Go-RealTime: A Lightweight Framework for Multiprocessor Real-Time System in User Space *

Zhou Fang †
University of California
San Diego

Mulong Luo
University of California
San Diego

Fatima M. Anwar
University of California
Los Angeles

Hao Zhuang
University of California
San Diego

Rajesh K. Gupta
University of California
San Diego

ABSTRACT

We present the design of Go-RealTime, a lightweight framework for real-time parallel programming based on Go language. Go-RealTime is implemented in user space with portability and efficiency as important design goals. It takes the advantage of Go language's ease of programming and natural model of concurrency. Goroutines are adapted to provide scheduling for real-time tasks, with resource reservation enabled by exposing Linux APIs to Go. We demonstrate nearly full utilization on 32 processors scheduling periodic heavy tasks using Least Laxity First (LLF) algorithm. With its abstraction and system support, Go-RealTime greatly simplifies the set up of sequential and parallel real-time programs on multiprocessor systems.

Keywords

real-time systems; multiprocessor systems; parallel programming; Go language; scheduling algorithms

1. INTRODUCTION

Multiprocessing is now the primary means of improving performance on diverse platforms from embedded systems to data-centers. Many frameworks and the associated tools make parallel programming easier, such as compiler support (OpenMP[1]), language extension (Intel Cilk Plus[2]) and library (Intel Thread Building Block (TBB)[3]). These tools provide easy-to-use programming support to construct parallel tasks, and orchestrate execution details behind the scene. Yet, the goal of minimizing execution time is only partially achieved due to the lack of resource reservation. This is particularly limiting for real-time (RT) applications.

The majority of prior work on real-time systems is focused on operating system (OS) support for scheduling of sequential tasks on uniprocessor models. Multiprocessor systems are attracting increasing research interest, which have more complex schedulers. Among notable multiprocessor system implementations, Calandrino *et. al.* proposed LITMUS^{RT} [7], a Linux based testbed for real-time multiprocessor schedulers. Based on LITMUS^{RT}, several different implementations of global Earliest Deadline First (EDF) algorithm [15] were tested and compared [6]. The drawback of global EDF algorithm on multiprocessor system is that schedulability of heavy utilization task is degraded dramatically [9]. Dynamic priority algorithms, such as Least Laxity First (LLF) [13, 17] and P-fairness [3], are able to achieve higher utilization than EDF on multiprocessor systems. However, they induce larger system cost due to frequent context switching and cache pollution.

Prior to scheduling, because there are many choices to decompose a parallel task into a group of sequential sub-tasks, parallel task scheduling algorithms have additional complexity on top of sequential scheduling. Lakshmanan *et. al.* [11] studied fork-join tasks and proposed the task stretch transform to reduce scheduling penalty. For a general class of parallel tasks, Saifullah *et. al.* [18] proposed to decompose a parallel task into several sequential sub-tasks with carefully computed deadlines, which are then scheduled using global EDF. This algorithm was then implemented by Ferry *et. al.* [1]. Due to lack of a direct API to make a thread switch, in [1] it indirectly switches threads by changing their priority. This indirect approach results in a complex implementation and limits the performance.

In this work, we propose a new RT framework, Go-RealTime to solve the above challenges using Go language[4]. It supports both real-time scheduling and parallel programming with a set of APIs. Go provides native support for concurrency by means of goroutines. We leverage this advantage of Go for real-time programs running on concurrency units in user space. OS thread scheduler is integrated into Go runtime and adopted for resource reservation. In Go-RealTime, we import OS APIs (thread scheduler, CPU affinity and timer) into Go, modify Go runtime to enable direct goroutine switch, and build external Go packages to support asynchronous events, parallel programming and real-time scheduling. Go-RealTime is able to make a given portion of processors (or a fraction of one processor) real-time and keep the rest unchanged, thus it supports implementation of mixed criticality systems. This approach is also safer and more portable than earlier works [7, 6] because no modifications to the OS kernel are needed.

In Go-RealTime, a user can program a parallel task as Directed Acyclic Graph (DAG) [16] of sub-tasks. Sub-tasks are executed by subroutines, so the overhead of process-level concurrency is saved. It supports multiple scheduling algorithms for different tasks running at the same time. Different types of tasks are scheduled by the suitable algorithm (EDF [15], LLF [13]). The total processor resources are partitioned dynamically among all schedulers by the resource balancing

*Copyright retained by the authors

†Email: zhoufang@ucsd.edu

[1]http://www.openmp.org

[2]https://www.cilkplus.org

[3]https://www.threadingbuildingblocks.org

[4]https://www.golang.org

mechanism. Implementation results show that our framework is lightweight, flexible and efficient to deploy real-time programs on platforms ranging from embedded devices to large-scale servers.

