The Phantom Serializing Compiler

André C. Nácul

Tony Givargis

Technical Report CECS-04-30

November 22, 2004

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{nacul, givargis}@cecs.uci.edu

The Phantom Serializing Compiler

André C. Nácul

Tony Givargis

Technical Report CECS-04-30

November 22, 2004

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{nacul, givargis}@cecs.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 design errors, design portability, and Intellectual Property (IP) reuse. In a large class of embedded systems, dynamic multitasking using traditional OS techniques is infeasible because of memory and processing overheads or lack of operating systems availability for the target embedded processor. We propose a serializing compiler as an alternative solution to enable a designer to develop multitasking applications without the need of OS support. 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. The output of our tool is a highly tuned ANSI C program that embodies the application-specific embedded scheduler and dynamic multitasking infrastructure along with the user code. In this work, we present the Phantom serializing compiler, discuss its architecture and scheduling technique, and show the feasibility of the proposed approach by comparing execution efficiency to solutions based on traditional OS implementations.

Contents

List of Figures

The Phantom Serializing Compiler

André C. Nácul, Tony Givargis

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

{nacul,givargis}@cecs.uci.edu

http://www.cecs.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 design errors, design portability, and Intellectual Property (IP) reuse. In a large class of embedded systems, dynamic multitasking using traditional OS techniques is infeasible because of memory and processing overheads or lack of operating systems availability for the target embedded processor. We propose a serializing compiler as an alternative solution to enable a designer to develop multitasking applications without the need of OS support. 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. The output of our tool is a highly tuned ANSI C program that embodies the application-specific embedded scheduler and dynamic multitasking infrastructure along with the user code. In this work, we present the Phantom serializing compiler, discuss its architecture and scheduling technique, and show the feasibility of the proposed approach by comparing execution efficiency to solutions based on traditional OS implementations.

1 Introduction

The functional complexity of embedded software continues to rise due to a number of factors such as consumer demand for more functionality, sophisticated user interfaces, seamless operation across multiple communication and computation protocols, need for encryption and 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 design errors, design portability, and Intellectual Property (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 (e.g., timing, energy, etc.) 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 [15] provides a large family of highly customized application-specific embedded processors (a.k.a., the Xtensa). Likewise, ARM [2] and MIPS [12] provide several derivatives of their respective core processors, in an effort to provide to their customers an application-specific solution.

These 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 (e.g., in the case of micro-controllers such as the Microchip PIC [10] and the Phillips 8051 [13]) coupled with the high performance penalty of having a full-fledged OS. Additionally, manually porting and verifying an OS to every embedded processor available is a high-cost job, in terms of time and money, and yet does not guarantee correctness.

Thus, there exists a gap in realizing a multitasking application targeted at a particular embedded processor, as shown in Figure 1. In this work, we fill this gap by providing a fully automated source-to-source translator, the *Phantom* compiler, that takes a multitasking C program as input and generates an equivalent, embedded processor independent, single-threaded ANSI C program, to be compiled using the embedded

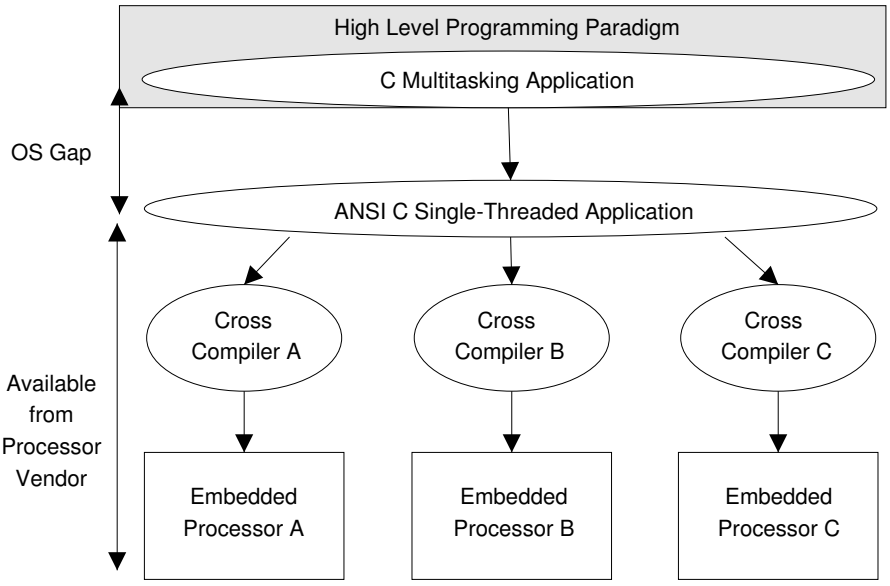

Figure 1: OS Gap in Embedded Software Designs

processor-specific tool chain. The output of our tool is a highly tuned, correct-by-construction ANSI C program that embodies the application-specific embedded scheduler and dynamic multitasking infrastructure along with the user code.

The remainder of this work is organized as follows. In Section 2, we summarize prior related work. In Section 3, we present the *Phantom* approach. Section 4 discusses the partitioning problem from the *Phantom* perspective. In Section 5, the code and memory layout, and scheduler organization are detailed. Section 6 contains experimental results, comparing *Phantom* to traditional approaches and showing the effect of different partitioning solutions. Finally, in Section 7 we conclude the paper.

2 Related Work

There are three categories of solutions that partially address the problem stated in this work, namely, a class of virtual machine (VM) based techniques, a class of template based OS generation techniques, and a class of static scheduling techniques.

2.1 VM Based Techniques

In the VM based techniques, an OS providing a multitasking execution environment is implemented to run on a virtual processor. A compiler for the VM is used to map the application program onto the VM. The virtual processor is in turn executed on the target processor. Portability here is achieved by porting the VM to the desired target embedded processor. The advantages of this class of techniques are that the application and OS code do not require recompilation when moving to a different embedded processor. The disadvantage of this class of techniques is a significant performance penalty (i.e., speed, energy, and memory footprint) incurred by the VM layer, and specifically the VM instruction set interpreter. Moreover, the porting of the VM to the target embedded processor may require more than recompilation efforts. Examples of such VM based techniques are Java [8] and C# [11]. Research in this area tries to address the above-mentioned disadvantages by proposing customized VM for embedded applications [17] or just in time (JIT) compilation techniques [3].

