A Multi-Core Version of FreeRTOS Verified for Datarace and Deadlock Freedom

Prakash Chandrasekaran, Shibu Kumar K B, Remish L. Minz, Deepak D'Souza, Lomesh Meshram
Department of Computer Science & Automation
Indian Institute of Science
Bangalore, India.
Email: prakash.chandrasekaran.IN@ieee.org,{shibu.kumar,remish.minz,deepakd,meshram.lomesh}@csa.iisc.ernet.in

***Abstract*— We present the design of a multicore version of FreeRTOS, a popular open source real-time operating system for embedded applications. We generalize the scheduling policy of FreeRTOS to schedule the n highest-priority longest-waiting tasks, for an n-core processsor. We use a locking mechanism that provides maximum decoupling between tasks, while ensuring mutually exclusive access to kernel data-structures. We provide an implementation of the portable part of FreeRTOS (written in C) and provide the device specific implementation of the locking mechanism for Intel and ARM Cortex multicore processors. We model the locking mechanism and the locking protocol used by the API's in the Spin model-checking tool and verify that the design is free from dataraces and deadlocks. Finally, we extend the existing FreeRTOS Windows simulator to simulate our multicore version of FreeRTOS, and evaluate its performance on some demo applications.**

I. Introduction

Embedded applications are increasingly being run on multicore processors, as they have lower power dissipation and smaller form factors, leading to smaller battery and enclosure requirements [1]. Many of the general purpose operatings systems commonly used by embedded applications (like the Linux-based Android and Ubuntu operating systems) have the capability to exploit multicore architectures by running different processes on the available cores. However, the smaller, microcontroller based Real-Time Operating Systems (RTOS's), like FreeRTOS, despite being ported on several multicore architectures, typically run only on a *single* core. A truly "multicore" version of such an RTOS that enables tasks that are ready to be scheduled parallely on available cores, would give embedded applications in this class a better chance of meeting their real-time requirements.

In this paper we describe an effort to design and implement such a multicore version of FreeRTOS for symmetric multiprocessors (SMP's). FreeRTOS [2] is one of the 3 most used operating systems for embedded applications [3] and by far the market leader in the class of microcontroller RTOS's, with more than a 100,000 downloads a year. It is modularly designed, with a portable layer written in C, and a port-specific layer for each compiler-processor pair written mostly in assembly. To begin with, we generalize the scheduling policy of FreeRTOS, to schedule the top n highest-priority ready tasks (using time spent waiting in the ready queue to break ties) on an n-core symmetric multiprocessor. To do this we redesign and modify the portable layer of FreeRTOS, to add new data structures like a "running queue". We use a bitvector of locks to provide mutually exclusive access to the kernel data-structures, in a way that maximizes the decoupling between tasks. We also give an implementation of the port-specific layer for the ARM Cortex A9 quadcore processor, which uses the Load-Link and Store-Conditional (LL/SC) instructions of the processor to implement locks. The implementation of the portable layer involves only 1,400 new lines of code, ensuring that the RTOS continues to have a small code footprint.

This work was partially funded by the Robert Bosch Center at IISc, Bangalore

An important issue in such a design that uses fine-grained locking is the problem of inadvertantly introducing data-races and deadlocks in the kernel API's. We systematically model the API's in the Spin model-checker [4] by running the Modex model extraction tool [5] on our C source code, and prove that every multicore-FreeRTOS application is free from data-races and deadlocks that might arise due to the API's. In a naive modelling in Spin one would have one process for each core which non-deterministically runs one of the API's. The problem with this approach is that we would have verified our implementation only for specific values of the number of cores n (and not for arbitrary n). Secondly the model-checker does not scale well with n: it is unable to complete an exhaustive check beyond small values of n like 2 or 3. We get around this problem by first showing that it is sufficient to consider only 2 cores, in that any data-race that manifests with $n > 2$ would also occur with $n = 2$. We then verify our model for $n = 2$. The verification exercise uncovered a couple of potential races due to improper locking and helped us fix these issues in our implementation.

Finally, we modify the Windows Simulator for FreeRTOS to obtain a simulator for multicore-FreeRTOS, which we use to evaluate our implementation of multicore-FreeRTOS on some demo applications. Our experiments on a quad-core machine, confirm the expected 4x increase in task time, and reduced API time, using multicore-FreeRTOS with only a slight increase in scheduler overhead.

The rest of this paper is structured as follows. In the next section we introduce FreeRTOS and describe our objectives for the design of multicore-FreeRTOS. In Secs. III, and IV we present the design and implementation of the portable layer of multicore-FreeRTOS. In Sec. V we describe the verification of data-race and deadlock freedom of our implementation. Secs. VI and VII describe the Windows simulator and our evaluation. Finally, we close with a section on related work.

```
int main(void) {
  xTaskCreate(foo, "A1", 1000, 1,...);
  xTaskCreate(foo, "B1", 1000, 1,...);
  xTaskCreate(bar, "C2", 1000, 2,...);
  vTaskStartScheduler();
}

void foo(void* params) {
  for(;;) {
    // do low priority processing
  }
}

void bar(void* params) {
  for(;;) {
    // do high priority processing
    vTaskDelay(2);
  }
}
```

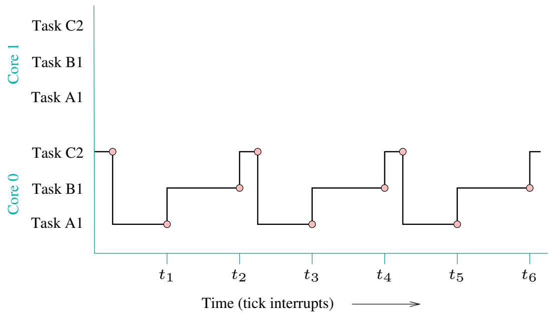

Fig. 1. An example FreeRTOS application and its timing diagram on a 2-core processor.

II. Background & Objectives

In this section we describe the key aspects of FreeRTOS and spell out the main requirements for the design of the multicore version of FreeRTOS.

A. About FreeRTOS

FreeRTOS [2] is a real-time kernel meant for use in embedded applications for microcontrollers with small to mid-sized memory. It allows an application running on top of it to organise itself into multiple independent tasks (or threads) that will be executed according to a priority-based preemptive scheduling policy. It is implemented as a set of Application Programmer Interface (API) functions written in C, and is compiled along with the application code, and loaded into memory. These API's provide the programmer ways to create and schedule tasks, communicate between tasks (via message queues, semaphores, etc), and carry out time-constrained blocking of tasks. The scheduler runs as part of the code loaded as Interrupt Service Routines (ISR's) for the software interrupt and the timer IRQ interrupt. This is done by a direction to the compiler, contained in the FreeRTOS code.