The main contributions of our work are: (1) we design a real-time parallel framework using Go language without modifying OS kernel; (2) we present the resource balancing approach to execute multiple scheduling algorithms in parallel; (3) we implement system support for handling asynchronous event in user space; (4) we implement a multiprocessor LLF scheduler with a simple tie breaking method, which can achieve near full utilization for randomly generated heavy task set ($0.5 \leq \mu_{task} \leq 0.9$) on a 32-processor machine; (5) we evaluate the implementation of EDF and LLF in Go-RealTime through analyzing system overhead and taking schedulability tests.

The rest of the paper is organized as follows. We introduce the system architecture of Go-RealTime in Section 2. It presents Go-RealTime's APIs, system model, resource balancing, and asynchronous event handling. Programming parallel task in the framework is presented in Section 3. The design, implementation and evaluation of real-time schedulers are presented in Section 4. Section 5 gives the conclusion and future works.

2. THE GO-REALTIME ARCHITECTURE

In this section we introduce the overall system architecture, APIs and data structure of Go-RealTime. It is currently implemented on top of Linux version of Go 1.4. Go-RealTime works concurrently with the original runtime of Go language. It creates RT goroutines which run real-time tasks. The threads carrying RT goroutines are separated from native Go threads (no goroutine migration). In the following discussion, threads and goroutines are created by Go-RealTime unless otherwise stated.

Go-RealTime is targeting for applications which require millisecond level timing precision. The requirement is given by δ_{async}, the upper bound of tolerable asynchronous event timing uncertainty. A real-time asynchronous event handler is designed to meet the requirement dynamically. The framework turns off Go's garbage collector to reduce timing uncertainty. In the current implementation, memory usage is managed by recycling dynamically allocated objects. Replacing the Go garbage collector by a more controllable approach which meets real-time requirement lies in the future work.

2.1 Go-RealTime's APIs

Go language's concurrency is enabled through goroutines and invoked with keyword *go*. A user creates a goroutine and associates it with a program using *go func(arg)*. After creation, go runtime scheduler automatically allocates goroutines to run on OS threads. Each thread created by the Go runtime keeps a runnable queue of goroutines. In the original Go runtime, a simple scheduling algorithm is adopted: each thread keeps a runnable goroutine queue in First-In-First-Out (FIFO) order. It tries to make a goroutine switch per $10ms$. User can yield the execution of a goroutine using the API *GoSched()*. Compared to Linux's Completely Fair Scheduler (CFS), this design is more lightweight and scalable. It is also much easier to use than thread libraries such as POSIX threads (Pthreads). However, the Go runtime completely hides system details from the programmer, thus making it difficult to carry out RT task scheduling due to lack of control over threads and processors. Go-RealTime modifies Go by adding resource and scheduling APIs (Table 1). These include:

Table 1: Go-RealTime APIs

Type	Method	Description
Resource	*SetSched(thread, policy, priority)*	Linux thread scheduler
	BindCPU(CPU)	processor affinity
	SetTimerFd(file_descriptor, time)	Linux timer
Scheduling	*GoSwitch(goroutine)*	switch to a goroutine
Task	*NewWorker(N_w)*	create N_w workers
	task.SetTimeSpec(t_s, t_p, t_d, t_r)	set timing specification
	task.Run(func, arg)	start a RT task

- *SetSched*: use Linux system call *sched_setscheduler* to change the scheduling policy of a Go thread from time-sharing (CFS as default) to RT (FIFO, Round-Robin).
- *BindCPU*: bind a Go thread to a processor using the system call *sched_setaffinity*. Then Go thread runs exclusively on the processor.
- *GoSwitch*: directly switch to a specific target goroutine. It is implemented by modifying Go runtime source code.

These modifications and additions make the execution of goroutines fully controllable in user space. The APIs can reserve a fraction f of one processor as well: two Linux timers are used to set Go thread RT at t_{rt} and set it back to time-sharing at t_{ts}. Given the period of timers t_{timer}, it satisfies $t_{ts} - t_{rt} = f \cdot t_{timer}$, so the Go thread occupies fraction f of the processor deterministically. It is useful to run RT tasks on uniprocessor systems. In this work we focus on taking N_{cpu} processors completely for RT tasks.

Go-RealTime relies on these APIs to implement RT schedulers in Section 4. As an alternative to directly using these APIs, the users can also use the easier Task APIs (Table 1), which assign the timing specifications and submit tasks to the RT scheduler.

2.2 Go-RealTime's System Model

The system model of Go-RealTime consists of three objects: *worker, task* and *goroutine*, as shown in Figure 1. A worker is a thread created by Go-RealTime. It is an abstraction of reserved processor resource. A Go-RealTime program creates a group of workers via *NewWorker(N_w)* method. The number of workers (N_w) is usually equal to N_{cpu} to maximize parallelism.