2.2 Template Based Techniques

In the template-based OS generation techniques, a reference OS is used as a template in generating customized derivatives of the OS for particular embedded processors. This class of techniques mainly relies on inclusion or exclusion of OS features depending on application requirements and embedded processor resource availabilities. The disadvantage of this class of techniques is that no single generic OS template can be used in the variety of embedded processors available. Instead, for optimal performance, a rather customized OS 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. [7] 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 OS partitions. The OS partition is subsequently refined to a final implementation. The mechanism used in this refinement is based on matching needed OS functionality against a library of OS functions. In a similar approach, Vercauteren et al. [16] have proposed a method based on an API providing OS primitives to the application programmer. This OS template is used to realize the subset of the API that is actually used in the

application program. Finally, Gauthier et al. [6] have proposed an environment for OS generation similar to the previous approaches. Here, a library of OS components that are parameterized is used to synthesize the target OS given a system level description of application program.

2.3 Static Scheduling Techniques

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. Our technique specifically addresses the dynamic multitasking issue. Moreover, our technique is orthogonal to such static scheduling. For example, the set of a priori known static tasks can be scheduled using static scheduling while the dynamically created tasks can be handled by a technique similar to ours.

A very good general survey on generating sequential code for a static set of tasks is done by Edwards [5]. In a more specific example, Lin [9] has proposed a technique that takes as input an extended C code that includes primitives for inter-task 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 and generate code simulating a correct execution order of the Petri Net. Similar techniques have also been proposed by Cortadella et al. [4]. One important aspect to note in both Lin's and Cortadella approaches is that the generated code could still be multitasking, thus requiring the existence of an OS layer that can schedule and manage the generated tasks.

3 The Phantom Approach

3.1 Introduction

Input to our translator is a multitasking program P_{input}, written in C. The multitasking is supported through the native *Phantom* API, which complies with the standard POSIX interface [14]. These primitives provide functions for task creation and management (e.g., task_create, task_join, etc.) as well as a set of synchroniza-

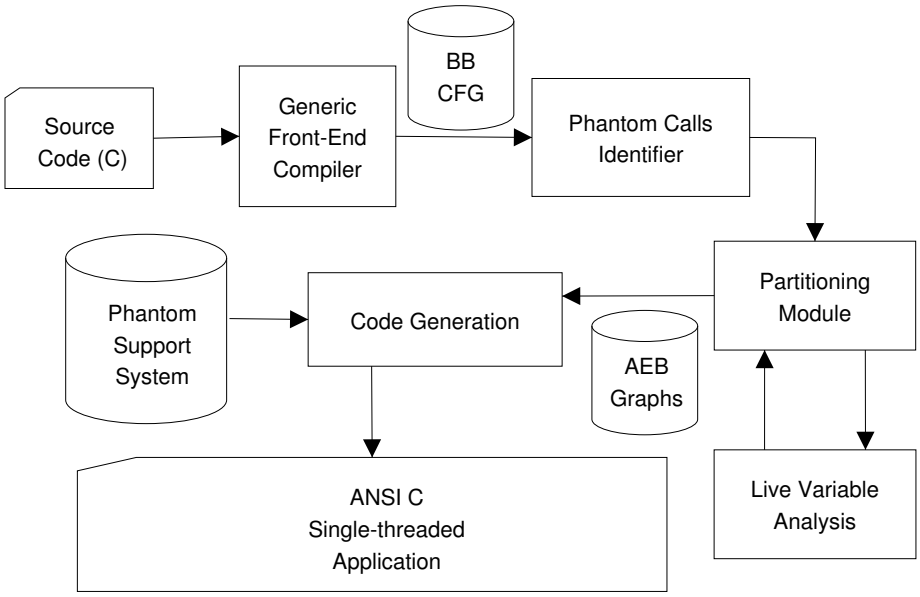

Figure 2: Phantom Compiler Architecture

tion variables (e.g., mutex_t, sema_t, etc.). Output of our system is a single-threaded strict ANSI C program P_{output} that is equivalent in function 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.

In order to support multitasking, there is a need for efficient sharing of the processor among multiple tasks, providing synchronization mechanisms, and communication primitives. Sharing of the processor among tasks requires preemption and, in turn, preemption requires a mechanism for saving/restoring task specific information (i.e., the task context). In conventional approaches, multitasking is implemented within the OS. When a task T_i is created, the OS allocates sufficient memory for saving T_i's context information (e.g., registers, function call stack, program counter, etc.). Periodically, an interrupt generated by the system timer invokes, among other things, the OS scheduler. The scheduler saves the context of the currently executing task T_{old} and restores the context of a new task T_{new} to be executed. The OS, in turn, relies on the underlying processor for invoking the scheduler (i.e., via a timer interrupt), context switching (register load/store instructions), and synchronization (i.e., test-and-set instruction).

In our approach, the challenge is to achieve the same at a higher level of abstraction, namely, by using the mechanisms provided by strict ANSI C language. Figure 2 is the block diagram of the *Phantom* Compiler.

The multitask 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 we call AEBs (Atomic Execution Blocks), to be executed by the scheduler. Every task in the original code is potentially 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 from *Phantom* are included from the *Phantom* system support library, resulting in the final ANSI C single-threaded code.

Next, we discuss the major components of *Phantom*, presenting implementation details for the source-level multitasking framework. Throughout the next sections, we will be referring to our running example shown in Figure 3. Our running example implements a simple game between two tasks that are picking up random numbers until one of them picks its own *id*, making it the winner of the game.

3.2 Preemption and Scheduling

Since the output of *Phantom* is a single-threaded program, the first problem faced is how to simulate a multitasking system with a single-threaded code, using ANSI C resources. In order to schedule the different tasks, we need to define a context switching mechanism and a basic unit of execution.

As mentioned earlier, 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. For example, Figure 4 pictures the CFG transformations for the function *game* of our running example. Figure 4(a) shows the output of the compiler front end that is fed to the partitioning module. The partitioner adds two control basic blocks, *setup* and *cleanup*, as shown in Figure 4(b), and subsequently divides the code into a number of AEBs, as shown in Figure 4(c).