Fig. 1 shows a simple application that uses FreeRTOS. The application creates three tasks "A1", "B1", and "C2", with priorities 1, 1, and 2 respectively, and then starts the FreeRTOS scheduler. A higher number indicates a higher priority, and we follow a naming convention that indicates the task's priority in its name. The graph shows the execution times of the tasks as scheduled by FreeRTOS, in accordance with its policy of time-slicing between tasks of the top ready priority. We look at the details of the scheduling in the following section.

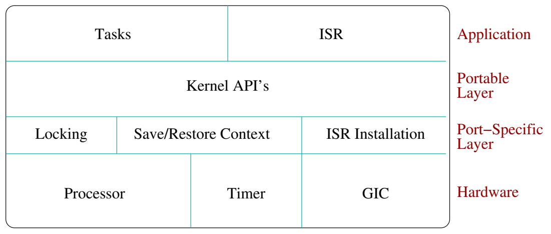

Fig. 2. Architecture of FreeRTOS

B. Inside the FreeRTOS Scheduler

We now take a look under the hood to get a closer look at what exactly happens when our example application executes. Execution begins with first instruction in `main` which happens to be a call to the `xTaskCreate` API. This API, allocates space in the heap for the task stack and its Task Control Block (TCB). It then creates and initializes the various task queues that the kernel maintains, including the ready queue which is an array of FIFO queues, one for each priority; and the delayed queue. It finally adds `A1` to the ready queue and returns. Next, `main` calls `xTaskCreate` for `B1` and `C2`, which adds them to the ready queue. Then, the call to `vTaskStartScheduler` API creates the "idle" task with priority 0, and adds it to the ready queue. It also sets the timer tick interrupt to occur at the required frequency. Finally, it does a context-switch to the highest priority ready task . In our example, this means that `C2` will now begin execution.

When `C2` begins execution it completes its processing and makes a call to the `vTaskDelay` API. This API call will add `C2` to the delayed queue,with a time-to-awake value equal to the current tick count plus 2. It then does a `yield`, causing the scheduler to schedule the (longest waiting) highest priority ready task, which in this case is `A1` .

`A1` now executes it's long-running processing, till it is preempted by the tick interrupt. Then, the scheduler increases the tick count, and checks if any of the delayed tasks have a time-to-awake value that equals the current tick count. There are none, and the scheduler proceeds to find the next longest-waiting highest-priority ready task, which happens to be `B1`. When the next timer interrupt takes place, the scheduler finds that `C2`'s time-to-awake equals the current tick count, and moves it to the ready queue. Since there is now a higher priority ready task, `B1` is moved to ready queue and `C2` is resumed. The execution continues in this way, ad infinitum.

C. FreeRTOS Source Code Structure

The FreeRTOS source is structured in a modular fashion. It has a generic part which contains compiler/processor-independent code, most of it in 3 C files `task.c`, `queue.c`, and `list.c`. The port-specific part (a "port" is a particular compiler/processor pair) is present in a separate directory associated with each compiler/processor pair, and is written in C and assembly. Fig. 2 shows the layered architecture of FreeRTOS schematically.

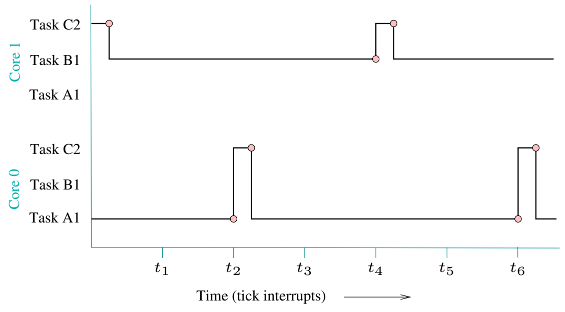

Fig. 3. Execution of example application with multicore-FreeRTOS on a 2 core processor.

D. Objectives for multicore-FreeRTOS

While FreeRTOS has been ported to several multicore architectures including the quadcore ARM Cortex A9, it does not make use of the multiple cores, but simply runs on a single core on these processors. For reasons that we have outlined earlier in Sec. I, we would like to create a version of FreeRTOS that runs on multicore architectures in a way that utilizes the cores available to it. Based on discussions with the developers of FreeRTOS [6], we have come up with the following requirements for a multicore version of FreeRTOS.

a) Scheduling policy: A basic requirement of multicore FreeRTOS is that it should *generalize* the scheduling policy of FreeRTOS, in the sense that if our multicore scheduling policy is parameterised by the number n of cores available, then when $n = 1$ we should obtain a scheduling policy similar to the existing FreeRTOS.

We could generalize the scheduling policy of FreeRTOS in two natural ways:

1) Schedule (at most) n of the longest-waiting ready tasks of the *top ready priority*. Thus if `A1, B1` and `C2` were the ready tasks, we would schedule only `C2` on a 4 core machine.
2) Schedule the top n longest-waiting highest-priority tasks. Thus in the above scenario we would schedule all 3 tasks.

The first approach ensures that legacy applications that rely on priority for ensuring mutual exclusion will continue to work, as a lower priority task will never interleave with the execution of a higher priority task. But, this policy will lead to under-utilization of the cores, when there are fewer than n tasks of the highest priority.

Whereas the later, optimizes core-utilization and preserves the priority scheduling of FreeRTOS, at the cost some legacy applications failing. Since, the objective of this effort is to better utilize the multiple cores, we chose to go with the later policy.

Thus, with this scheduling policy, the example application of Fig. 1 would run on a 2-core processor as shown in Fig. 3.

b) Decoupling tasks: It is desirable to have as much decoupling between tasks as possible, so that tasks can proceed independently of each other whenever possible. Contention for shared resources, leads to a queuing effect that directly impacts the responsiveness of a system [7]. This requires that access to kernel data-structures be controlled by fine-grained (rather than coarse-grained) locking protocols.

c) Data-race and deadlock safety: A serious problem that typically arises with locking at a fine-grained level is the possibility of data-races while tasks access shared kernel data-structures through the API's provided by the kernel. Also, the locking protocols should be free from deadlocks. In particular, we should minimize "hold-and-wait" situations (where an API tries to acquire locks in a nested, sequential manner), and rule out any *circular* hold-and-wait conditions.

