significant impact on context switch time. As before, there is a slight difference in context switch time when few tasks are present. In this case, the priority queue has to be reorganized more often, increasing the context switch by a small margin. The efficiency of the *Phantom* generated code makes it practical for designing multimedia, digital signal processing, or other highly parallel applications, using the concurrent programming model.

5. CONCLUSIONS

In this work, we have outlined the code architecture and memory layout of *Phantom* serializing compiler. A serializing compiler is a source-to-source translator that takes a POSIX compliant multitasking C program as input and generates an equivalent, embedded processor independent, single-threaded ANSI C program, to be compiled using the embedded processor-specific tool chain. Serializing compilers have been proposed as an alternative solution, enabling a designer to develop multitasking applications without the need of OS support. We have also analyzed the quality of the generated code with respect to memory and processing overheads. Our results show that such serializing compilers are extremely efficient, making them ideal to be used in design of highly parallel applications (e.g., multimedia, graphics, and DSP applications).

Our future direction of research is to incorporate real-time scheduling into the *Phantom* serializing compiler. Specifically, we plan to allow deadline driven task scheduling using dynamic priorities. In addition, we would like to investigate power management strategies (e.g, voltage scaling, power modes, memory management, etc.) to be automatically generated by the *Phantom* serializing compiler.

Acknowledgment

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

6. REFERENCES

[1] J. Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, Jun. 2003.

[2] J. Cortadella et. al. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *Proc. of DAC*, Jun. 2000.

[3] S. Edwards. Tutorial: Compiling Concurrent Languages for Sequential Processors. *ACM Trans. on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.

[4] L. Gauthier, S. Yoo, and A. Jerraya. Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1293–1301, Nov. 2001.

[5] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *Proc. of DATE*, Mar. 2003.

[6] B. Lin. Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling. In *Proc. of DATE*, Feb. 1998.

[7] Microchip Inc. http://www.microchip.com.

[8] A. Nacul and T. Givargis. Code Partitioning for Synthesis of Embedded Applications with Phantom. In *Proc. of ICCAD*, Nov. 2004.

[9] Phillips Inc. http://www.philips.com.

[10] POSIX Open Group. http://www.opengroup.org.

[11] Tensilica Inc. http://www.tensilica.com.

[12] 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.

[13] 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.