Figure 4(c) shows the AEB graph of function `game` as being composed of AEBs *aeb_0*, *aeb_1*, *aeb_2*,

```
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 3: Code Example

aeb_3, *aeb_4* and *aeb_5*. Within an AEB graph, the *setup* basic block is implemented as a function, with the appropriate parameters derived from the original function in the multitasking C source. All the other AEBs are implemented as a region of code, composed of one or more basic blocks, with a corresponding entry label. For instance, *aeb_3* implementation is shown in Figure 5 (label `game_aeb3`). The termination of an AEB region transfers the control back to the scheduler (Figure 5, label `sched`). 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 5.

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 a special function called f_{setup}, and cleaned up

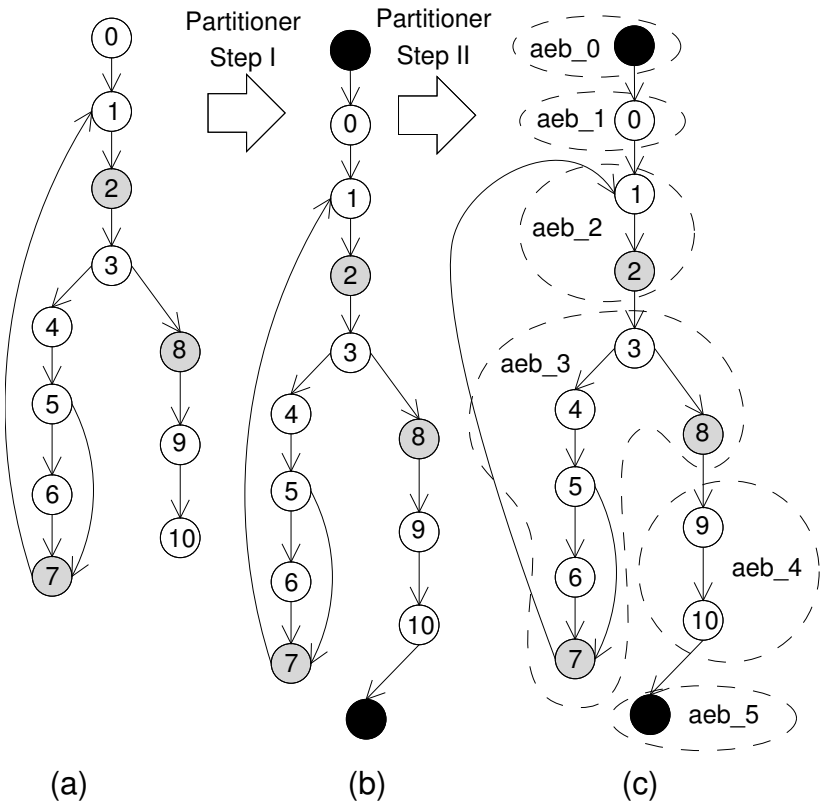

Figure 4: CFG transformations for function *game*

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 4(b). For an example of the generated ANSI C code, refer to Figure 5, function `game`, for *setup*, and label `game_aeb5`, for *cleanup*.

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 (Figure 5, structure `context_t`). 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 a queue of existing task, called `tasks`, to be processed by the embedded scheduler.

We note that in *Phantom* implementation, the partitioning is performed on the basic block intermediate representation of the input source program. In that sense, almost no high level code constructs, like while, for loops, and switch statements are preserved in the equivalent ANSI C output (see Figure 5, label `game_aeb2`). Moreover, we note that an AEB node may be composed of one or more basic blocks. Code

```
typedef struct {
  int           id;
  status_t      status;
  task_info_t   info;
  stack_t       frames;
  join_info_t   join_info;
  aeb_t         next_aeb;
  void          *ret;
}context_t;
char *game(void *arg, void **ret_val){
  // allocate and setup frame
  frame = push(...);
  frame->arg = arg;
  // save the ret_val in the frame
  frame->ret = ret_val;
  // setup next aeb
  current->next_aeb = 1;
  return frame;
}
context_t *current;
static pqueue_t tasks;
```

```
static void scheduler() {
  while(queue_size(&tasks) > 0) {
 sched:
    if(current->status == RUNNABLE)
      queue_push(&tasks, current);
    current = queue_pop(&tasks);
    if(current->next_aeb != 0) {
      switch(current->next_aeb) {
        case 1: goto game_aeb3;
        case 2: goto game_aeb2;
        case 3: goto game_aeb5;
        case 4: ......
        .....
      }
    }
game_aeb5: {
  // clean up frame structure
  frame = pop(current->frames);
  free(frame);
  goto sched
}
```