III. Design of Multicore FreeRTOS

In this section we describe the design of our multicore-FreeRTOS. As mentioned in the previous section, the scheduling policy we implement is to schedule the n longest waiting, highest priority ready tasks, on a system with n-cores. More precisely, if we order the ready tasks in decreasing order of priority, and within the same priority in decreasing order of time spent waiting in the ready queue (or, equivalently, increasing order of time of arrival in the ready queue), then we want to schedule the first n tasks in this ordering.

A. Data Structures

We make use of the existing data structures in FreeRTOS as far as possible. FreeRTOS uses the following key data structures to maintain the task information:

- *ready list*: an array of FIFO lists, one for each priority, that contains the ready tasks of a given priority.
- *delayed list*: a priority queue of tasks that have been delayed, along with their time-to-awake, sorted in decreasing order of the time-to-awake. FreeRTOS also maintains an *overflow delayed list*, which contains tasks whose time-to-awake is beyond the max tick count.
- *suspended list* and *deleted list*: list of tasks that have been suspended and deleted respectively.
- *pending ready list*: tasks that enter the ready state when the scheduler is suspended are put here rather than the *ready list*.
- *event lists*: priority queue of tasks waiting on an event, like the arrival of a message in a message queue.
- `pxCurrentTCB`: a pointer to the currently executing task.

In multicore-FreeRTOS, we need some data-structures in addition to the above lists. Since there is more than one running task, we maintain a *running list* which contains the running tasks. This list is maintained in *increasing* order of priority and, within this, in *decreasing* amount of running time.

We do not need the *pending ready list*, as our scheduler is never suspended. However, we use a *local pending ready* on each core to which tasks that enter the ready state, and are not immediately runnable, are added. A newly enabled task with a higher priority than a running task, can also get added to the

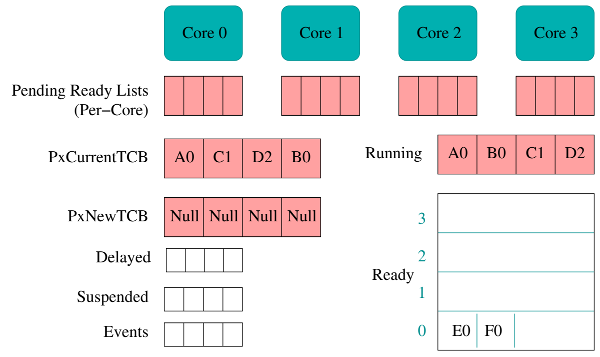

Fig. 4. Data structures used in multicore-FreeRTOS. Shaded ones are newly added.

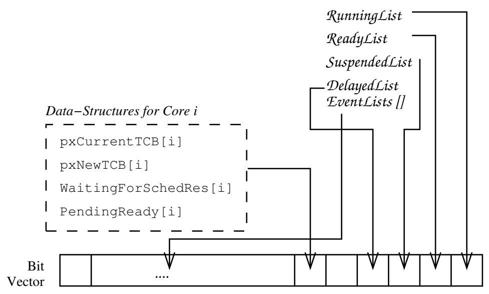

Fig. 5. Lock bits corresponding to data-structures in multicore-FreeRTOS.

local pending ready list, when locks needed to schedule the new task are not immediately available.

Instead of a single `pxCurrentTCB` pointer, we use an array `pxCurrentTCB[]` of size n, to store our current TCB's. Additionally, to facilitate context switching, we introduce a same sized array of TCB's, `pxNewTCB[]`, which holds the pointers to tasks to run next in the corresponding cores. Fig. 4 shows the main data structures in multicore-FreeRTOS, with the shaded ones being the new data structures added.

B. Decoupling using fine-grained locks

The primary challenge in a concurrent execution environment is ensuring data races don't happen as the tasks running in different cores compete for system resources. The next is to ensure that the use of mutual exclusion doesn't sequentialize the tasks. In FreeRTOS, mutual exclusion is obtained by either masking the interrupts or suspending the scheduler while a task is in its critical section. Note that in the first approach, it is possible that some vital interrupts, including the tick, can be lost. In the second method, if a tick comes when a task is in its critical section, it is not processed immediately, but a variable `uxPendedTicks` is incremented and once the critical section is over, the missed ticks are processed.

We implement a bit-vector based locking mechanism, to control access to the kernel data structures. The kernel data structures like the *running list*, *ready list*, *delayed list*, etc., are assigned a designated bit in the vector, as shown in Fig. 5. Other core-specific resources like *pending ready list*, `pxCurrentTCB`, and `pxNewTCB` are protected by a lock on the core-id. The locks are obtained and released by atomically modifying this bit-vector. The use of a bit-vector of word-length allows for the whole vector to be addressed in a single CPU load/store instruction. This then allows to examine the vector and acquire all the necessary locks at one instant, eliminating a hold-and-wait scenario.

While there exist complex schemes for implementing an atomic modification in a multicore environment without relying on support from the micro-controller [7], they are prone to errors and difficult to verify. Also, all widely-used modern SMP platforms offer their own mechanisms, at the micro-controller level to perform atomic mutations of system memory. To keep our implementation simple and verifiable, we rely on atomic modification of memory locations using atomic lock free instructions provided by the micro-controller. Load-Link and Store-Conditional (LL/SC) and Compare-And-Swap (CAS) are two techniques by which lock-free atomic read-modify-write operations can be performed in a concurrent execution environment. These are used by ARM and Intel, respectively, to provide for mutual exclusion in their processor architectures. In LL/SC, the load and store operations are linked, and any intervening store operation would cause the linked store-conditional to fail.

In CAS, the store succeeds only if the current value matches the previous read value. However, this causes the ABA [8] problem, where multiple intervening updates to the location that restore the initial value will allow the CAS to succeed. This is commonly addressed by using a double length CAS (DCAS), where the second half holds a mutation counter.

We are working towards implementing our multicore-FreeRTOS on a ARM Cortex A9 based system, as described in Sec. IV-C. Hence, we chose to use the LL/SC approach to implement atomic read-modify-write operations in our Promela model, as well, for verification.

1) Minimizing lock-contention across API's: We minimize lock contention in each of the API, by maximizing use of localized (i.e. core-specific) data structures, and synchronizing the data only at specific times, like when the *FullReschedule* happens.