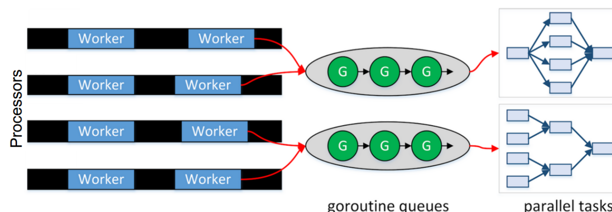

Figure 1: System model of Go RealTime. A worker is the abstraction of a thread with one processor resource reservation. Goroutines are sorted in a few priority queues. A parallel task is decomposed into DAG of sequential sub-tasks.

Task is the object to describe a RT program. It has the following features:

- Time specification: starting time t_s, period t_p, deadline t_d, and worst case running budget t_r, set by the API *task.SetTimeSpec(t_s, t_p, t_d, t_r)*.
- Utilization: $\mu_{task} = t_r/t_p$ (μ_{task} can be larger than 1 for parallel tasks).

Goroutine is the concurrency unit. A new RT goroutine is created by calling *task.Run(func, arg)*. *task* then starts to run on the new goroutine. A group of workers shares some runnable queues of goroutines. N_{queue}, the number of queues, is a parameter of Go-RealTime scheduler. Priority queue is used to sort goroutines by a key ("deadline" in EDF, "laxity" in LLF). We use the lock free priority queue data structure [10] to reduce the cost of parallel access contention.

Workers fetch the head goroutine in queue to execute. When a worker becomes idle, it fetches the first goroutine in queue. The scheduler is preemptive: if a goroutine g_0 becomes runnable and its priority is higher than g_1 which is running, the scheduler switches off g_1 and let g_0 run. A sequential task is executed by one goroutine. A parallel task is decomposed into sequential sub-tasks described by DAG model, executed by a group of goroutines. Since no preemption happens among sub-tasks of the same parallel task, they are implemented simply as subroutines.

2.3 Resource Balancing in Go-RealTime

Go-RealTime is able to keep more than one goroutine queue to support multiple scheduling algorithms in parallel. This approach has two benefits: (1) it supports partitioned scheduling policy, and different algorithms can be deployed for different sets of tasks; (2) scheduling algorithm can be changed on-the-fly without interrupting running tasks.

An example with two queues is shown in Figure 2, one queue uses EDF algorithm, and the other uses LLF. Load imbalance happens when the total task utilization of EDF and LLF queues do not match their allocated processor resources, which induces utilization loss. Load balancing technique such as work stealing [5] is a solution. Whereas for RT scheduling, work stealing becomes complex because it must respect task priority during stealing. Otherwise it may run lower priority tasks but leave higher priority tasks waiting. Go-RealTime uses resource balancing instead of load balancing. Because all tasks are periodic and their utilization μ_{task} are known, total utilization of a queue is simply the sum $\mu_{queue} = \sum \mu_{task}$. It is the amount of processor resource the queue should get. The total resources, N_w workers, are allocated to the queues proportionally according to μ_{queue}.

When the load of one queue (μ_{queue}) changes, the number of workers allocated to each queue (N_w^{queue}) is recomputed. Go-RealTime dynamically assign workers to queues according to N_w^{queue}. In most cases, N_w^{queue} is not an integer, so a queue may be assigned a fraction of a worker. In this example (Figure 2), *worker 1* is allocated to both queues. A fraction f_{EDF} of this worker should be allocated for EDF queue, and f_{LLF} for LLF queue ($f_{EDF} + f_{LLF} = 1$). *Worker 1* tries to give the right fractions of total running time to both queues. It records the total running time of each queue. When it becomes idle and is going to grab a goroutine, it firstly checks the current fraction of total running time of each queue, and selects the queue which has received the smallest fraction. Go-RealTime asynchronously checks and enforces resource balancing while goroutine is running, repeated by a given time interval $t_{balance}$. The implementation will be explained in the next section.

2.4 Handling of Asynchronous Events

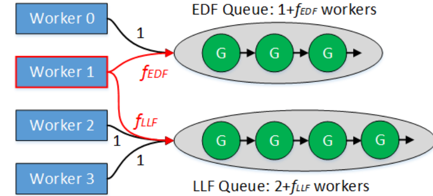

Figure 2: Resource balancing: allocating the correct fraction of the workers to goroutine queues.