```
game_aeb2: {
  // restore locals from frame
  game_t g = frame->g;
  if(1){
    current->next_aeb = 1;
    pthread_mutex_lock(g->lock);
  }
  goto sched;
}
game_aeb3: {
  int num;
  // restore locals from frame
  game_t g = frame->g;
  if(!winner) goto bb_4;
  current->next_aeb = 7;
  pthread_mutex_unlock(g->unlock);
  goto exit;
bb_4:
  num = rand();
  if(num != g->id) goto bb_7;
  winner = g->id;
bb_7:
  current->next_aeb = 2;
  pthread_mutex_unlock(g->unlock);
exit:
  goto sched;
}
........
........
```

Figure 5: Excerpt of the Generated Code

partitioning and its implications on runtime behavior are described in Section 4.

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. A zeroed `next_aeb` indicates that T_i has reached its termination point, and thus is removed from the queue of existing tasks.

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, priority inversion, and so on. Additionally, priorities can also be changed at run-time, so that scheduling algorithms based on dynamic priorities can be implemented.

3.3 Synchronization

Phantom implements the basic semaphore (`sema_t` in POSIX) synchronization primitive, upon which any

other synchronization construct can be built. A semaphore is an integer variable with two operations, *wait* and *signal* (`sema_wait` and `sema_post` in POSIX). A task T_i calling *wait* on a semaphore S will be blocked if the S's integer value is zero. Otherwise, S's integer value is decremented and T_i is allowed to continue. T_i calling *signal* on S will increment S's integer value and unblock one task that is currently blocked waiting on S. 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 4. 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.

Right before any synchronization, an AEB will set the task's `next_aeb` to the successor AEB according to the AEB Graph. If the task is not blocked at the synchronization, it will continue and the `next_aeb` will be executed next. Otherwise, the `next_aeb` will be postponed, and it will be executed as soon as the task is released on the synchronization point.

3.4 Interrupts

Preempting an AEB when an interrupt occurs would break the principle that every AEB executes until completion without preemption. Instead, in *Phantom*, the code for an interrupt service routine I is treated as a task, with its associated AEBs. On an interrupt destined for I, a corresponding task is created, having a priority higher than all existing tasks. Note that if multiple interrupts destined for I occur, multiple tasks will be created and scheduled for execution. This is a uniform and powerful mechanism for handling interrupts in a multitasking environment. However, the latency for handling the interrupt will depend on the average execution time of the AEBs, which in turn depends on the partitioning scheme used. Some interrupts

may need architecture specific code, like those associated with some device drivers. Architecture specific constructs in the original code are preserved by the *Phantom* serializing compiler, and copied verbatim to the output.

4 Partitioning

The partitioning of the code into AEB graphs is the key to implementing multitasking at a high-level of abstraction. 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. In the case when a task invokes a synchronization operation and thus is blocked, the embedded scheduler must regain and transfer control to one of the runnable tasks. Likewise, when a task creates another, possibly higher priority task, the embedded scheduler must regain and possibly transfer control to the new task in accordance with the priority based scheduling scheme. Additionally, the programmer can also manually specify points in the code where a context switch should happen by calling the `yield` function of the *Phantom* API.

Any original multitasking C program is composed of a set of functions (or routines). In *Phantom*, and for correctness, all functions that are the entry point of a task need to be partittioned. In addition, and for correctness, any function that invokes a synchronization primitive also needs to be partitioned. We call the process of partitioning functions into AEBs *phantomization*. Finally, and for correctness, a function that calls a *phantomized* function also needs to be *phantomized*. To illustrate why this is, consider `f` calling a *phantomized* function `g`. Upon termination of `g`, the scheduler must transfer control back to `f`. Since transfer of control in *Phantom* is achieved through a branch in the scheduler, `f` must at least be decomposed into two blocks, $\texttt{f}_1$ and $\texttt{f}_2$. Moreover, $\texttt{f}_1$'s last instruction will be the instruction that transfered control to `g`, and $\texttt{f}_2$'s first instruction will be the one immediately following the call to `g`. However, partitioning beyond what is needed for correctness impacts timing issues as described next.