The global data structures that are accessed most in FreeRTOS are the *ready list* and the *running list*. When a running task is blocked, the *ready list* is accessed to select the next high-priority task, and the *delayed list* or *suspended list* is accessed to insert the blocked task, and the *running list* is accessed to insert the newly runnable task and remove the blocked task. When a task is made runnable, the *running list* is accessed to see if the new task has higher priority than any of runnning task. If so, the new task is inserted into *running list* and the displaced task is put into the *ready list*. Otherwise, the new task is inserted into the *ready list*. Here, we introduce a *pending ready list*, local to each core for inserting any tasks that would be normally inserted into the *ready list*.

2) Minimizing contention with the scheduler: Although we would like to acquire all the locks at one instant in all our methods, the scheduling algorithm cannot afford to do so, as it would mean that the tasks running on all other cores will be denied resources irrespective of whether the scheduler is using it or not. Hence, we take only the essential set of locks in each section of the scheduler. Now, to achieve higher performance, it is essential that the scheduler is never unreasonably delayed or kept waiting for resources. Note that in the standard FreeRTOS, this requirement never arises because, by design no other task is running when the scheduler is.

We enhance the bit-vector with an additional *scheduler-waiting* bit to indicate that the scheduler is either waiting-for or using the task lists. If any of the APIs need to acquire a lock on any of the task lists, and this bit is set, then the API will flag the current core as waiting for scheduler resources, and block until the flag is unset by the scheduler, before proceeding for the lock. This prevents the task control APIs from competing with the scheduler, and also allows them to be swapped out if required by the scheduling policy. At the end of each scheduling run, the scheduler simply clears the waiting for scheduler flag for all the cores in the system. Note that the scheduler need not read the state of these flags.

C. Invoking the Scheduler

In FreeRTOS, the scheduler typically runs as part of each API call, as well as part of the the ISR for the tick interrupt. In multicore-FreeRTOS, scheduling happens in two different methods, *Schedule* and *FullReschedule*. The former is used to schedule one newly enabled task displacing the lowest priority running task. The latter is the scheduling routine that schedules the next round of runnable tasks on all the cores.

- *Schedule* is invoked, whenever a task is unblocked by the availability of the resource it was blocked on, or by an explicit task resume. Also, when a running task is suspended or deleted, we invoke *Schedule* to run the next top-priority longest waiting ready task.
- *FullReschedule* is invoked by the tick ISR which resides on core 0. In the case that the task executing in core 0 has locked core 0, the ISR will increment `uxPendedTicks` by 1 and exit. The *FullReschedule* will be invoked either by the tick ISR next time (if core 0 doesn't hold its core lock) or by an API call, whichever happens first. This approach makes sure that a tick is never lost.

In a multicore environment, we have the option to either handle the timer interrupt in a designated core, or by any of the cores. Note that in the latter strategy which core gets to run the scheduler is not determinable , as only one core gets to service the interrupt.

Since the scheduling algorithm can take a significant amount of CPU cycles to execute, it makes sense to execute the scheduler on the core running the task with the least priority. This ensures that maximum CPU cycles are available for the other higher priority tasks.

Note that this allows for a task whose priority is not the least of those running, to be scheduled on core 0, if it is made runnable by an API call. However, at the next scheduling cycle we rectify this by swapping the task onto another core, if it still remains the task with not the least priority. This is an optimization feature, that doesn't affect the priority based scheduling principles, and at the same time reduces overhead caused by context switching.

IV. Implementation

In this section we describe our implementation of the portable layer of multicore-FreeRTOS, and also a port-specific layer for ARM Cortex A9. We classify the changes to the portable layer as changes to API's and changes to the scheduler.

A. Changes to API's

a) Initialization: The `vTaskStartScheduler` API call must initialize the new data-structures, like the running list and the `pxCurrentTCB` array.

b) Mutual exclusion: In our API's before accessing the kernel data-structures we acquire locks on the appropriate lock bits, via calls to `acquireLock` and `releaseLock` functions provided by the port-specific layer. Whenever we accquire a lock, we lock the current core as well to indicate to other APIs and ISRs that the task executing on the core is holding locks on system resources and should not be pre-empted.

c) Invoking the scheduler: Depending on the value of `uxPendedTicks` an API must call either the *Schedule* or *fullReSchedule* function.

d) Pending ready: If a newly created or unblocked task is not immediately runnable (i.e. its priority is less than that of the lowest proirity running task) then the task is added to the *local pending ready* list for the current core.

e) New Helper Method: We introduce a new helper method called `WaitOnSchedulerResources` that checks if the *fullReschedule* method is waiting for or using the system resources, and if so, blocks until the resources are freed by the scheduler.

B. Changes to scheduler

The scheduler finds the next n tasks to schedule by checking the running and ready lists, and allocates them to appropriate cores by setting the `pxNewTCB` variables on each core. It takes care to avoid unnecessary task migration between cores in this process. It must then send an inter-process interrupt (IPI) to the cores that need to do a context-switch. As in the case of API's the scheduler also implements mutual exclusion using acquire and release of locks. The scheduler must also take care of any missed IPI's for context-switch by resending the IPI.

C. Implementation for ARM Cortex A9

The main components of the port-specific layer for multicore-FreeRTOS are the implementation of routines that save and restore task context, register ISR's for the tick interrupt on core-0 and IPI's on all cores, and the locking implementation. We implement the `acquireLock` and `releaseLock` routines using the LDREX/STREX instructions of the native processor.

V. Verifying data-race and deadlock freedom

Our aim in this section is to prove that our design of multicore-FreeRTOS is free from data-races on the kernel data structures, as well as from deadlocks. By a *data-race* in a

```
 1. int x;

 2. void thread1() {
 3.   int tmp;
 4.   lock(1);
 5.   x = x + 2;
 6.   unlock(1);
 7.   tmp = x;
 8. }

 9. void thread2() {
10.   lock(1);
11.   x = x - 1;
12.   unlock(1);
13. }
```

(a)

```
 1. int cx = 0;

 2. proctype thread1() {
 3.   lock(1);
 4.   cx++; cx--;
 5.   unlock(1);
 6.   cx++; cx--;
 7. }

 8. proctype thread2() {
 9.   lock(1);
10.   cx++; cx--;
11.   unlock(1);
12. }

13. init {
14. run thread1(); run thread2();
15. }

16. ltl norace { [](cx < 2) }
```

(b)

Fig. 6. Example illustrating datarace modelling in Spin. Part (a) shows a C program and (b) its abstraction in Promela.