Go-RealTime handles asynchronous events via *check_async* method. It checks all asynchronous events (timeout, message, etc.) and responds to the events which should have occurred. The challenge is how to call *check_async* method repeatedly in an asynchronous manner. Our current implementation is to insert *check_async* calls in the program until the interval between two consecutive checks is less than δ_{async}. By skipping some of these checks, we can control the timing precision. Algorithm 1 gives an example of instrumenting a *ConvertGray* function with *check_async*. The function converts a colored image in memory to gray scale. The three locations to insert *check_async* (*@out*, *@lpy*, *@lpx*) result in different timing precisions and overheads. To tune precision in a finer way, it can skip a fixed number of *check_async* calls using a counter. For example, it can call *check_async* once every ten times when the program runs to location *lpy* (denote as "*lpy, 1/10*").

Algorithm 1 Instrumented *ConvertGray*

```
 1: function ConvertGray
 2:     @out: check_async()
 3:     for Loop over Y coordinate do
 4:         @lpy: check_async()
 5:         for Loop over X coordinate do
 6:             @lpx: check_async()
 7:             SetPixelGray(image, x, y)
 8:         end for
 9:     end for
10: end function
```

The *check_async* facility is used in a number of components in Go-RealTime as discussed below.

2.4.1 Timer

Go-RealTime implements a software timer upon *check_async*. It keeps a priority queue of active timers at each worker. The queue uses "time" value to assign priorities. The current running goroutine on a worker is responsible for checking the timers. *check_timer* method gets called inside *check_async*. It compares the head timer in queue with the current clock time. If any timeout is due, it calls the handler function of the timer and removes the timer from the queue. We use timer uncertainty to evaluate timing precision of the framework. Timer uncertainty is computed as $\Delta_{timer} = t_{notify} - t_{expect}$. t_{expect} is the expected timeout of timer. t_{notify} is when the program is notified and responds to the timer.

We compare timers of Linux (notify via file descriptor), Go and Go-RealTime. Δ_{timer} is measured by repeatedly setting a timer and computing $t_{notify} - t_{expect}$. All tests are running on real-time threads on a server with 32 Intel Xeon E5-2640 2.6GHz CPUs running CentOS 6.6. The statistics of Δ_{timer} is shown in Figure 3. The x-coordinate

is the value of Δ_{timer} and the y-coordinate gives the cumulative probability function. The overhead of Go-RealTime timer is given near the curve. The results show that the uncertainty of the Linux timer is around $100\mu s$. The native Go timer daemon is running on a goroutine, which may be influenced by other goroutines. In order to test its performance under load, we create a few dummy goroutines running on the same thread as the timer goroutine (load=1 means one dummy goroutine). We see that 10 goroutines increase the uncertainty to sub-second level, which means the native Go timer can not guarantee its precision. The precision of Go-RealTime timer depends on program details (*check_async* location). We use the program in Figure 1 for the test. Because all Go-RealTime goroutines are checking the timers generated by other goroutines as well, running multiple goroutines does not influence the timer precision. The result shows *check_async* method is able to achieve millisecond-level precision with small overhead ($2.9ms/s$ for *lpy*, $0.3ms/s$ for *lpy* 1/10). Go-RealTime timer does not rely on OS timer. Compared to the Linux timer, it provides the ability to tune precision and overhead in a large range.

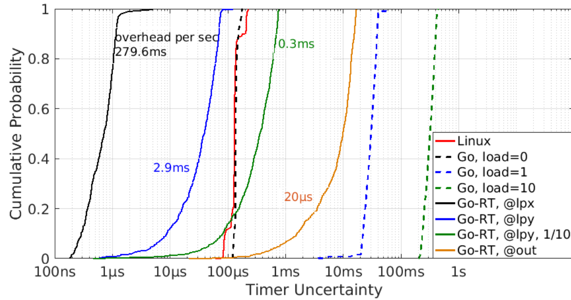

Figure 3: Timer uncertainty comparison of Linux, Go and Go-RealTime. Each test contains 500 samples.

2.4.2 Messages

Passing asynchronous messages and getting on-time responses is important in RT systems. In Go-RealTime, each worker has a message queue to be updated by asynchronous events. The current running goroutine checks the messages when *check_async* runs. In this way the running goroutine can respond to asynchronous messages quickly. As an instance, for EDF and LLF scheduling, an asynchronous message is sent to the running task when a new task becomes runnable. Upon receiving the message, the running task compares its priority with the new one, then the task with higher priority continues to run.

2.4.3 Other Events

Go-RealTime implements a few other asynchronous events based on *check_async*. Two representative examples are (i) checking resource balancing is done in *check_async* method every $t_{balance}$, and (ii) in LLF scheduling, all goroutines actively update and compare laxity in *check_async* method every t_{llf}.

