Code Partitioning for Synthesis of Embedded Applications with Phantom

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

ABSTRACT

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. Serializing compilers have been proposed as an alternative solution, enabling 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. Such serializing compilers work by partitioning each task into blocks of code and synthesizing a scheduler that dynamically switches among these blocks. The quality of the compiled code in terms of multitasking overhead and task latency is highly dependent on the partitioning algorithm. In this work, we give our solution to the partitioning problem in the context of serializing compilers. We show that it is possible to provide the designer with a set of Pareto-Optimal solutions that trade off multitasking overhead for task latency.

Keywords

Dynamic Multitasking, Code Generation, Scheduling, Serializing Compilers, Software Synthesis

1. INTRODUCTION

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 particular, the concurrent programming paradigm is an ideal model of computation for design of embedded systems, which often encompass inherent concurrency.

On the other hand, 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 [13] provides a large family of highly customized application-specific embedded processors, the Xtensa. Likewise, ARM [2] and MIPS [10] provide several derivatives of their respective core processors, in an effort to provide to their customers an application-specific solution.

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 (e.g., in the case of micro-controllers such as the Microchip PIC [9] and the Phillips 8051 [11]) 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.

To fill the gap in realizing a multitasking application targeted at a particular embedded processor, we have proposed *Phantom*. *Phantom* provides a fully automated source-to-source translator, taking a multitasking C program 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.

One important issue in code generation with *Phantom* is that of code partitioning. In order to implement multitasking with a single-threaded ANSI C code, *Phantom* makes use of compile time information, and partitions the code into non-preemptive units of execution, which are call atomic execution blocks (AEBs). Some of these partitions are mandatory to maintain the correct execution of the application, such as those implied by synchronization points. Others are not mandatory, but directly affect response time, latency, and the multitasking overheads. In this paper, we specifically address the partitioning issue in *Phantom*, discussing the impact of partitioning on the final generated code, metrics to measure the quality of different partitions, and algorithms to explore the different possible partitions.

As for related work, we can identify three different approaches that address some of the issues 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.

0-7803-8702-3/04/$20.00 ©2004 IEEE.

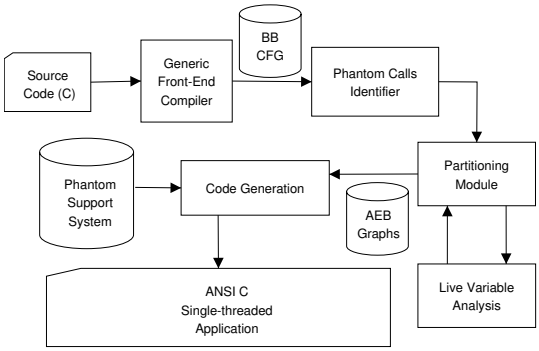

Figure 1: Phantom Compiler Architecture

```
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

To improve on these, solutions like JITs[3] and customized VMs for embedded platforms[15] have been proposed. In the template based OS generation, a custom OS is generated from a generic library of templates [6][7][14]. However, no single generic OS template can be used in the variety of embedded processors available. Finally, the static scheduling techniques [4][5][8] solve the static, a priori known, tasks class of problems, without addressing the dynamic multitasking issues.

Phantom is a new approach in addressing the challenge of multitasking support for embedded applications. We are unaware of any work that addresses the partitioning problem as stated in this work.

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 partitioning issues related to code generation with *Phantom*. In Section 4, we show experimental results. 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, which complies with the standard POSIX interface[12]. 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 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.

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*. Throughout the next sections, we will be referring to our running example shown in Figure 2. 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.

2.2 Preemption and Scheduling

Since the output of *Phantom* is a single-threaded program, there is a need for a context switching mechanism and a basic unit of execution in order to achieve multitasking. 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 3 pictures the CFG transformations for the function `game` of our running example. Figure 3(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 3(b), and subsequently divides the code into a number of AEBs, as shown in Figure 3(c).

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*. Within an AEB graph, each node is implemented as an ANSI C function with no return value. For instance, *aeb_3* implementation is shown in Figure 4 (function `game_aeb3`). The termination of an AEB function transfers the control back to the scheduler (Figure 4, function

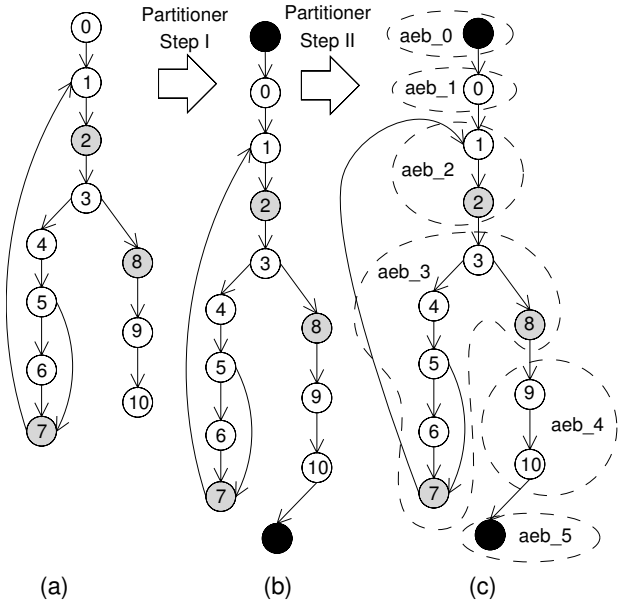

Figure 3: CFG Transformations for Function *game*