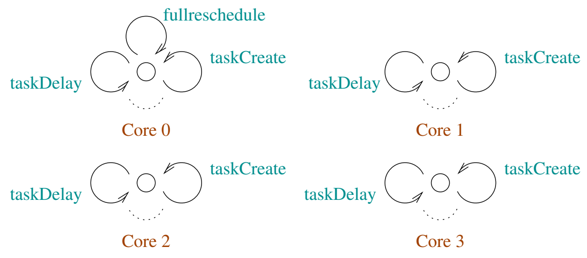

Fig. 7. Schematic representation of Promela model M_4.

multi-threaded program (or model) P we mean an execution of P in which we have two consecutive accesses (either reads or writes, with at least one of them a write) to the same memory location, without an intervening acquire/release of a lock. A *deadlock* in P is a state that is reached along an execution of P, in which a subset of processes X are all blocked waiting for resources that are held by other processes in the set X.

We prove that any application A running with multicore-FreeRTOS on an n-core processor (for any n) will never have a data-race due to the kernel APIs it calls; and also that it never deadlocks due to the kernel APIs it calls. We do this by constructing a Promela/Spin [4] model M_n that overapproximates the behaviour of any application running with multicore-FreeRTOS on an n-core machine, and proving that M_n is free from data-races and deadlocks. We do this using the Spin model-checker and the Modex model-extraction tool [5]. We describe these steps in more detail below.

A. Modelling data-races in Spin

We begin with a simple example that illustrates how we model data-races in Promela, the modelling language of Spin. Fig. 6(a) shows a C program with 2 threads that access a shared variable `x`. In the Promela model shown alongside in Fig. 6(b) we model each thread as a `proctype`, and capture each access of `x` (both reads and writes) as an increment followed by a decrement of a corresponding variable `cx`, which is initialized to 0. The `lock` and `unlock` statements are captured as is, and are assumed to be implemented in Promela as (blocking)

```
#define mReady (1<<0)
#define mRun (1<<1)
...

int lockVector = 0;
int cCore[NUM_CORES]; int cReady = 0; int cRun = 0;


inline taskDelay(coreid) {
  if
  :: true -> acquireLock(mCore(coreid)|mRun|mDelay);
             cCore[coreid]++; cCore[coreid]--;
             cRun++; cRun--;
             cDelay++; cDelay--;
             releaseLock(mCore(coreid)|mRun|mDelay);

  :: true -> acquireLock(mCore(coreid)|mReady);
             cCore[coreid]++; cCore[coreid]--;
             cReady++; cReady--;
             releaseLock(mCore(coreid)|mReady);
  fi;
}

proctype taskproc(byte coreid) {
    do
        ...
    :: true -> taskDelay(coreid);
        ...
    od;
}

init {  run taskproc(0); run taskproc(1); }
```

Fig. 8. Excerpt from the Promela model M_2

atomic read-and-update's, as shown in Fig. 9. Each process is then run parallelly and Spin checks for violations of the LTL assertion "`norace`" which says that it is always the case that the value of `cx` is less than 2. The Spin model-checker duely reports a violation of this property when `thread1` increments `cx` in line 6 (corresponding to the read of `x` in line 7 of the C program), and `thread2` also increments `cx` in line 10 (corresponding to the write to `x` in line 11).

B. Promela model M_n

Let us consider a multicore-FreeRTOS application A running on an n-core machine. Each core could be running tasks that in turn make calls to the multicore-FreeRTOS API's. In addition, core 0 could sometimes run the ISR for the tick interrupt, leading to a call to `fullreschedule`. This view of an application is visualized in Fig. 7. The figure also represents the Promela model M_n when $n = 4$.

For each API function, we construct a Promela function that abstracts away the details of each API, and keeps track of only the locking behaviour (acquiring and releasing locks) and the accesses (reads or writes) to the kernel data-structures. Locks are modelled in a similar way to the multicore-FreeRTOS implementation as a vector of bits called `lockVector`. The `acquireLock` and `releaseLock` routines are modelled as atomic read-and-write accesses to the lock vector, as shown in Fig. 9.

The accesses to kernel data-structures are modelled by incrementing the value of a variable associated with the data-structure (for example `cRun` is the variable associated with the running queue) before the access and decrementing it soon after the access. Fig. 8 shows the translation of the `vtaskDelay` API to its Promela function.

```
inline acquireLock(mask) {
  atomic {
    if
    :: ((lockVector & (mask)) == 0) ->
        lockVector = (lockVector | (mask));
    fi;
  }
}

inline releaseLock(mask) {
  atomic{ lockVector = lockVector & (~(mask)); }
}
```

Fig. 9. Locking routines in Promela model

To obtain the Promela model M_n, we run a process on each core that repeatedly chooses an API (non-deterministically) and runs it. Fig. 7 shows a schematic of the Promela model M_4. Note that the `fullreschedule` function runs only on Core 0. The model M_n overapproximates an application A, in that any execution of A on an n-core machine has a corresponding execution of M_n in which the sequence of acquires/releases and accesses to kernel data-structures is exactly the same. It follows that if for each n, M_n has no data-races or deadlocks, then every multicore-FreeRTOS application is also data-race and deadlock free. In the next couple of subsections we focus on showing that M_n is data-race and deadlock free for each n.

C. Checking M_n for data-races

We note that there is a data-race in M_n iff there is an execution of M_n in which one of the counter variables has a value of 2 or more. This can be phrased as an LTL property in Promela as follows (where "`[]`" means "globally").

```
[]((cReady < 2) && (cRun < 2) && (cCore[0] < 2) && ...)
```

We would like to argue that for each $n \geq 1$, M_n has no data-races. We note here that even if one were to check M_n for specific values of n, scalability is an issue. For $n = 4$ the state vector has 168 bytes and Spin runs out of memory while doing an exhaustive search of the state-space on a machine with 14GB of available memory.

We get around this problem by observing that—due to certain properties of our particular model—it is sufficient to check for data-races with $n = 2$. We note that the model M_n satisfies the following properties:

(C1) In any execution, an `acquireLock` and its matching `releaseLock` occur in the *same* process.

(C2) The only blocking statements (i.e. statements whose enabledness depends on the state of global variables) in a process are the `acquireLock` statements. This property would not be true if we had, for example, an access to `cReady` that depended on the value of a shared variable.

Lemma 1: If there is a data-race in M_n for some $n > 2$ then there is one in M_2 as well.