3. PARALLEL TASKS IN GO-REALTIME

Go language provides goroutine and channel constructs to build scalable concurrent programs. Parallel programming in Go is studied in [19] using a dynamic programming example that we will use to illustrate the advantages of Go-RealTime. The example addresses a search method: given N_{key} keys and the probability of each key, find the optimal binary search tree that minimize average search time. The graph of computing nodes is shown in Figure 5 (a). It starts from the diagonal nodes and ends at the right upper corner node. The program is parallelized by grouping nodes as sub-task as Figure 5 (b). In the reference implementation in [19], each sub-task is executed as a goroutine. Because there is a fixed dependency among all sub-tasks, there is no need for each sub-task to exclusively occupy a concurrency unit, which costs resource and increases switch time. Executing each sub-task as a subroutine is more efficient.

Go-RealTime models a parallel task as a DAG of sub-tasks. It stores runnable sub-tasks in a global pool protected by a spin lock. A sub-task is pushed into the runnable pool after all its predecessors have completed. When a goroutine is idle, it fetches a sub-task to execute in the pool of its associated parallel task. A goroutine sleeps if the pool is empty and is woken up when new sub-tasks are runnable. Go-RealTime creates a group of goroutines for a parallel task. It includes a initializing goroutine, a finalizing goroutine, and a few goroutines to execute the parallel section, constructed in the fork-join pattern, as shown by Figure 4. A goroutine is blocked until all its predecessors finish. The initializing goroutine is responsible for preparing for parallel computation, such as loading data and initializing sub-task pool. The finalizing goroutine does clearing work and handles computation result. The API to construct a parallel task is the *subtask* object: users create a set of sub-tasks, assign each sub-task a sequential program to run and a list of predecessor/follower in the DAG.

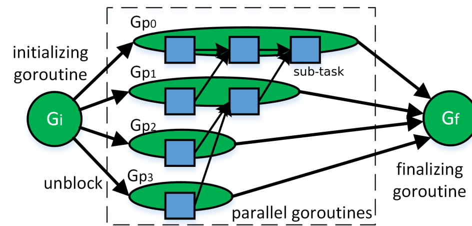

Figure 4: In this example, a parallel reduction task is decomposed into a DAG of sequential sub-tasks. It is executed by a group of goroutines constructed in the fork-join pattern.

The topology of a DAG is decided by the decomposition of a parallel task. A finer grain of decomposition leads to better parallelization. But it also induces larger system overhead of sub-task switching and contention at spin lock. For the example in Figure 5, we denote the number of sub-tasks on the diagonal as N_p, the total number of sub-tasks is $N_p \cdot (N_p + 1)/2$. Figure 5 (b) illustrates that $N_p = 8$ leads to better utilization of parallel resource than $N_p = 4$. We test the program on 16 processors using different values of N_p with $N_{key} = 1024$. As shown in Figure 6 (a), when $N_p = 16$ the parallel resource can not be effectively utilized. When N_p becomes 128, as Figure 6 (b) shows, the parallel resource is well utilized. The span is largely reduced by around 4 times. User should decide the most suitable granularity considering both span and cost. The scheduling problem of a set of finely parallelized tasks, for which sequential sections are ignorable, can be simplified as uniprocessor scheduling problem, thus global EDF allows us to achieve optimal scheduler implementation.

4. REAL-TIME SCHEDULER

Our goal was to design a RT scheduler for Go-RealTime that manages both sequential and parallel tasks concur-

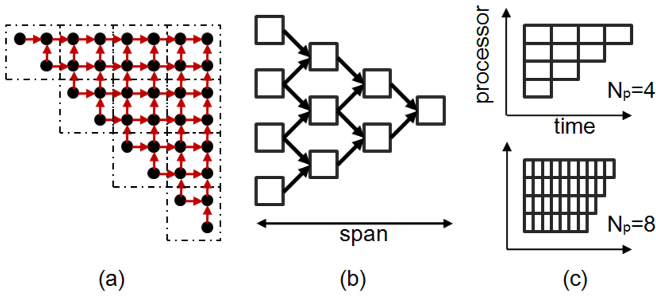

Figure 5: Parallel programming example. (a) dynamic programming grids are group into boxes as sub-tasks, (b) DAG of sub-tasks and (c) impact of decomposition granularity.

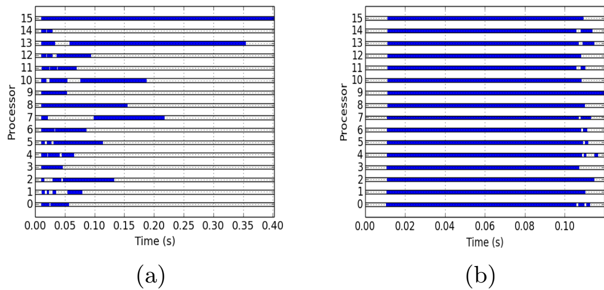

Figure 6: Parallelize dynamic programming on 16 processors with N_p as (a) $N_p = 16$ and (b) $N_p = 128$.

rently. With the help of the resource balancing design, several different scheduling algorithms can work in parallel with correct portions of processors. To simplify the implementation of parallel scheduling, Go-RealTime uses EDF to schedule all parallel tasks. It is optimal for finely parallelized programs. In this section, we focus on the sequential scheduling problem. We present our EDF/LLF implementation, and demonstrate the gains due to on-the-fly scheduling algorithm change in Go-RealTime.