In general, partitioning will determine the granularity level of the scheduling (i.e., the time quantum), as well as the task latency. A good partitioning of the tasks into AEBs would be one where all AEBs have approximately the same average case execution time μ and a relatively low deviation δ from the average, which can be computed if the average case execution time of each AEB is known. In this case, the application

would have a very predictable and stable behavior in terms of timing. Note that the average case execution time W_i of an AEB N_i is defined as the time taken to execute the code C_i in N_i plus the time taken to store and restore all live variables V_i at the entry and exit of N_i. Moreover, an estimate of V_i can be obtained by performing a live variable analysis. An estimate of C_i can be obtained by static profiling.

The range of partitioning granularities is marked by two scenarios. On one end of the spectrum, partitioning is performed only for correctness, and yields cooperative multitasking[1]. On the other end of the spectrum, every basic block is placed in its own partition, resulting in a preemptive multitasking with extremely low latency, but high overhead. Specifically, to evaluate a partition we can apply the following metrics, *average*, *minimum*, and *maximum* latency; *standard deviation* of latency; and context switch *overhead*. Clearly, to shorten latency, there is a need to context switch more often, and thus pay a penalty in terms of overhead.

In the next sections, we explore the range of partitioning possibilities, defining a strategy for clustering and an exploration framework for obtaining a set of Pareto-Optimal partitions.

4.1 Strategy for Clustering

The generic clustering algorithm used to group basic blocks into partitions that correspond to AEBs is based on two algorithms traditionally used for data flow analysis by compilers, namely *interval partitioning* and *interval graphs* [1]. The generic clustering algorithm takes as input a CFG, and returns a set of disjoint clusters, each cluster grouping one or more of the basic blocks of the original CFG. The generic clustering algorithm ensures that a cluster of basic blocks has a single entry point (i.e., the head of the cluster), but possibly multiple exit points. This requirement is necessary since every cluster is implemented as a non-preemptive block of code, with one single entry.

Our generic clustering technique is shown in Figure 6. Initially, for a given CFG and its entry basic block n_0, a set of clusters is computed, each containing one (reachable from n_0) basic block of the CFG (line 3). Subsequently, pairs of clusters c_i, c_j are merged if all of c_j's predecessors are in cluster c_i. The predecessors of c_j are all clusters containing one or more basic block(s) that are predecessor(s) of at least one basic block in c_j. The algorithm iterates until no more clusters can be merged.

[1]Cooperative multitasking is when tasks explicitly yield to each other or are preempted by a synchronization primitive.

Input: $cfg, n_0 \in cfg$ the entry point of the CFG
Output: clusters $c_1, c_2, \ldots, c_n$
  $clust \leftarrow \{c_i \leftarrow b_i | b_i \in cfg$ and reachable from $n_0\}$
  $changed \leftarrow 1$
  while $changed = 1$
    $changed \leftarrow 0$
    for each $c_i, c_j \in clust$
      if every pred. of c_j is in c_i
        $c_{new} \leftarrow c_i \cup c_j$
        $clust \leftarrow (clust - c_i - c_j) \cup \{c_{new}\}$
        $changed \leftarrow 1$
      endif
    endfor
  endwhile

Figure 6: The Generic Clustering Algorithm

Note that if the algorithm described in Figure 6 were to run on a CFG it would cluster all the basic blocks into a single partition, as expected. Therefore, we introduce a mechanism to modify the input CFG such that, using the same algorithm, we obtain a desired partitioning for correctness and timing. The mechanism is to modify the original CFG with two special empty basic blocks, *synch-mark* and *time-mark*. Neither of these marker basic blocks are reachable from the entry basic block n_0, and are, for that reason, not a member of a cluster (line 3). All points of partitioning that are required for correctness or timing will be pointed to by one of the markers prior to running the algorithm shown in Figure 6.

Figure 7 shows, step-by-step, the working of the clustering algorithm. Figure 7(a) is the CFG for the function `game`, augmented with the *setup* and *cleanup* basic blocks, where gray nodes represent those basic blocks with a synchronization point. Figure 7(b) shows the addition of the *synch-mark* basic block *s*. Next, every reachable basic block b_i of the sample CFG is assigned to cluster c_i as shown in Figure 7(c). Then, by successive iterations, clusters are merged until the final partitioning is reached, as shown in Figure 7(c)-(f).

The introduction of the *synch-mark* block is taken care of by the *Phantom* compiler. The introduction of the *time-mark* is performed by the exploration framework, to be described later. In other words, the exploration of the different partitions and the search for the Pareto-Optimal set of partitions is a matter of determining the set of basic blocks that the *time-mark* points to.