Proof: Let $n > 2$, and let π be an execution of M_n in which processes 1 and 2 (without loss of generality) are involved in a data-race. Then we claim that the projection π' of π to the processes 1 and 2, is also an execution of M_n (and hence of M_2). By property (C2), the only way π' may not be a valid execution of M_n is if there is an acquire of l in say process 1, which is not enabled in π'. Suppose this acquire statement was the i-th statement in π. Then since π was a valid execution, all preceding acquires of l must have been released. Moreover, by the property (C1), each of these acquires have the matching release in the same process. It follows that after projecting away the statements in processes other than 1 and 2, the acquire statement in process 1 is still enabled. This is pictured in Fig. 10, which shows an execution π, and the bold lines representing the projection π' of π to processes 1 and 2. ■

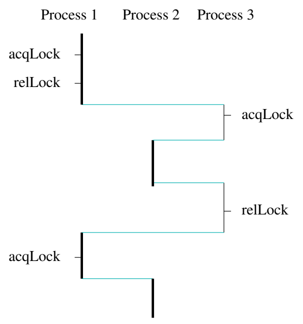

Fig. 10. Projecting an execution to 2 processes.

Spin quickly verifies that M_2 has no data-races. Hence we can conclude that our design of multicore-FreeRTOS is data-race free. The verification exercise helped us to debug our implementation of multicore-FreeRTOS as it pointed out a couple of potential data-races in earlier models. One of these was in the `TaskCreate` API, where it writes to the shared variable `uxNumberofTasks` (representing the number of tasks created so far) without taking an appropriate lock. Some other were benign races, like when the `WaitOnSchedulerResources` flag is read without taking an appropriate lock.

D. Checking M_n for deadlocks

We now argue that M_n is deadlock-free for all n. Let $n \geq 2$ and consider M_n. As is well known, for any set of processes to have a deadlock, there must be a subset of processes that have a circular hold-and-wait condition [9]. In the case of M_n this means a state in an execution in which, for example, 3 processes P_1, P_2 and P_3 which have acquired a set of locks X_1, X_2, and X_3 respectively, and are each trying to acquire a set of locks Y_1, Y_2, and Y_3, such that $Y_1 \cap X_2 \neq \emptyset$, $Y_2 \cap X_3 \neq \emptyset$, and $Y_3 \cap X_1 \neq \emptyset$.

We argue that such a condition cannot arise in M_n for the following reasons. The only hold-and-wait condition that can arise is in the `fullreschedule` function, which runs in process 0 only. We checked this in Spin, by analysing *each* API separately. We added a boolean `wait` flag that is set in the beginning of the `acquireLock` routine and unset when it exits. We then checked the LTL property

```
[](!((lockVector != 0) && wait).
```

Spin initially reported several instances of the hold-and-wait condition. Some of these were false postives due to the abstracting away of `if` conditions. Others were genuine problems in APIs like `vTaskDelay` and `vTaskSuspend` that we had to fix to remove hold-and-wait situations in these APIs. We also found, and fixed, a spurious unlock in `vTaskSuspend` API that unlocked `uxDelayedList` without having acquired it earlier. Finally the only violation of this property reported by Spin is in the `fullreschedule` function, where it has taken a set of locks including the one on the running queue, and then tries to take other corelocks. Thus the only remaining hold-and-wait situation was in the `fullreschedule` function, which is intentional, as we want to minimize the time for which other corelocks are held by the scheduler. . Since this function runs only as part of the process on Core 0, a circular hold-and-wait condition cannot arise in M_n. This proves that M_n is deadlock-free.

E. Extracting the Promela model using Modex

We used the Modex model extraction tool [5] to automate the construction of the Promela model from the C implementation. Modex translates C statements to Promela according to the modex translation rules provided. Fig. 11 shows an excerpt from the Modex rules we used for translation.

An automated extraction has several advantages over a manually built model: in particular it helps to extract locking and accesses in a systematic and exhaustive manner, and allows automatic regeneration of the Promela model with every change in the source code.

To begin with we inlined all the helper function calls in each of the API's. This helped us to simplify the control flow with an API and avoid introducing unnecessary variables in the model. We then created a list of all shared variables we were interested in tracking for data-races. We used the %T option of Modex to define a sed (stream editor) script that translated every access of these variables say `var` to the output lines `cvar++; cvar--`. "If" statements were translated as they were, with an additional access statement in case it was reading a shared variable.

Some access statements were difficult to extract directly. For instance the `taskDelete` API makes a call to the `listRemove` function which takes a pointer to a node representing a task, and removes it from its *container* list, a pointer to which is present in the `container` field of the node. Before this it takes a lock on the container list. Thus, one knows the exact list accessed only at runtime. We got around this problem by conservatively translating such statements to a non-deterministic select statement where a list is chosen to be locked and subsequently the same list is accessed.

VI. Multicore FreeRTOS Windows Simulator

FreeRTOS offers a port for Microsoft Windows where the Timer and Global Interrupt Controller (GIC) components from the hardware layer are now simulated in the port-pecific layer. FreeRTOS tasks are created as a low priority Windows threads, in suspended state. Task execution and preemption is simulated by resuming and suspending the corresponding thread. Also, being a single core simulator it runs all tasks only on the first CPU core.

Interrupt are simulated by raising an windows event after setting the appropriate interrupt number. Another thread having the highest priority, preempts the running task and invokes the corresponding ISR. The interrupt generator and interrupt processor are synchronized on a Windows mutex.

```
//Source Filename for modelling.
%F newtasks.c
//Functions to be modelled.
%X -L vtaskdelete -i vTaskDelete
%X -L taskDelay -i vTaskDelay
//Translations table for all functions.
%L
Lock(...                                keep
Unlock(...                              keep
currentCore=portGetCurrentCore()        currentCore=coreid;
%%
//Translation table for vtaskdelete.
%L vtaskdelete
(!IsTheReadyListEmpty())                true
return                                  goto ENDDELETE
...
//Translation table for vtaskDelay.
%L vtaskDelay
mask=(mask|uxReadyListMask)             keep
xNextTaskUnblockTime=xTimeToWake        hide
return...                               goto ENDDELAY
...
//Globle Promela Code at start of the model.
%P
int lockVector = 0;
inline Lock(mask){
    wait = 1; hold = (lockVector & one);
int uxReadyListMask = 1 << 0;
...
//Globle Promela Code  at the end of the model.
%A
proctype taskproc(byte coreid){
true -> vTaskDelay(coreid);
init { run taskproc(0); }
ltl p1 { []( (cxRunningList < 2)
%%
```

Fig. 11. Excerpt from the Modex rules.

