ISSN:0975-9646

Cache Related Pre-emption Delays in Probabilistic Real Time System

Prof. Minal V. Domke* , *Prof.* Pritee Saktel, *Prof.* Ekta Gupta
Shri. Ramdeobaba College of Engineering and Management,
Nagpur University

***Abstract*— In Real Time systems with cache, multiple tasks can share this common resource which can lead to *cache-related pre-emption delays* (CRPD) being introduced. CRPD is the additional cost incurred from resuming a pre-empted task that no longer has the instructions or data it was using in cache, because the pre-empting task(s) evicted them from cache. It is therefore important to be able to account for CRPD when performing schedulability analysis. This research focuses on the effects of CRPD on a single processor system, further expanding understanding of CRPD and ability to analyse and optimise for it. It present new CRPD analysis for *Earliest Deadline First* (EDF) scheduling that significantly outperforms existing analysis, and then perform the first comparison between *Fixed Priority* (FP) and EDF accounting for CRPD. In this comparison,the effects of CRPD across a wide range of system and taskset parameters are explored and a new *task layout optimisation* technique that maximises system schedulability via reduced CRPD.**

***Keywords*—EDF,SCHEDULABILITY,DELAY,CPRD,PRE-EMPTION**

1. INTRODUCTION

We are surrounded by e*mbedded* systems contained within larger devices, from medical pacemakers to the engine and control systems in large commercial aircraft. many of these embedded systems are also *real-time* systems that have specific deadlines that they must meet, and are often required to interact with an outside environment. it is therefore important that these real-time systems meet their temporal requirements, as well as being functionally correct. real-time systems can be categorised as *soft* and *hard* real-time. a soft real-time system can tolerate a moderate number of deadline misses, at the expense of reduced quality of service, such as in a live video streaming system. in contrast, a deadline miss in a hard real-time system would constitute a failure of the system. some hard real-time systems are also safety critical systems such that a deadline miss, and thus a system failure, could cause someone physical harm. most real-time systems are *multi-tasking* systems built up of a number of individual tasks. to verify the temporal behaviour of a multi-tasking system, the execution time of each task must be determined, and then combined together with information about the scheduling policy to ensure that there are enough resources to run all of the tasks that make up the system. this is usually achieved by performing *timing analysis* on the individual tasks, and then *schedulability analysis* on the system as a whole.

Cache Related Pre-emption Delays

In a pre-emptive multi-tasking system with cache, when a task is pre-empted, *cache-related pre-emption delays* (CRPD) can be introduced. CRPD is the additional cost incurred from resuming a pre-empted task that no longer has the instructions or data it was using in cache, because the pre-empting task(s) evicted them from cache. CRPD will be incurred as the task uses data and invokes instructions during the remainder of its execution that were evicted by the pre-empting task(s). CRPD is not a fixed cost per pre-emption, as is usually the case for traditional *context switch costs*, so simply subsuming an upper bound on the CRPD into the execution time of the pre-empting task could be very pessimistic. It is therefore important to accurately account for CRPD when performing schedulability analysis on a real-time system. There are techniques that can be used to reduce or completely eliminate CRPD, usually at the expense of increased task WCETs. For example, the cache can be partitioning so that each task has its own space in cache. However, Altmeyer *et al.* [3] recently noted that the increased predictability of a partitioned cache, in terms of eliminating CRPD, does not compensate for the performance degradation in the WCETs due to the smaller cache space per task.

1.1 Real-Time Scheduling

In real applications a system is usually built up of a number of tasks, collectively called a *taskset*. In addition to calculating the WCET of every task in isolation it is just as important to ensure that all the tasks, when running on the same platform and sharing resources, will meet their deadlines.

A scheduling policy is used to determine which task in the taskset should run at any given point in time. Scheduling policies can be classified as either *offline* or *online*. Offline scheduling, often referred to as static cyclic scheduling, uses a pre-computed schedule with very low runtime overhead. Online scheduling does not generate a schedule in advance, and instead determines which task should run at runtime. Under offline scheduling, the pre-determined schedule ensures that the schedulability of the system is known in advance. Sporadic jobs are more difficult to accommodate, but can be served using spare capacity.