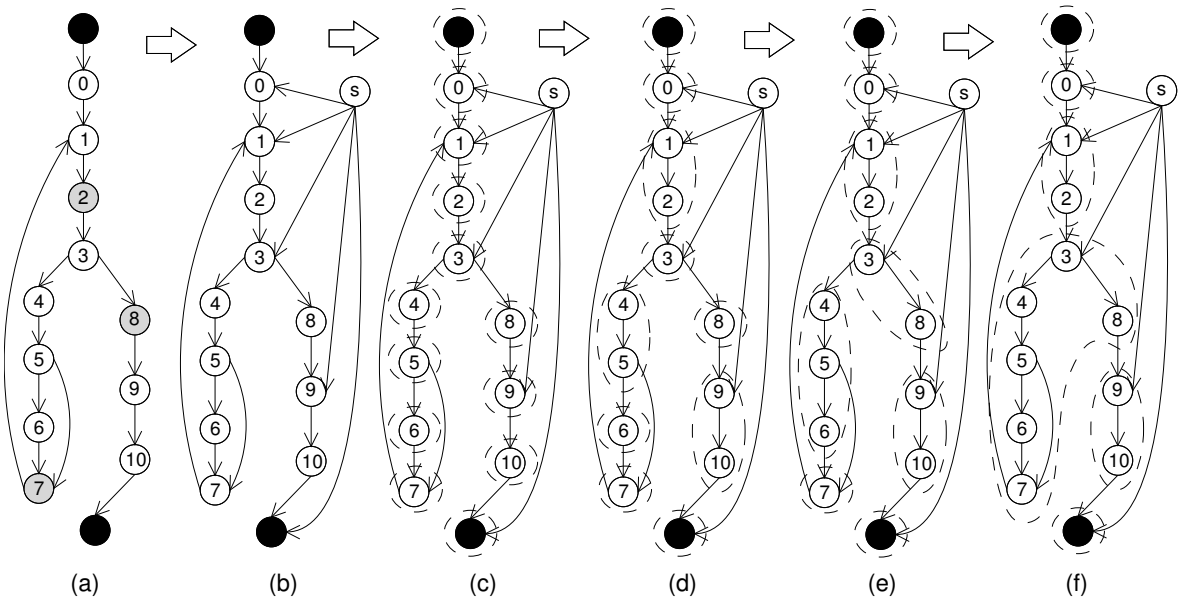

Figure 7: Execution of Clustering Algorithm

4.2 Exploration Framework

Our overall exploration framework is pictured in Figure 9 and works as follows. Initially, the multitasking application is processed by the *Phantom* compiler, as shown in Figure 2, using the cooperative partitioning scheme. Then, the generated code is instrumented with profiling instructions (i.e., basic block execution counters). Next, the instrumented code is executed and a trace containing profiling information is retrieved. Moreover, traces obtained from multiple runs of the same instrumented code but different input are merged to obtain a single representative trace (i.e., by averaging the basic block counts). The trace is then processed to extract performance numbers for each possible partition.

A partition is defined in terms of a set of edges from the *time-mark* basic block to the basic blocks of the original CFG. Thus, given a CFG with *N* basic blocks, there are an exponential number of possible ways to introduce such edges, hence there are an exponential number of possible partitions. For each partition, and using the profiling data, we can quickly compute all the evaluation metrics. Our search goal is to obtain a set of Pareto-Optimal [2] partitions that tradeoff latency, context switch overhead, and other metrics.

Our exploration technique employs a simple heuristic to obtain different clusters and is shown in Figure

[2]In a multi-objective optimization problem, a Pareto-Optimal set contains design instances where each design instance is guaranteed to be optimal with respect to at least one objective.

Input: cfg
Input: K number of tries
Output: $cfg_1, cfg_2, \ldots, cfg_n$ where $n = |cfg|$
  $N \leftarrow |cfg|$ number of basic blocks in cfg
  for $i = 1$ to N
    for $j = 1$ to K
      pick i random basic blocks in cfg
      place an edge from *time_mark* to basic block i
      execute Algorithm Fig. 6 and evaluate metrics
    endfor
  endfor

Figure 8: The Search Heuristic

8. For a CFG with N basic blocks, our algorithm attempt K random placements of $1, 2, \ldots N$ edges from the *time-mark* to basic blocks of the CFG. The parameter K is an arbitrary number and depends on the amount of compute time available for exploration. Clearly, larger values for K are expected to yield a better approximation of the Pareto-Optimal set. Although simple, this heuristic allows us to quickly reach a reasonably good number of partitions and obtain a fairly good approximation of the Pareto-Optimal set.

Once the Pareto-Optimal set is computed, there is the final process of selecting the best cluster to meet the application constraints. To do this, there are three different possibilities. The first is to have the designer select the desired partition by examining the Pareto-Optimal set. Another alternative is to apply a single constraint (e.g., specifying either a minimum latency, or maximum overhead) and let the tool select the partition that meets the constraint while optimizing the other metrics. Finally, it is possible to define a cost function (e.g., a weighted sum of the various metrics) to compute a unique goodness measure for each point in the Pareto-Optimal set, allowing the tool to select the partition with the minimum cost.

5 Architecture of Generated Code

5.1 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 10(a). 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

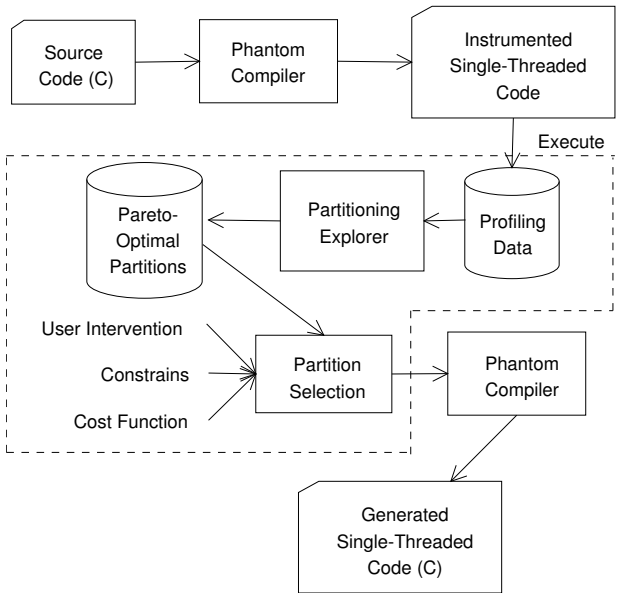

Figure 9: Clustering Exploration Methodology

entry point of the application. The *Phantom* output program P_{output} is organized in five sections, as shown in Figure 10(b). 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}. We describe each of these sections in more detail next.

The first section contains global declarations and variables, which are copied verbatim from P_{input}.

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}. In the current implementation of *Phantom*, non-phantomized functions are compiled into intermediate form by the front-end, and re-assembled into an equivalent low level C representation by the back-end. Thus, while functionally identical, the non-phantomized functions of P_{output} lack the high level code constructs (e.g., loops) found originally in P_{input}.

The third section contains the setup functions, each corresponding to a phantomized function of P_{input}.

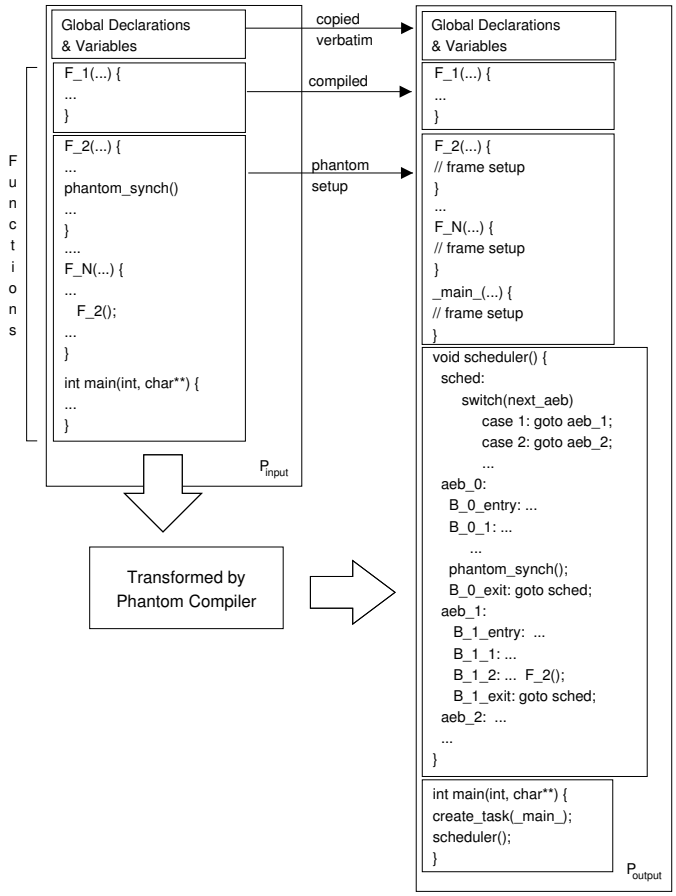

Figure 10: Code Layout of Input and Output Programs

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.

5.2 Memory Layout

As described earlier, 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 the priorities of each task, by the scheduler, as described in the following subsection. The context of a task is further defined in Figure 11.