```
void game(void *arg, void **ret_val) {
  // allocate and setup frame
  frame = malloc(...);
  frame->arg = arg;
  // save the ret_val in the frame
  frame->ret = ret_val;
  // setup next aeb
  curr_thr->next_aeb = game_aeb1;
  push(curr_thr->frames, frame);
}
void game_aeb3(void) {
  int num;
  // restore locals from frame
  frame = top(current->frames);
  game_t g = frame->g;
  if(!winner) goto bb_4;
  curr_thr->next_aeb = game_aeb4;
  pthread_mutex_unlock(g->unlock);
  goto exit;
bb_4:
  num = rand();
  if(num != g->id) goto bb_7;
  winner = g->id;
bb_7:
  curr_thr->next_aeb = game_aeb2;
  pthread_mutex_unlock(g->unlock);
exit:
}
 void game_aeb5(void) {
   // clean up frame structure
   frame = pop(current->frames);
   free(frame);
 }
```

```
typedef struct {
  int              id;
  status_t      status;
  task_info_t info;
  stack_t       heap;
  join_info_t  join_info;
  aeb_t          next_aeb;
  void           *ret;
}task_t;
task_t *curr_thr;
static queue_t tasks;
static void phantom_scheduler() {
  while(queue_size(&tasks) > 0) {
    curr_thr = queue_pop(&tasks);
    if(curr_thr->next_aeb != 0) {
      curr_thr->next_aeb();
      if(curr_thr->status == RUNNABLE)
        queue_push(&tasks, curr_thr);
    }
    else
      terminate_task(curr_thr->ret);
  }
}
void game_aeb2(void) {
  // restore locals from frame
  frame = top(current->frames);
  game_t g = frame->g;
  if(1){
    curr_thr->next_aeb = game_aeb3;
    pthread_mutex_lock(g->lock);
  }
}
```

Figure 4: Excerpt of the Generated Code

`phantom_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.

It may happen that a function in the original input code is partitioned into more than one AEB, each one of them being implemented as a separate ANSI C function. 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 code. *Phantom* solves this issue by storing the values of local variables of the original C function in a structure inside the task context, emulating the concept of a *function frame*. The frame is initialized in the first AEB of a given function (i.e., *setup*), and cleaned up in the last AEB of the same function (i.e., *cleanup*). These operations are included by the partitioner for every function that needs to be *phantomized*, i.e., divided into AEBs. They are represented by the dark nodes in Figure 3(b). For an example of the generated ANSI C code, refer to Figure 4, functions `game`, for *setup*, and `game_aeb5`, for *cleanup*.

During runtime, there is a need to maintain, among others, a pointer to the next AEB node that is to be executed in the future, called `next_aeb`, in the context information for each task that has been created (Figure 4, structure `task_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.

The embedded scheduler is responsible for selecting and executing the next task, by calling the corresponding AEB function of the task to be executed. The `next_aeb` pointer of a task T_i is used to resume the execution of T_i by making a function call to the function corresponding to the next AEB of T_i. At termination, every AEB updates the `next_aeb` of the currently running task to point 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.

2.3 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 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 partitioned into AEBs when synchronization points are encountered, and a call to a synchronization function is always the last statement in its AEB. 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.

2.4 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.

2.5 Experiments with Phantom

The *Phantom* approach has been successfully applied to a number of applications developed for testing the translation flow. In summary, *Phantom* outperforms standard POSIX implementations, being 2 to 3 times faster in execution time. On the average, multitasking with *Phantom* achieves a speed-up of 2.3, with a maximum of 2.9. In general, multitasking applications synthesized with *Phantom* show a much improved performance (i.e., low operating overhead). The reason is two fold. First, the generated application encompass a highly tuned multitasking framework that meets the application-specific needs. Second, the multitasking infrastructure itself is very compact and efficient, resulting in a much lighter overhead for context switching, task creation, and synchronization.

3. PARTITIONING

As described earlier, 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 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 partitioned. 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*. 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.

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 need to context switch more often, and thus pay a penalty in terms of overhead. In this work, we explore the range of partitioning possibilities.

In the next sections, we define a strategy for clustering and an exploration framework for obtaining a set of Pareto-Optimal partitions.

3.1 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 an ANSI C function in *Phantom*.

Our generic clustering technique is shown in Algorithm 1. 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.

Note that if Algorithm 1 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 Algorithm 1, 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

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

Algorithm 1 The Generic Clustering Algorithm