4.1 Scheduling Algorithms

EDF is a fixed priority scheduling algorithm. Given a set of N_{task} tasks, the total number of switching is at most N_{task}. However, for multiprocessor system, a single sequential task can not utilize parallel resources. Keeping a high number of concurrent runnable tasks is important to better utilize parallel resources. LLF uses laxity instead of deadline as the metric of priority. Laxity t_{laxity} is the latest starting time to meet deadline, calculated by $t_{laxity} = t_d - t_r + t_e$ (it follows the same definition in [17]). t_e is the time that a task has already been executed. LLF scheduler uses the worst case t_r given by user. When a task is running, its laxity increases. Laxity ties happen when two tasks have the same laxity and repeatedly switch to each other. Theoretically LLF may induce infinite task switching because of tie. Previous work on LLF proposed several strategies to break laxity ties and bound switching times [17], but they are designed for single processor and do not utilize parallel resources. Go-RealTime breaks laxity ties in a simple way: it compares laxities of the running task with the head task in queue in *check_async* method. A switch is allowed only after t_{llf} since the previous. Therefore t_{llf} is the minimal interval of task switching caused by LLF. $t_{llf} = 10ms$ is used by default.

We test the EDF and LLF algorithms using 4 and 32 processors, respectively. The test program is a simple infinite loop. It checks timer queue every $1ms$. Period and utilization of tasks are randomly generated from an uniform distribution. Period t_p ranges in $100ms \leq t_p \leq 300ms$. Two types of tasks are considered: light task ($0.1 \leq \mu_{task} \leq 0.5$) and heavy task ($0.5 \leq \mu_{task} \leq 0.9$). In the test we run each set of tasks for $10s$, which has a random total utilization μ_{set}. We calculate the ratio of successfully scheduled sets (sets without deadline miss) r_{sched} for small ranges of μ_{set}. The result is given in Figure 7. It shows that for sets of light tasks (Figure 7 (a)(c)), EDF and LLF can achieve high utilization close to upper bound, on both 4 and 32 processors. Performance of EDF is slightly worse than LLF. For heavy tasks, LLF still achieves high utilization, whereas a large ratio of task sets is not schedulable using EDF (Figure 7 (b)(d)). The result confirms with the bad performance of EDF for high utilization tasks. LLF is the first choice for higher utilization in such scenarios. However, frequent goroutine switching induced by LLF may incur large system overhead. Next we quantify the overhead to comprehensively understand the performance of schedulers.

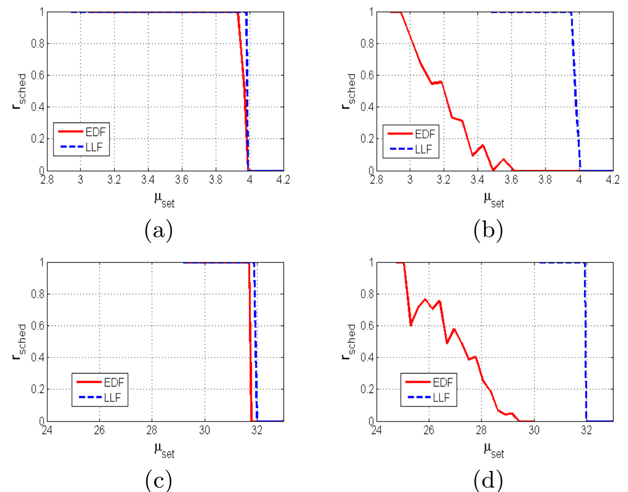

Figure 7: Real-Time scheduler performance. The tests for EDF on heavy tasks contain 400 samples. The other tests contain 100 samples each. (a) 4-processor, light tasks, (b) 4-processor, heavy tasks, (c) 32-processor, light tasks and (d) 32-processor, heavy tasks.

4.2 System Cost

System cost is an important consideration for the timing overhead it represents. The cost includes direct and indirect components. Direct cost consists of time consumed by the framework code and goroutine switch. Indirect cost that we consider is mainly cache pollution.

4.2.1 Direct Cost

We classify direct cost into three sources: queueing, switching and timer. Queueing cost is the time consumed by scheduler code, the majority comes from the cost of *insert* and *fetch* operations on goroutine queues. Increasing number of concurrent tasks leads to higher queueing cost. Shorter period of tasks also increases the queueing cost. This is because goroutine queueing operation is more frequent, which results in intense contention of parallel access. Switching cost is the direct cost of goroutine switch. The overhead of

timer is mainly induced by *check_timer* call, decided by its calling interval. It also includes the cost of operations on timer queue such as insertion.