```
struct context_t {
  id              // an integer unique identifier
  status          // one of runnable or blocked
  priority        // one of possible priority levels
  next_aeb        // a reference to the next aeb to be executed
  stack           // an array based stack set aside for function frames
  waiting         // a reference to a task waiting to join this task
  ret_val         // memory to hold the exit value of this task
}
```

Figure 11: 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 12. 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.

The stack is managed by the setup functions 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 13.

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

Figure 12: The Frame Data Structure

```
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 13: 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 14.

```
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 14: Code Structure of Cleanup AEB

5.3 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.

As stated eariler, 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. (This level of indirection is necessary because ANSI C does not allow for indirect jumps.) 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 15.

```
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 15: Code Structure of Scheduling Function

An optimization in the scheduling algorithm allows a task to execute more than one AEB each time it

```
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 16: Code Structure of Optimized Scheduling Function

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 16 shows the modified scheduler algorithm, incorporating the short context switch optimization.

In *Phantom*, and 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 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.

6 Experimental Results

The *Phantom* approach described in this paper was successfully applied to a number of applications developed for testing the translation flow. In general, multitasking applications synthesized with *Phantom* showed a much improved performance. The reason is two fold. Firstly, 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.

Eight different applications were implemented using the *Phantom* POSIX interface, so that its performance could be compared to the Unix implementation of POSIX. Unix represent a generic OS layer, similar to those that would be found in a traditional multitasking environment for embedded systems. The benchmarking applications that were used in the experiments are described in Table 1.

6.1 General Execution

Table 2 summarizes the performance of the benchmarks with *Phantom* and POSIX. All benchmarks were executed on an UltraSPARC-IIe workstation with 256Mb of RAM. One can easily see that*Phantom* outperforms standard UNIX-POSIX implementations, being 2 to 3 times faster in execution time. Figure 17 plots the speed-up obtained for each application by using *Phantom*. On the average, multitasking with *Phantom* achieved a speed-up of 2.07, with a maximum of 2.8. These results come specially due to the lightweight implementation of *Phantom*, and as a consequence of being able to, at compile time, generate specific code for each different application.

Table 1: Benchmark applications

Name	Description
client_server	Client-Server implementation of a calculator. Communication through shared memory. 100 servers and 2000 clients.
consumer_producer	Classical consumer producer problem, 100 consumers and 100 producers. Buffer with 1000 entries.
dct	Multitask implementation of 8x8 dct. One task for each point in the result matrix.
deep_stack	Multiple recursive tasks. Tests the cost of recursive function calls in the *Phantom* system.
matrix_mul	Multitask implementation of matrix multiplication. Resulting matrix is 150x150 elements. One task per element in the result.
quick_sort	Multitask implementation of the traditional sorting algorithm.
vm	Multitask simulator for a simple processor.
watch	Time-keeper application, used to test timing behavior of the generated code.

It is important to point out that with embedded applications, being fast is not always the most desired functionality. Instead, many times we are interested only in being accurate and precise, i.e., meeting time constrains, given that embedded systems interact with the physical environment very constantly. In that sense, *watch* is an application of our benchmark that is worth discussing further. *Watch* was implemented as two tasks, a background task that periodically updates physical time information, including hours, minutes, seconds, and milliseconds, to a shared data structure, and a foreground task that uses the shared data structure to perform some action. Further, the foreground task waits for 67 seconds before terminating itself. The overhead of the *Phantom* generated code was sufficiently efficient not to disturb the timing behavior of this particular application.

Table 2: Performance Results

Application	POSIX	Phantom	#Threads	#Ctx.Sw.
client_server	5.14 s	1.84 s	501	110179
consumer_producer	7.23 s	3.54 s	201	2000198
dct	1.02 s	0.49 s	12673	32670
deep_stack	2.05 s	0.84 s	1001	91556
matrix_mul	1.10 s	0.55 s	22501	57518
quick_sort	2.97 s	1.12 s	6640	11019
vm	2.83 s	5.35 s	501	3834
watch	67.01 s	67.00 s	2	1492

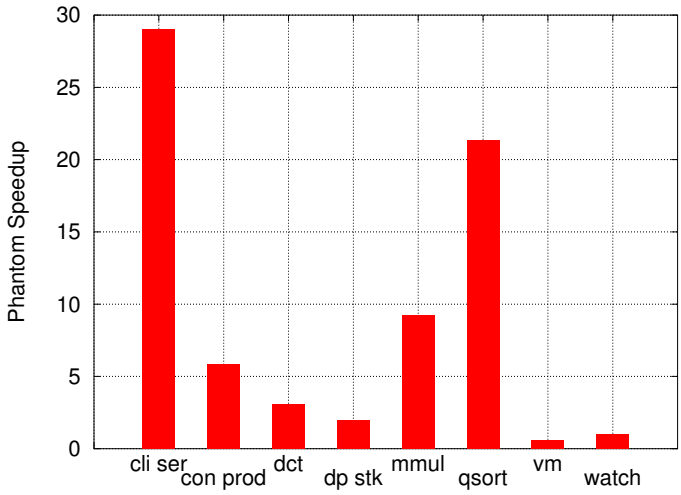

Figure 17: *Phantom* Speedup

6.2 Partitioning Exploration

We used the same algorithms of Table 1 to evaluate the impact of partitioning in the generated code. We applied the partitioning exploration methodology described earlier to obtain Pareto-Optimal sets of partitions for all the applications. Figures 18, 19, 20, and 21 show the resulting Pareto-Optimal partitions for the most interesting cases.

Overall, we observe the trend of increased overhead as latency is reduced (i.e., more partitions are created). Furthermore, by using different partitioning schemes, it is possible to modify latency by as much as two orders of magnitude at the expense of an increase in the overhead by a factor of 120.