Some classical online scheduling policies include:

 Fixed Priority (FP) [1] [2] - Fixed priority policy where tasks are allocated priorities offline and then scheduled according to those priorities at runtime .

 Earliest Deadline First (EDF) [2] - Dynamic priority policy where jobs with earlier absolute deadlines are given higher priorities. As the priorities are based on absolute deadlines of the individual jobs, task priorities change dynamically over the course of the schedule.

1.2 Schedulability Analysis

We now briefly cover existing schedulability analysis for FP and EDF scheduling assuming context switch costs are constant and subsumed into the tasks' execution times.

FP Scheduling

FP scheduling assigns each task a fixed priority which is then used as the priorities of the tasks' jobs. Under FP scheduling the sets of tasks that can pre-empt each other are based on the statically assigned fixed task priorities. Using the fixed priorities, we can define the following sets of tasks for determining which tasks can pre-empt each other: hp(i) and lp(i) are the sets of tasks with higher and lower priorities than task τi, and hep(i) and lep(i) are the sets containing tasks with higher or equal and lower or equal priorities to task τi.

The exact schedulability test for FP scheduling assuming constrained deadlines calculates the worst case response time for each task and then compares it to its deadline. The equation used to calculate Ri is :

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil C_j \qquad \text{(a)}$$

Equation (a) can be solved using fixed point iteration. Iteration starts with the minimum possible response time, Under FP there are a number of techniques that can be used to assign the fixed priorities. *Deadline Monotonic* [1] assigns higher priorities to tasks with shorter deadlines. *Rate Monotonic* [2] assigns higher priorities to tasks with shorter periods. Audsley's *Optimal Priority Assignment* (OPA) algorithm [14] takes a different approach. Using a greedy algorithm it evaluates the schedulability of each task, from lowest to highest priority, to devise an optimal priority for each task. It can be applied assuming the schedulability of a task meets certain conditions, such as not being dependent on the relative priority ordering of higher priority tasks. A drawback of OPA is that it selects the first schedulable priority assignment that it finds, which may result in a taskset that is only just schedulable. The *Robust Priority Assignment* (RPA) algorithm [8] improves on OPA by avoiding this drawback.

Assuming negligible pre-emption costs, Leung and Whitehead [1] showed that Deadline Monotonic priority ordering is an optimal priority ordering for constrained deadline tasks which can have synchronous releases. Rate Monotonic is an optimal assignment for tasks with implicit deadlines [2], and OPA can generate an optimal assignment for tasks with arbitrary deadlines and periodic tasksets with offset release times .

EDF Scheduling

In 673, Liu and Layland [2] gave an exact schedulability test that indicates whether a taskset is schedulable under EDF *if and only if* (iff) , under the assumption that all tasks have *implicit* deadlines ($Di = Ti$). In the case where $Di \neq Ti$ this test is still necessary, but is no longer sufficient.

Assuming negligible pre-emption costs, in 141 Dertouzos [4] proved EDF to be optimal among all scheduling algorithms on a uniprocessor. In 14, Leung and Merrill [5] showed that a set of periodic tasks is schedulable under EDF iff all absolute deadlines in the period $[0, \max\{si\} + 2H]$ are met, where si is the start time of task τi, $\min\{si\}=0$, and H is the hyperperiod (least common multiple) of all tasks periods.

In 690 Baruah *et al.* [6], [7] extended Leung and Merrill's work [5] to sporadic tasksets. They introduced $h(t)$, the *processor demand function*, which denotes the maximum execution time requirement of all tasks' jobs which have both their arrival times and their deadlines in a contiguous interval of length t.

$$h(t) = \sum_{i=1} \max\left\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right\} C_i \qquad \text{(b)}$$

1.3 Real-Time Systems and Cache

There are a number of features in modern processors that improve the average case performance, but make analysis of systems difficult due to the uncertainty that they introduce. These performance enhancing features include *caches*, *pipelines*, *branch predication* and *out-of-order execution*. When performing timing analysis they must be accounted for as they can affect the execution time of the basic blocks of code depending on what has been executed previously. Furthermore, in a pre-emptive multi-tasking system a pre-empting task can affect the execution time of a pre-empted task by altering the state of these hardware features, for example by evicting the contents of the cache. In this thesis we focus on analysing the effects caused by caches in real-time systems using pre-emptive multi-tasking.