The histogram of direct cost on 32-processor tests is given in Figure 8. Beside light and heavy tasks, we consider ultra light tasks which has $0.01 \leq \mu_{task} \leq 0.1$, to study the queueing cost with a large number of tasks in queue. The result shows that the direct cost is still small when 500 ultra light tasks run on 32 processors (smaller than 0.055%, shown in (a)). As the number of tasks reduces and period increases (like most application cases), the direct cost becomes even smaller and can be ignored.

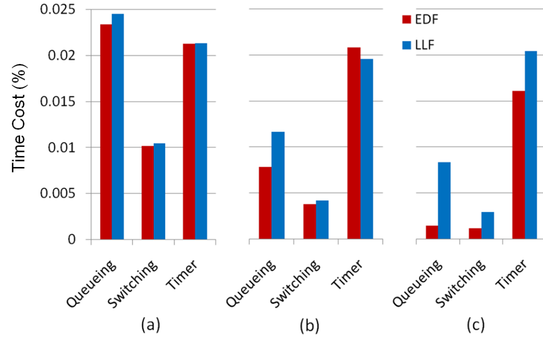

Figure 8: System cost of EDF and LLF scheduler on a 32-processor machine: (a) ultra light tasks, (b) light tasks and (c) heavy tasks.

4.2.2 Cache Pollution

When goroutine switch happens, the current cache status may be invalid due to context change of data and code. The time required to update cache from memory is called cache pollution. The cost highly depends on programs and machines, thus it is hard to be exactly quantified. The cost reported by [14] is in the range of milliseconds. Therefore it should be considered as a major cost. We use the number of goroutine switch as the metric of indirect cost. Figure 9 gives the histogram of average number of goroutine switches per second·processor. It shows the switch induced by LLF is evidently larger than EDF. Because of similar performance in scheduling ultra light and light tasks, EDF is preferred in these cases to reduce switching cost.

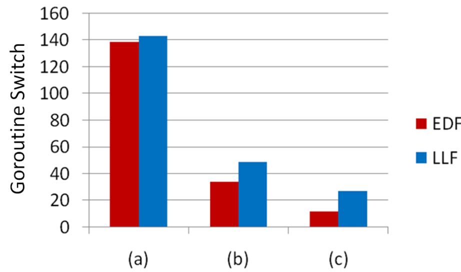

Figure 9: Average count of goroutine switching per second·processor: (a) ultra light tasks, (b) light tasks and (c) heavy tasks.

4.3 Schedulability Evaluation

We give the details of schedulability evaluation of Go-RealTime using EDF and LLF. We use the convention in [2, 12, 4, 8] to generate the test sets. That is, starting with zero task in the task set, we randomly generate tasks with uniformly distributed period and utilization, then add this task to the current task set. We check the total utilization of this task set, if it is less than the lower bound of the predetermined utilization, we continue to add new task into this task set until the total utilization is in the range, then this task set is tagged as valid for schedulability test. If the total utilization exceeds the upper bound, we abandon this task set and start with a new task set with zero task. For the total utilization of each task set, we set it to be within 2.0 to 3.9. The period of each task is uniformly distributed between $100ms$ to $300ms$. For light tasks, the utilization per task is uniformly distributed between 0.1 to 0.5, for heavy tasks the utilization per task is uniformly distributed between 0.5 to 0.9. We generate 5000 task sets for each case of testing.

Figure 10 shows the results of schedulability experiments. To be specific, Figure 10(a) shows the number of schedulable task sets with only light tasks. For light tasks, the number of tasks for each total utilization is almost uniformly distributed. We can see that for both EDF and LLF algorithms, all the task sets are schedulable. Figure 10(b) shows the number of schedulable task sets with heavy tasks. For heavy tasks, the distribution of tasks under different total utilization is not uniform. It first increases with the total utilization then decreases with it. We see that for EDF, the number of successful task sets drops quickly with the increase of total utilization. While for LLF, the number of successful task sets decreases slowly. This implies that LLF performs better than EDF for this settings in Go-RealTime.

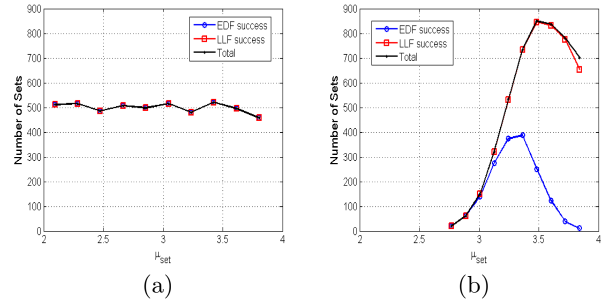

Figure 10: Number of schedulable task sets for (a) light and (b) heavy tasks out of 5000 task sets on 4 processors.