Figure 18 shows the Pareto-Optimal partitions for the function `server` in the *client_server* benchmark. In this example, there is a fairly regular behavior. The maximum and the minimum partitions differ by a factor of 3 in latency, and by a factor of 3.5 in performance. The range of latencies is covered reasonably

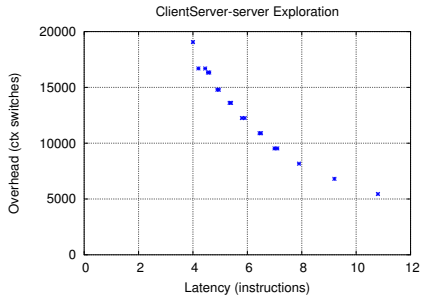

Figure 18: Client Server - server

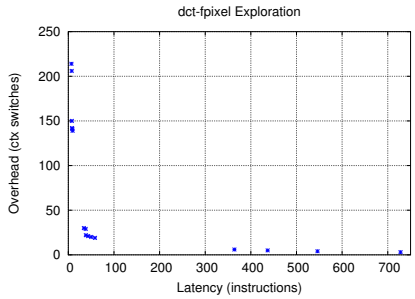

Figure 19: DCT - fpixel

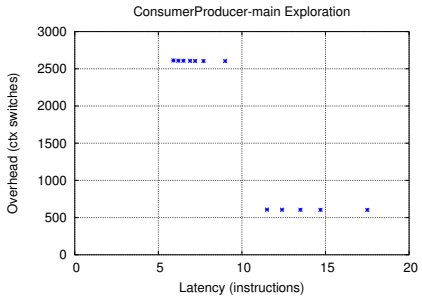

Figure 20: Consumer Producer - main

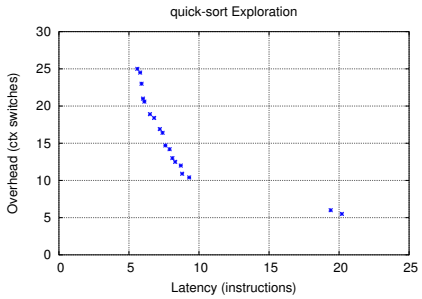

Figure 21: Quick Sort - quick_sort

well by our partitioning methodology.

A completely different picture is shown in Figure 19, the Pareto-Optimal partitions for function `fpixel` in *DCT*. Here, latency ranges from a large 720 instruction delay to a tiny 5 instruction delay on the other extreme. The overhead also changes significantly, from a minimal number of context switches in one case to a large overhead in the other. Moreover, it is possible to detect *islands* of partitions as we break the code in different parts. One can identify at least four separate grouping of the Pareto-Optimal partitions.

Figure 20 shows yet a different scenario as a result of *phantomizing* function `main` of the *consumer_producer* benchmark. Here, the latency reduces very quickly with almost no penalty in performance up to a certain point. Then, for a quite small improvement in latency, there is a huge penalty in performance. After that, latency continues to decrease at almost no cost. In this case, it is easy to estimate that the large cost imposed by one specific partition is caused by breaking a large `for` loop, causing a context switch to happen at every iteration of the loop.

Finally, Figure 21 shows the Pareto-Optimal partitions for the `quick_sort` function, and again we have a different picture. Here, there is a large gap in reducing the average latency initially. Once that barrier is broken, latency can be further reduced, but in the processes, the overhead increases at a steep rate, being

Table 3: Partitioning quick_sort

part number	min latency	max latency	avg latency	std deviation	ctx_sw overhead
0	4	100.7	20.2	32.9	5.5
1	4	87.2	19.4	26.5	6.0
2	4	34.3	9.3	9.3	10.3
13	4	12.3	6.5	3.3	18.9
16	4	11.0	5.9	3.2	23.3
18	4	11.0	5.6	3.4	25.0

almost 5 times higher than the case with the largest partitions.

Table 3 details the minimum, maximum, and average latency; standard deviation; and context switching overhead for some of the partitions explored in the `quick_sort` function. The table shows that, for the larger partitions, the average latency is high, but standard deviation is also high, due to the highly irregular sizes of each cluster, while the overhead due to context switching is minimal. Then, as the clustering methodology explores different partitions, one can see that the latency and the standard deviation are reduced significantly, resulting in a more uniform clustering.

6.3 Phantom Performance

A set of synthetic benchmarks was implemented to evaluate the overhead imposed by the Phantom multi-tasking 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 the UltraSPARC-IIe workstation, 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 4.

Table 4: *Phantom* Multitasking Performance Results

	No optimization (-O0)		With optimization (-O2)	
Metric	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

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 22 and 23 show 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/time for performing a context switch.

We note from Figures 22 and 23 that the overhead of task creation and context switch is *small*, *fairly constant*, and *independent* of the number of runnable tasks in the system. 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

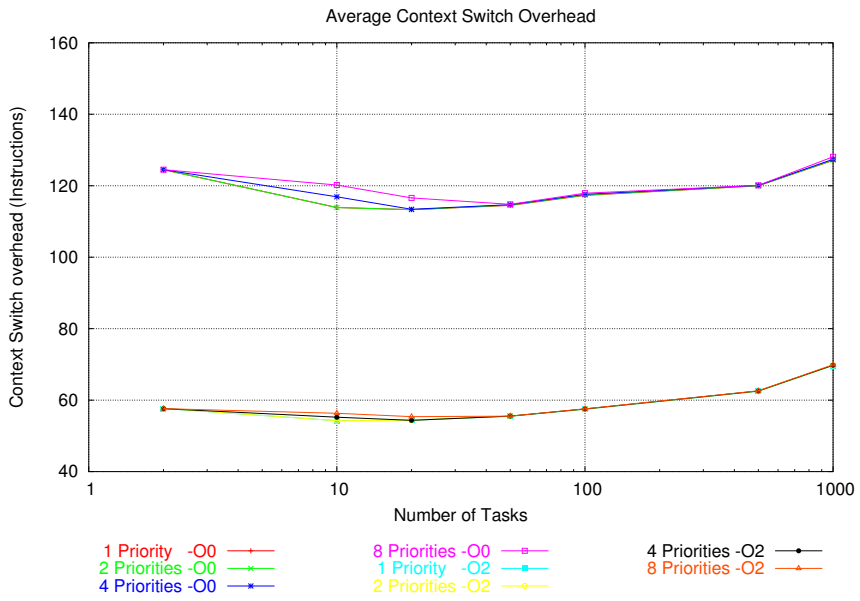

Figure 22: *Phantom* Context Switch Cost in Instructions with Multiple Threads and Multiple Priorities

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.

7 Conclusions

We have presented a scheme for source-to-source translation of a multitasking application written in C extended with POSIX into a single-threaded ANSI C program which can be compiled using a standard C compiler for any target embedded processor. While compiler tool chains are commonly available for any of the large number of customized embedded processors, the same is not true for operating systems, which traditionally provides the primitives for multitasking at the application level. Our source-to-source translator fills this missing OS gap by automatically generating a platform independent C program that encapsulates multitasking support customized for the input application.

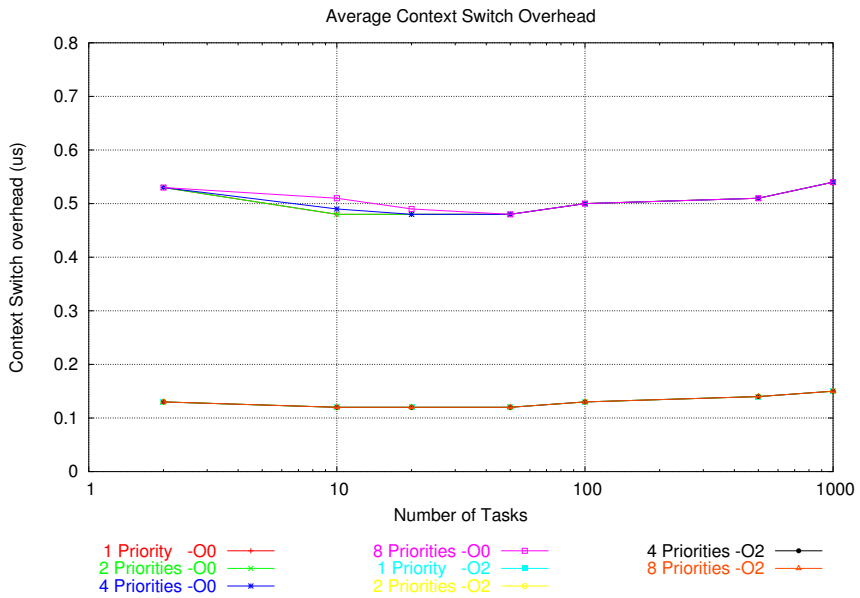

Figure 23: *Phantom* Context Switch Cost in μs with Multiple Threads and Multiple Priorities

8 Acknowledgements

This work was supported by the National Science Foundation award number CCR-0205712 and by CAPES Foundation, Brazil, award number BEX1054/01-5.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] ARM Inc. http://www.arm.com.

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

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

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

[6] 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.

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

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

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

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

[11] Microsoft Corporation. The C# 2.0 Specification, Jul. 2003. Available at http://msdn.microsoft.com/vcsharp.

[12] MIPS Inc. http://www.mips.com.

[13] Phillips Inc. http://www.phillips.com.

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

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

[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] 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.