A. Modification of the Windows simulator for multi core behavior

We modify the FreeRTOS Windows simulator, to use our multicore-FreeRTOS implementation of the portable layer. We also modify the port-specific layer to implement locking.

We protect the lock bit vector with a Windows mutex, that should be acquired by any task that wants to acquire or release a resource guarded by the bit vector. When we schedule a task on core-i, we set the cpu-affinity of the corresponding thread to core-i. While this offers a realistic multicore behaviour, it also requires that to simulate a n-core multicore-FreeRTOS, the host windows machine should be having n cores.

VII. Evaluation of Performance

We evaluated our multicore implementation by comparing the performance on Windows simulator for single (n = 1), dual (n = 2) and quad (n = 4) cores against that on the standard single core (SC) FreeRTOS port for Windows simulator. The environment is Visual Studio 2012 Team Suite Edition under 32-bit Windows 7 Professional on a machine with an Intel CORE^2 quad CPU.

We used variants of standard FreeRTOS demos, the *full demo* and *blinky demo*. The *full demo* uses a large number of tasks, and many of them call different APIs. In our variant, we use 21 tasks and the different APIs called are *vTaskDelay*, *vTaskDelayUntil*, *vTaskSuspend*, *vTaskResume* and *vTaskPrioritySet* and the queue APIs *xQueueSend* and *xQueueReceive*.

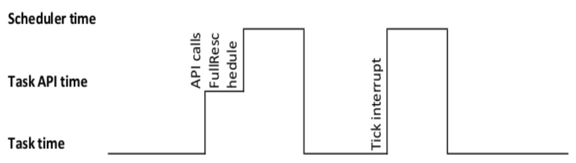

Fig. 12. Accounting time for taks, API and scheduler.

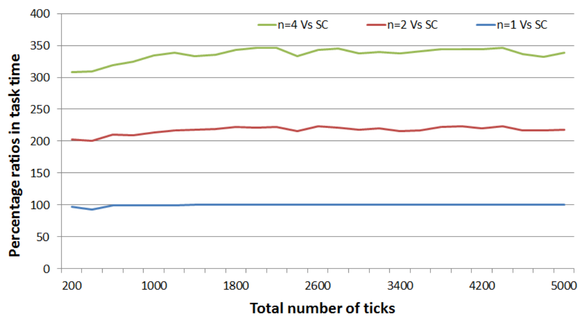

Fig. 13. Task time analysis of demo 1. Percentage ratios in task time are plotted against ticks. The graph shows that the task time multiplies with number of cores.

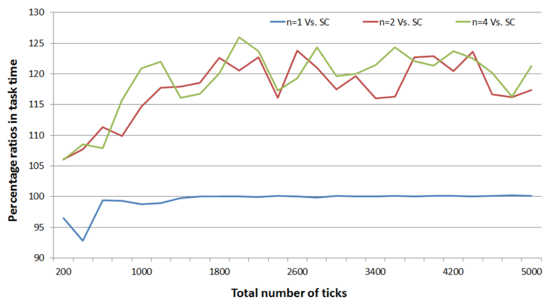

Fig. 14. Task time analysis of demo 2. Percentage ratios in task time are plotted against ticks. The graph shows that the task time does not multiply as in demo 1, the reason being the cores getting idle due to the absence of enough tasks.

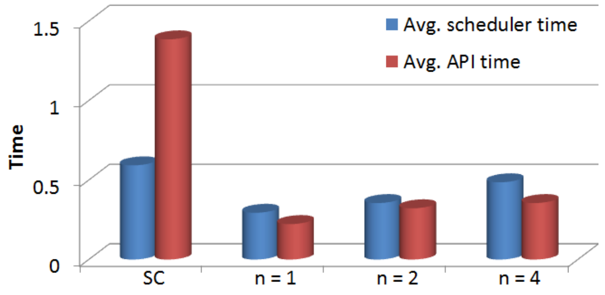

Fig. 15. Overhead statistics for demo 1. It shows that multicore-FreeRTOS overheads grow only fractionally with the number of cores, ensuring scalability.

The *blinky demo* is a producer-consumer demo where we have two tasks. The producer sends a message to a one slot queue periodically. The consumer continuously tries to read from the queue and if it finds the message there, it flashes an LED. We measure the timing using the standard FreeRTOS function *ulGetRunTimeCounterValue*, which measures the time elapsed in units of 1/100th of a millisecond.

Fig 12 shows the parameters we use for evaluation are *task time* and *scheduler time*. The *task time* is the total CPU time allocated to a task when the system is allowed to run for some fixed number of ticks. A task gets the CPU at the very moment it is swapped in for execution and continues to use it until the task is preempted. The *task time* includes the total processing time for the task and the time spent in invoking different task APIs. On the other hand, *scheduler time* is the total CPU time the scheduler consumed when it runs on a tick interrupt, and provides a measure for the system overhead.

Fig. 13 shows the percentage ratios of task times for different implementations, in comparison with that in single core, for demo 1, where we get n-times increase in task time. Fig. 14 shows the task time analysis of demo 2, where we don't get n-times increase in task time as we don't have enough tasks to fully utilize the n cores.

We also measure the average scheduler time, which is the average time needed to execute one round of scheduling, and the average API time, computed as the ratio of the total time spent in executing the APIs to the total number of API calls. Fig. 15[1] shows the average scheduler and API times for demo 1. Though these timings are highly dependent on hardware and the underlying primitives used for achieving mutual exclusion, they give an idea about the scalability of the algorithm. For $n = 1$, the average scheduler time is 0.25, for $n = 2$, it is 0.33 and for $n = 4$, it is 0.43 showing just a 30% (approximately) increase in scheduler time when the number of cores are doubled whereas the task time doubles. Similarly, the API timings also increase only marginally with the number of cores. This shows that our implementation is highly scalable.

[1]The high overhead statistics for SC is because it is using Windows mutexes for interrupt generation/handling and the operations disabling and enabling of interrupts are essential for scheduler and almost every API. Operations on a mutex are always slow.

