Real-Time Final Project W2020

Justin Washington WASHINGTONJ@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

Dawson d'Almeida DALMEIDAD@CARLETON.EDU
Carleton College, 300 North College St. Northfield Minnesota 55057 USA

1. Abstract

The 2016 paper *Real-time scheduling algorithm for safety-critical systems on faulty multicore environments* details a scheduling algorithm designed for multicore systems where any number of cores can fail temporarily or permanently. The algorithm, called FTM, schedules and executes backups in order to recover from core failures. FTM is a global offline preemptive fixed-priority algorithm.

2. Introduction

This goal of this project was to implement a multicore environment with core dropouts and attempt to find trends in scheduling feasibility with the the algorithm of the paper, FTM. FTM is a global scheduling algorithm that was designed to resolve the constraints of real-time systems with the fact that there are real-world situations that cause temporary or permanent core failures.

Particularly with regard to safety-critical real-time systems (such as those in a vehicle or medical device), certain real-time and fault-tolerant constraints must be met. FTM capitalizes on the ability to execute tasks in parallel on a multicore system to meet these design constraints. The salient feature of FTM that allows this are backups. Active backups of a job are released alongside the original, or primary, job of a task. Passive backups are released when the system has detected that the primary and all active backups have failed.

FTM's main contribution is an expression that computes the probabilistic schedulability guarantee of a schedule, which translates to a high probability of a multicore platform completing the jobs in the specified taskset in a correct and timely manner. This process is done offline, as one of the main components of FTM is determining the appropriate number of active backups to include for each task's jobs. Given a probabilistic scheduability guarantee, a level of assurance can be derived and whether or not the multicore safety-critical system can be used is determined.

3. Background

Previous work around both global and partitioned multicore scheduling exists, but has limitations in the modern world, such as the lack of differentiation between job errors and core failures, static allocation of backups to cores, or lack of preemptive ability (ie the abort-and-restart computation model). FTM seeks specifically to fill the void of a global scheduling algorithm that distinguishes between job errors and core failures, particularly because studies in the 2000's revealed that job errors happen with much more frequency than core failures.

Error Model Errors handled by FTM are categorized as either job errors or core failures. Both are handled in the same way with backups executing after either is detected, but separating their probabilities allows for more resource-efficient fault-tolerance. Specifically, this separation of application-level error (job error) and stochastic hardware-level faults (core failure) merit more realistic calculations for the worst-case schedulability analysis, which is explored in the original paper and contributes to the computation of the probabilistic schedulability guarantee but not implemented here. We assume that the output of a job is correct when the job finishes.

Task Model The paper describes a sporadic, constrained-deadline task model.

Backups Each job in a task has the primary job, some number of backup jobs, and passive jobs. All jobs (primary and backups included) must execute sequentially, though different primaries and backups can execute in parallel. All backups have the same priority as their primary. Active backups are released alongside primaries, meaning they have the potential to execute in parallel with the primary. Passive backups are released on a one-by-one basis as soon as the primary and all previously released backups fail (or execute erroneously). This means that if any primary or active backup successfully completes, no passive backups are released.

Tasks and Cores Each task has an id, period, relative deadline, and worst case execution time. Cores track if they are faulty, if they are currently executing, and what job is executing on them. In our implementation, jobs of a task are released as many times as possible in the time frame (implicit-deadline). We also use cores in a work-conserving manner. We included an additional id for each job, called the backup id, which refers to what copy of some task's job a particular job is. Note that this model differs slightly from the one described in the original paper. See the future work section for differences.

Fault Model Each core is independent and either working correctly or not functioning for any given unit of time (known as fail-stop cores). Core failures are represented stochastically. Faults in the real world tend to occur in bursts as well as randomly. To simulate this, the fault model uses the following:

- Bursty Period: the section of time where faults occur in bursts randomly (ie with higher probability)
- Non-Bursty Period: the section of time where faults occur only randomly
- L_B: expected average length of the bursty period, which is geometrically distributed
- L_G: expected average length of the non-bursty period, which is geometrically distributed
- λ_b: core fail rate in a bursty period
- λ_r: core fail rate in a non-bursty period
- λ_c: core permanent fail rate

For our implementation, it is worth noting that the expected inter-arrival time of fault periods, or L_B+L_G, changed for each fault period based on the outcome of the result of the geometric distribution. This means that between and within each individual faulty core(s), all fail rates are the same, but the lengths of their L_B and L_G can differ. Additionally, we could not find any indication to how long each faulty period lasts. We handled this by scaling up the expected values of L_B and L_G by a passed-in value.