4.4 On-the-fly Scheduling Algorithm Change

The support for multiple schedulers in Go-RealTime is the key to combine advantages of different algorithms, such as EDF's low cost and LLF's high schedulability. The other benefit of the design is that one scheduling algorithm can be changed to another on-the-fly, when the features of the task set become different. For example, when the total utilization increases, the scheduler should change from EDF to LLF in order to avoid deadline miss. During the changing process, both EDF and LLF goroutine queues are kept. Each task is inserted into LLF queue at the beginning of its next period, while the current state (state may be "running", "runnable in EDF queue" or "idle until next period starting") of the task remains unchanged. An example of this process is shown in Figure 11. Scheduler changes from EDF to LLF at time $4s$. It can be found that after the change, the schedulability of the task set is improved upon more frequent task switches.

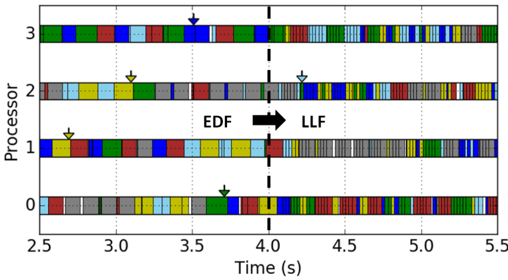

Figure 11: On-the-fly scheduling algorithm change: a task is represented by a colored box. The box boarder means an attempt of task switch. For EDF, this attempt is only made when a current task ends or a new task is runnable. LLF makes a switch attempt per t_{llf} ($t_{llf} = 20ms$ used here). For both EDF and LLF, an attempt successes if a current task ends or a new task has higher priority. An arrow stands for the time when a deadline miss happens.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we present the design and implementation of Go-RealTime, a real-time parallel programming framework implemented in user space using Go programming language. Important design choices related to resource reservation, asynchronous event handling and programming interface make it possible for the application developer to embed strong timing requirements and ensure their satisfaction through a flexible runtime scheduler. It also supports DAG-based parallel programming to deploy parallel program on multiprocessor system.

Go-RealTime is implemented by modifying open source Go runtime, implementing Go packages and by importing important Linux system calls into Go. It uses Linux thread scheduler for resource reservation. Goroutine running on top of thread is the unit of concurrency. Our framework greatly simplifies the implementation and deployment of RT programs, and improves the flexibility in system extension.

Our prototype of Go-RealTime is moving forward in the following directions: (1) design a strategy to automatically place *check_async* method based on timing profile of programs; (2) design a controllable approach to run garbage collection which respects the priority of RT tasks.

6. REFERENCES

[1] K. Agrawal, C. Gill, J. Li, M. Mahadevan, D. Ferry, and C. Lu. A Real-Time Scheduling Service for Parallel Tasks. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272, 2013.

[2] T. P. Baker. Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time. *FSU Technical Report, TR-050601*, 2005.

[3] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 345–354, 1993.

[4] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2009.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[6] B. B. Brandenburg and J. H. Anderson. On the Implementation of Global Real-Time Schedulers. In *Proceedings of the 30th IEEE International Real-Time Systems Symposium (RTSS)*, pages 214–224, Dec 2009.

[7] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 111–126, 2006.

[8] M. Cirinei and T. P. Baker. EDZL Scheduling Analysis. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 9–18, 2007.

[9] J. Goossens, S. Funk, and S. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, 25(2-3):187–205, Sept. 2003.

[10] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

[11] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *Proceedings of the 31st IEEE International Real-Time Systems Symposium (RTSS)*, pages 259–268, 2010.

[12] J. Lee and I. Shin. EDZL Schedulability Analysis in Real-Time Multicore Scheduling. *IEEE Transactions on Software Engineering*, 39(7):910–916, July 2013.

[13] J. Y. T. Leung. A New Algorithm for Scheduling Periodic, Real-Time Tasks. *Algorithmica*, 4(1):209–219, 1989.

[14] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, 2007.

[15] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[16] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers, 2012.

[17] S.-H. Oh and S.-M. Yang. A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks. In *Proceedings of the 15th International Conference on Real-Time Computing Systems and Applications (RTAS)*, pages 31–36, 1998.

[18] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In *Proceedings of the 32nd International Real-Time Systems Symposium (RTSS)*, pages 217–226, 2011.

[19] P. Tang. Multi-core Parallel Programming in Go. In *Proceedings of the 1st International Conference on Advanced Computing and Communications*, 2010.