VIII. Related Work

The most closely related work we are aware of is the effort to create a multicore version of FreeRTOS for an FPGA realization of a MicroBlaze soft processor [10]. As far as we can make out, they use essentially the same scheduling policy, though they don't spell out what happens for example when a high proirity task has to preempt one of the running tasks (in our case we use the running list to identify the longest-running lowest priority task to evict). Nevertheless, their work differs significantly from ours in a few ways. Firstly, it is not clear how fine-grained the locking protocol followed by the kernel API's is as this is not elaborated in the paper. Secondly, they use a version of Peterson's mutual exclusion protocol to provide mutually exclusive access to the kernel data structures. This has some drawbacks compared to the hardware-based locking we use (or the synchronisation mechanism provided by Windows in the case of the Windows simulator): the protocol is sensitive to instruction reordering carried out by processors, and explicit use of memory barriers (before each read and after each write) is needed to ensure mutual exclusion. This leads to considerable loss of decoupling. In addition, they also need to disable interrupts while accessing kernel data structures, due to the way mutex's are "owned." Finally, they don't verify any aspect of their implementation like freedom from data-races and deadlocks.

Other commercially developed multicore operating systems are QNX [11], On Time RTOS [12], OpenComRTOS [13], PikeOS [14] and SeL4 [15].

QNX implements priority based scheduling like FreeRTOS, and allows for FIFO, Round-Robin and Sporadic scheduling among tasks of equal priority. FreeRTOS's policy supports only round-robin, and the other can be simulated by the tasks by disabling preemption. In QNX it is possible that a lower priority task can get to run while a higher priority task is waiting to be scheduled on the processor it ran on (last), because the scheduler tries to maintain cpu affinity. In our multicore-FreeRTOS, we schedule only based on priority and processor affinity is not considered. While QNX has multicore support for both ARM and Intel x86 architctures, they are not verified. QNX Neutrino RTOS supports richer APIs like POSIX `fork`, and uses spin-locks and interrupt masking to achieve mutual exclusion. Whereas, multicore-FreeRTOS has simpler and easy to verify APIs, and uses atomic mutation to a lock bit-vector to achieve mutual exclusion. Also, not all parts of the QNX Neutrino RTOS is available as open source, making external verification impossible.

On Time RTOS uses an event-driven kernel, and scheduling is made in response to events generated by tasks. The timer interrupt is used only to bring the waiting tasks into ready state when appropriate. Otherwise, it also implements a priority based scheduler and has options for FIFO or Round-Robin scheduling among tasks of same priority. Also, development for On Time RTOS is supported only on the windows platform and only Intel x86 family of CPUs are supported. There is no verification or testing done for On Time RTOS either.

The OpenComRTOS project developed a RTOS for embedded systems, using formal modelling in TLA to verify the design of some components. However, it uses a virtual single processor model, that abstracts the underlying concurrency from the tasks.

The PikeOS and SeL4 projects both carry out formal verification of functional correctness of (single-core) RTOS's, using code-level verifiers (like VCC) or theorem provers. The interleaving of API's here is only due to interrupts and hence data-races and deadlocks are less of an issue here.

In work related to our verification effort, Voung et al [16] apply a lock-set based static analysis to detect data-races in the Linux kernel. While their technique has the advantage of scaling to a million lines of code, it is *unsound* in that it could fail to report some data-races. In contrast, our verification is sound and applies to an unbounded number of cores.

In [17], Havelund and Skakkebaek identify deadlocks in Java code, using Java PathFinder [18] which translates Java to Promela. However, they also *underapproximate* the program to cut down the size of the model and focus on potential deadlocks, leading to possible unsoundness of the check.

IX. Conclusion & Future Work

In this work, we have presented an efficient design and implementation of a multicore version of FreeRTOS, with verified guarantees on the freedom from data-races and deadlocks.

In future work, our overall aim is to specify and formally verify the functional correctness of multicore-FreeRTOS, along the lines of the work on verifying the the functional correctness of task-related functionality of FreeRTOS [19]. We would also like to statically check FreeRTOS applications for compatibility with multicore-FreeRTOS.

References

[1] W. Webb, "Embedded moves to multicore," April 2011, [posted 8-April-2011]. [Online]. Available: http://embedded-computing.com/articles/embedded-moves-multicore/

[2] Real Time Engineers Ltd., "The FreeRTOS Real Time Operating System," 2014. [Online]. Available: www.freertos.org

[3] EETimes, "Android, FreeRTOS top EE Times 2013 embedded survey," February 2013, [posted 27-February-2013]. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1263083

[4] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, 1997.

[5] G. J. Holzmann and M. H. Smith, "Software model checking: extracting verification models from source code," *Softw. Test., Verif. Reliab.*, vol. 11, no. 2, pp. 65–79, 2001.

[6] R. Barry, personal communication.

[7] S. Al-Bahra, "Nonblocking algorithms and scalable multicore programming," *Commun. ACM*, vol. 56, no. 7, pp. 50–61, 2013.

[8] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Trans. Comput. Syst.*, vol. 15, no. 2, pp. 134–165, May 1997.

[9] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971.

[10] J. Mistry, M. Naylor, and J. Woodcock, "Adapting FreeRTOS for multicores: an experience report," *Softw: Pract. Exper*, 2013.

[11] QNX Software Systems Ltd., "QNZ Neutrino RTOS System Architecture," 2014. [Online]. Available: http://www.qnx.org/download/feature.html?programid=26183

[12] On Time, "On Time RTOS-32 Documentation," 2014. [Online]. Available: http://www.on-time.com/rtos-32-docs/

[13] B. Sputh, O. Guast, E. Verhulst, V. Mezhuyev, and T. Tierens, "OpenComRTOS: Reliable performance for hetrogeneous real-time systems with a small code size," http://www.altreonic.com/sites/default/files/Whitepaper_OpenComRTOS.pdf, 2013. [Online]. Available: http://www.altreonic.com/sites/default/files/Whitepaper_OpenComRTOS.pdf

[14] B. Beckert and M. Moskal, "Deductive Verification of System Software in the Verisoft XT Project," *KI*, vol. 24, no. 1, pp. 57–61, 2010.

[15] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, p. 2, 2014.

[16] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *ESEC/SIGSOFT FSE*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 205–214.

[17] K. Havelund and J. U. Skakkebæk, "Applying Model Checking in Java Verification," in *SPIN*, vol. 1680. Springer, 1999, pp. 216–231.

[18] K. Havelund, "Java PathFinder, A Translator from Java to Promela," in *SPIN*, vol. 1680. Springer, 1999, p. 152.

[19] FreeRTOS verification project, "Project artifacts," http://www.csa.iisc.ernet.in/~deepakd/FreeRTOS, 2014. [Online]. Available: http://www.csa.iisc.ernet.in/~deepakd/FreeRTOS

[20] D. Dams, *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24, 1999, Proceedings*. Springer, 1999, vol. 1680.