Terms

- Real-Time Constraint: a job must finish correctly before its deadline
- Fault-Tolerance Constraint: a job recover from any possible occurrence of error during execution
- Fault: the source of an error; what causes failure
- Core Failure: a core is faulty (permanent or non-permanent)
- Chip Failure: all cores on a multicore system are faulty (we are not concerned with this)
- Job Failure: a job misses its deadline
- Job Error: the behavior of a job is incorrect, ie wrong output or path (we also are not concerned with this in our implementation, though technically FTM handles this well. The goal of the algorithm is to avoid job failures with and obscure job errors)
- Primary: the job in the original taskset
- Active Backup: duplicates of a primary job released at the same time as the primary
- Passive Backup: duplicates of a primary job released after a primary and its active backups have all failed

4. Code

We used Tanya Amert's boilerplate code for uniprocessor scheduling as a template for our multiprocessor ones. Cores are components of a CoreSet. Cores know if they are faulty, their id, if they are executing, and if so, what job they are executing. CoreSets hold the fail rate information and use a geometric distributions to calculate the bursty and non-bursty periods of each core. Tasks are components of a TaskSet. Tasks have an id, period, wcet, relative deadline, and jobs. Jobs have an id, a backup id, the task they're a part of, release time, remaining execution time, absolute deadline, etc. JSON files are parsed to create the initial batch of jobs. The backups are then made for each job, with each successive backup having an incrementally increasing backup ids. For example, if there is 1 active backup for each job, $T_{2,4,0}$ would be the primary job of task 2 job 4, $T_{2,4,1}$ would be the active backup, and $T_{2,4,2\ldots n}$ would be passive backups. We build our schedule by stepping through each unit of time and making a scheduling decision for each core.

We would preprocess the inter-arrival time of fault periods to determine what probability to use to see if a core is failing for a unit of time. A failing core meant we would interrupt whatever job the core was running and prevent any job from being scheduled on it. Otherwise, all cores would check for the earliest deadline of ready-tasks and currently running tasks to determine what task to run and/or preempt. Preemptions only occur when all cores are executing (work-conserving) and always occur on the core with the lowest priority job first (fixed-priority by job). In the event that a job is not completed and the primary and active backups fail, a passive backup is released into the ready queue prior to making the scheduling decision for that time unit. Our display is based off of the display we've used for uniprocessor systems. We modified Tanya's code to also create a display that shows cores and the jobs executing on them. Because we don't add intervals to the display in large chunks, we greatly altered the original file to handle the $m \times t$ one-unit intervals passed in.

5. Algorithm

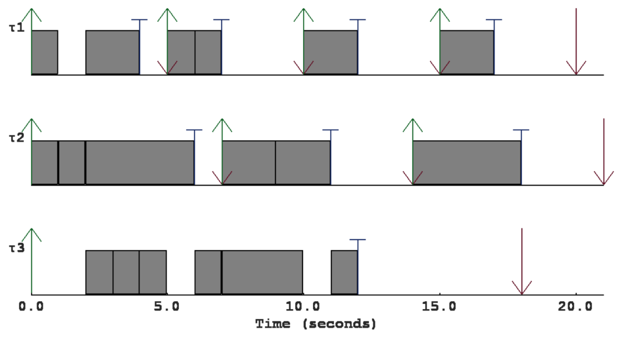
Given m cores and $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$:

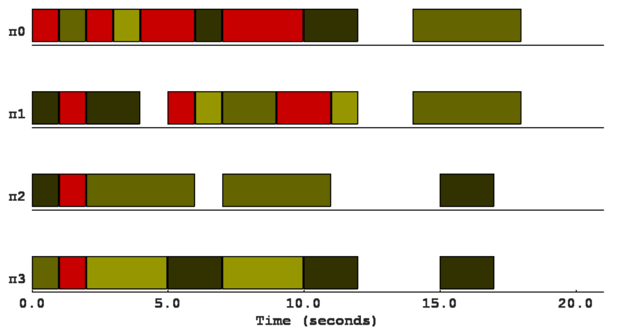
1. Add the primary and active backups of τ_i to the ready queue when a new job of τ_i is released.
2. No jobs are dispatched on failing cores. (In our implementation, computation of faulty core periods happens on an individual core basis and precedes all scheduling computation.)
3. Jobs from the ready queue are dispatched based on preemptive global fixed-priority scheduling. If an active core is idle, the HP job in the queue is placed there. If all cores are active, preemption can occur. (In our implementation, we make decisions on cores ordered by lowest priority job first.)
4. If a primary and all active backups fail, passive backups are released to the ready queue on a 1-by-1 basis.