Caches are small fast memories which are used to speed up access to frequently used blocks that reside in main memory, either RAM or permanent storage such as EPROM. CPU caches are either split into *instruction* and *data* caches, or combined into a *unified* cache.

Figure. a shows a simplified representation of a CPU, 4KB of cache and 4MB of EPROM that could be found in an embedded system. Only a small percentage of the data or instructions from memory can be stored in the cache at any point in time, but accesses to the cache require significantly fewer cycles. If the instruction or data resides in cache, then accessing it will result in a *cache hit*, if not, it will result in a *cache miss* and the instruction or data must be fetched from memory first.

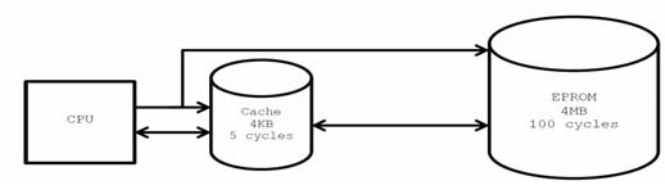

Figure .a - Layout of the CPU, Cache and EPROM Memory showing relative size and access times

Caches provide a predictable, but almost chaotic performance boost. Provided the current state of the cache is known, whether the next access will result in a hit or a miss can be calculated. However, it can be very difficult to keep track of the contents of the cache. Accessing data which is in the cache will always be faster than accessing data from memory. However, under some scenarios the time taken to execute a set of instructions that are in cache can even be slower than when the instructions are not in cache. This is referred to as a *timing anomaly* and is caused when other hardware features interact and result in additional blocks having to be loaded from the cache. This makes the ability to classify if a fetch will result in a hit or a miss even more important [12]. One solution is to simply disable the cache. However, as the demands of embedded systems increase it becomes increasingly cost ineffective to keep caches disabled as they can provide such a significant performance increase]. It is therefore important to be able to analyse systems with cache in order to verify todays' embedded systems.

Many aerospace systems partition different software systems so that they cannot interfere with each other. As caches are shared amongst everything running on a processor this is a cause for concern. CAST-7 [9] investigated caches in aerospace systems. In particular, it noted that "cache memory should receive special scrutiny in a partitioned system because the cache mechanism is not aware of the address partitioning architecture" [9]. This is a concern as the partitions are supposed to ensure that tasks in one partition do not affect another. However, as caches are not aware of the partitioning tasks in one partition can evict instructions and data belonging to a task in a different partition. This in turn can then affect the execution time of the other task, despite them being separated.

Another problem with cache and predicting its behaviour is that an empty cache is not always the worst case. For example, when the write back policy is being used on a data cache, blocks have to be written back to memory before they can be evicted.

An additional case where an empty cache is not the worst case is the *domino effect* [8]. The domino effect describes a situation where a repeating pattern of instructions cause the cache to transition through a number of states without converging. This could occur when a loop repeatedly calls a number of functions/instructions that are laid out in memory in a specific way. Due to the initial state and replacement policy, the cache does not end up in a consistent state, which means a different number of cache misses can occur on each loop iteration. Due to this effect, it must be assumed that the worst case number of cache misses occur on every iteration of the loop.

These factors combine together to make our ability to accurately analyse caches very important when verifying the temporal behaviour of real-time systems.