```
1: Input: cfg, n_0 ∈ cfg the entry point of the CFG
2: Output: clusters c_1, c_2, ..., c_n
3: clust ← {c_i ← b_i | b_i ∈ cfg and reachable from n_0}
4: changed ← 1
5: while changed = 1 do
6:    changed ← 0
7:    for each c_i, c_j ∈ clust do
8:       if every pred. of c_j is in c_i then
9:          c_new ← c_i ∪ c_j
10:         clust ← (clust − c_i − c_j) ∪ {c_new}
11:         changed ← 1
12:      end if
13:   end for
14: end while
```

running Algorithm 1.

Figure 5 shows, step-by-step, the working of the clustering algorithm. Figure 5(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 5(b) shows the addition of the *synch-mark* basic block s. Next, every reachable basic block b_i of the CFG is assigned to cluster c_i as shown in Figure 5(c). Then, by successive iterations, clusters are merged until the final partitioning is reached, as shown in Figure 5(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 tradeoffs is a matter of determining the set of basic blocks that the *time-mark* points to.

3.2 Exploration Framework

Our overall exploration framework is pictured in Figure 6 and works as follows. Initially, the multitasking application is processed by the *Phantom* compiler, as shown in Figure 1, 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 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 Algorithm 2. For a CFG with N basic blocks, Algorithm 2 attempt K random

[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.

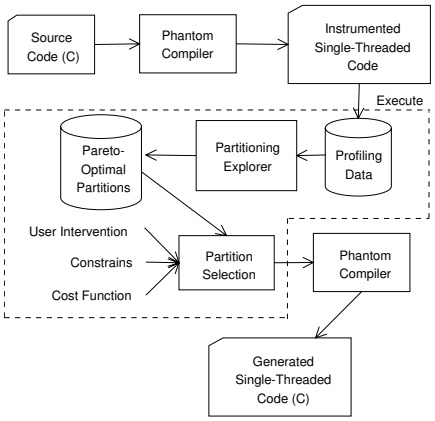

Figure 6: Clustering Exploration Methodology

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.

Algorithm 2 The Search Heuristic