6. Results

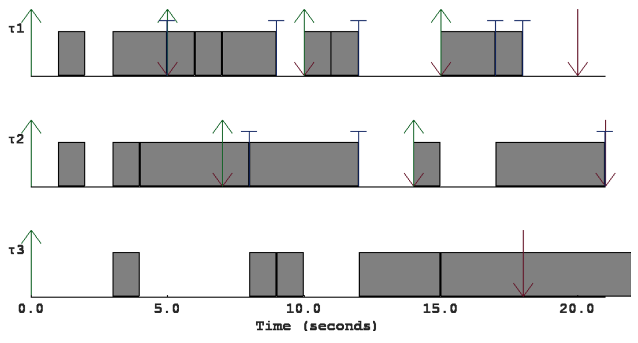
Since we did not have time to implement the worst-case schedulability test or the math that calculates the optimal number of active backups to include for each task, we did a couple random tests, and also provided code to run tests and build graphs for the proportion of time a taskset is scheduable when altering coreset elements. Following are examples of feasible and infeasible schedules, and the displays that would accompany them.

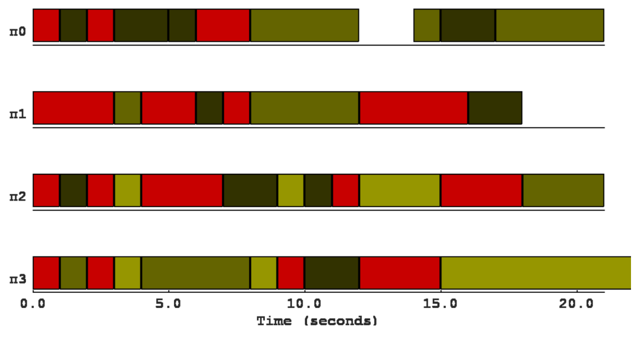
Displays of feasible schedule





Displays of infeasible schedule





7. Conclusion

Implementing the foundational pieces of this paper were challenging. Writing and debugging the code to simulate a multicore platform with core dropouts and displays for both tasks and cores took many hours. While we did not get to the schedulability test and calculations for number of active backups at the task-level, having a codebase that uses the same modular structure of the uniprocessor simulator makes this project's deliverables applicable to many other real-time systems projects (namely, multicore ones). We did our best to leave ways to conduct further tests for probabilistic feasibility given varying parameters of the coresets, such as number of cores, number of faulty cores, bursty period chance, and fail rates.
FTM is a really cool algorithm that has the potential to save lives, and working with it was both a great learning experience and a rewarding process.

8. Future Work

We encountered elements of the original paper that were very cool but did not have time to include. Here is a list of them:

- Aborting active backups as soon as a non-erroneous completion occurs.
- Implementing job error, instead of assuming all jobs execute correctly.
- Changing C_i to $\vec{B}$, which gives different worst case execution times for the primary and each backup.
- Active backups as a property of each individual task, rather than having all tasks have the same number of backups. This would have included implementing the fancy math used to calculate the optimal number of backups for each task in a given taskset.
- Worst-case scheduability analysis (which would have been built off the previous point).
- Combining the two displays used so that tasks and cores are visible at the same time.
- (Other) Ensuring the structure of our codebase allows for expansion to other multi-core scheduling algorithms.

9. References

Baruah, S. K., Mok, A. K., & Rosier, L. E. (1990, December). Preemptively scheduling hard-real-time sporadic tasks on one processor. In [1990] Proceedings 11th Real-Time Systems Symposium (pp. 182-190). IEEE.

Risat, M. P. (2017). Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. Real - Time Systems, 53(1), 45-81. doi:http://dx.doi.org/10.1007/s11241-016-9258-z

E. N. Gilbert, "Capacity of a burst-noise channel," in The Bell System Technical Journal, vol. 39, no. 5, pp. 1253-1265, Sept. 1960. IEEE.