```
1: Input: cfg
2: Input: K {number of tries}
3: Output: cfg_1, cfg_2, ..., cfg_n where n = |cfg|
4: N ← |cfg| {number of basic blocks in cfg}
5: for i = 1 to N do
6:    for j = 1 to K do
7:       pick i random basic blocks in cfg
8:       place an edge from time_mark to basic block i
9:       execute Algorithm 1 and evaluate metrics
10:   end for
11: end for
```

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.

4. EXPERIMENTAL RESULTS

Eight different applications were implemented using the *Phantom* POSIX interface, in order to test the *Phantom* compiler and partitioner. The application benchmarks used in our experiments are described in Table 1.

Our exploration methodology was applied to all the application benchmarks. Figures 7, 8, 9, and 10 show the resulting Pareto-Optimal partitions for the most interest-

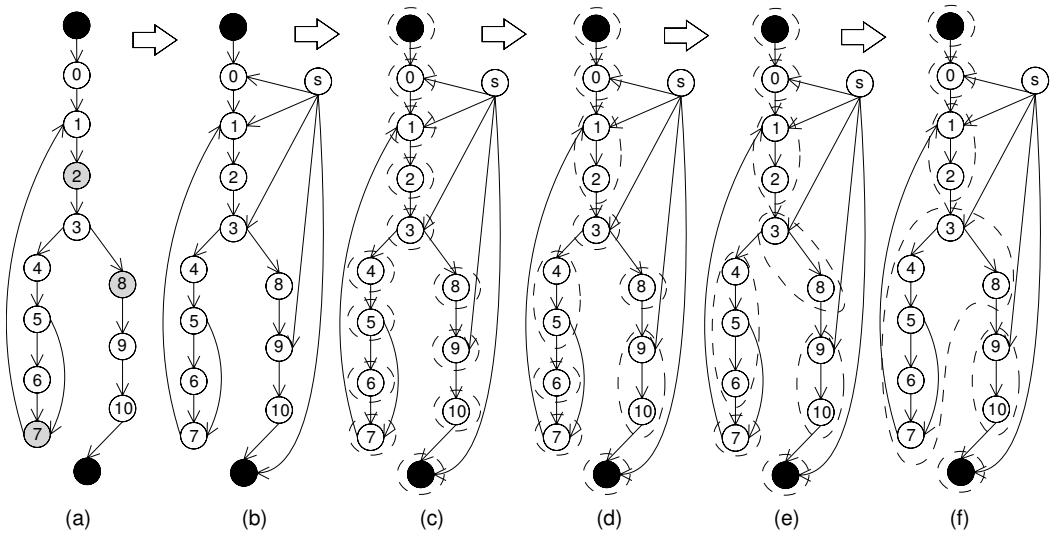

Figure 5: Execution of Clustering Algorithm

Table 1: Application Benchmarks

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.

ing cases. In these figures, each point represents a different partition, showing the latency and context-switch overhead of an especific partitioning solution.The rightmost point in each graph is the cooperative schedule, while the leftmost point is the most responsive scenario, where each AEB has only one basic block.

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 overhead increase by a factor of 120.

Figure 7 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

Table 2: 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

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 well by our partitioning methodology.

A completely different picture is shown in Figure 8, 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.

Figures 9 and 10 show yet different scenario as a result of partitioning. Clearly, the trade off between latency and context switch overhead is variable with different applications.

Table 2 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.

5. CONCLUSIONS

In this work, we have presented our solution to the partitioning problem in the context of serializing compilers. A serializing compiler is a source-to-source translator that takes

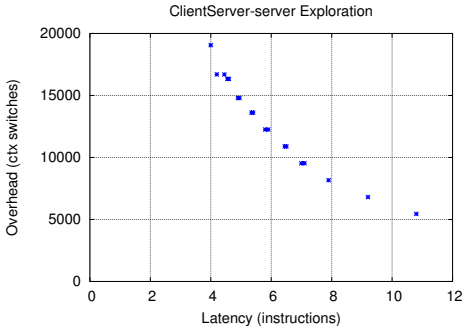

Figure 7: Client Server - server

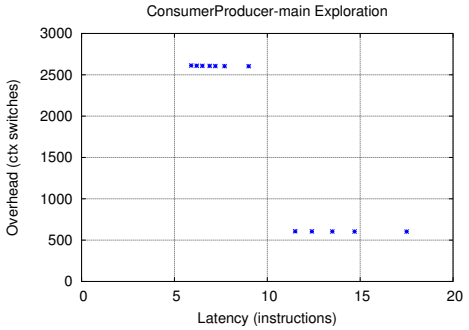

Figure 9: Consumer Producer - main

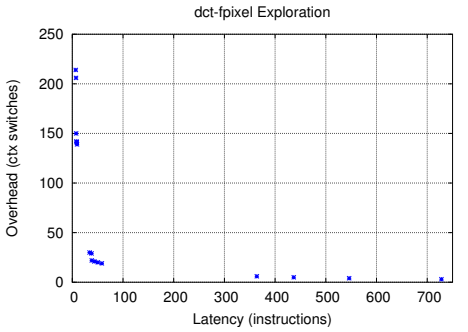

Figure 8: DCT - fpixel

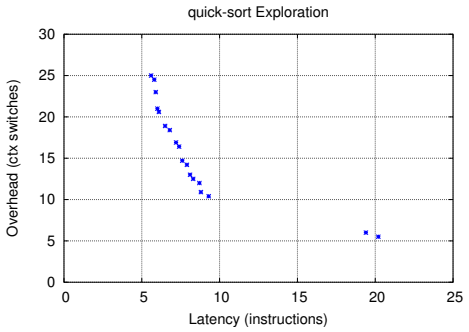

Figure 10: Quick Sort - quick_sort

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 shown that it is possible to provide the designer with a set of Pareto-Optimal solutions that tradeoff multitasking overhead, task latency, and other metrics when serializing compilers are used. Our results show that it is possible to reduce latency (from the cooperative multitasking scheme) by as much as two orders of magnitude at the expense of an increase in the overhead by a factor of up to 120.

Our future direction of research is to investigate approaches where the task latency and multitasking overhead are balanced during execution time. In other words, we are interested in introducing mechanisms for context switching between multiple tasks at arbitrary points (i.e., determined dynamically) during execution.

6. ACKNOWLEDGEMENTS

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

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

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

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

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

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

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

[14] 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.

[15] 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.