1.4 Timing Analysis

In order to determine if a taskset is schedulable when running on a multi-tasking system, it is essential to know how long each of the tasks could take to execute. This is achieved by performing timing analysis on the tasks. Timing analysis methods can be classified into three types of analysis; *static*, *measurement-based*, and a combination of the two *hybrid measurement-based* analysis. Static analysis calculates the execution time for blocks using a model of the hardware. Measurement-based analysis executes the software on the target hardware and records execution time measurements. Hybrid measurement-based analysis combines the two. It determines the execution times by measuring small sections of code, and then calculates a bound on execution time based on the program structure obtained using static analysis and the collected measurements. While this thesis does not focus on timing analysis, we present a brief review of the literature as it forms the basis for later work on the cache analysis required by CRPD analysis.

1.5 Static Analysis

Static WCET analysis aims to calculate an upper bound on the WCET by statically calculating what the execution time for each block of code will be, and then combining them together to find the *worst-case path* (WC path) through the code.

II INITIAL WORK

Early work on static WCET analysis was driven by the seminal paper by Puschner and Koza in 689 [13]. In [13], Puschner and Koza used source code to try to calculate an upper bound on the maximum execution time of tasks. Calculating an estimate for the WCET of an arbitrary program reduces to the *Halting problem* [11]. It was therefore apparent from the onset that a number of restrictions would have to be placed on the code in order to facilitate estimation by bounding the execution time. Some of those restrictions such as not using GOTOs and not having unbounded loops and recursive procedures are still present in today's techniques. In order to add additional information to the source code a number of high level path description constructs were defined. These were based on C like syntax and include things such as the ability to specify the maximum number of iterations for loops using bounds, and markers for dealing with multiple paths through loops. They proposed a set of formula, or timing schema, that could be used to combine together execution times for simple language constructions, assuming the execution time for them could be obtained. For example the execution time for a sequence of statements is the sum of the execution times for each statement. A downside of this approach is that it requires modifying the source code in ways such as replacing standard loops with their modified bounded versions.

2.1WCET Analysis Processes

Modern static WCET analysis uses IPET to express the analysis problem as an ILP that is solved by maximising an objective function to find the path with maximal length. The execution times of basic blocks are determined using very detailed and accurate hardware. There are different approaches that can be used to find and combine all the required information, but it is usually broken down into the following phases [14] . Reconstruction of the *call graph* (CG) and *control flow graph* (CFG), *architecture modelling* broken down into *pipeline analysis* and *cache analysis*, and *value analysis*. Finally, *path analysis,* which is the process of generating and solving an ILP, to compute the path through the program that maximises the execution time.

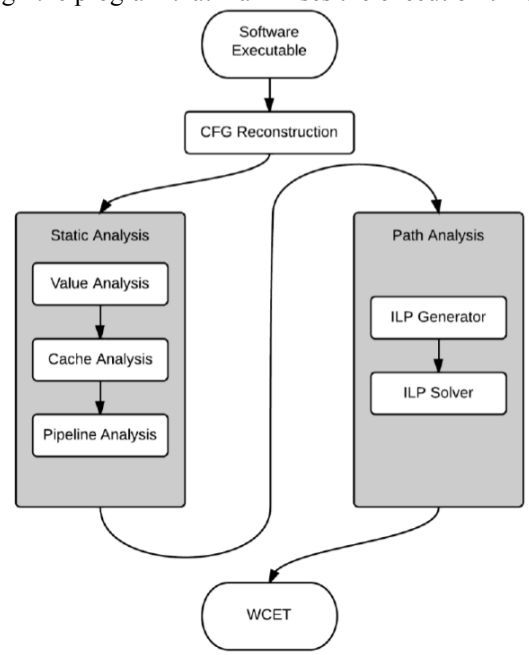

Figure .C - WCET analysis process for a typical static analysis tool

2.2 Cache Related Pre-emption Delays

When a pre-emption occurs there is a mandatory delay introduced by the need to save the state of the current task, decide which task to switch to, and then setup the new task. This delay is known as the *context switch cost* (CSC). As this is a fairly constant cost, it can usually be upper bounded and then *subsumed* into the execution time of the pre-empting task. In other words, in order to perform schedulability analysis on a taskset, the execution time of each task in the system is inflated by a bound on the time taken by the scheduler/operating system to switch to and then back from a task.

In a system with cache after a pre-emption occurs there can be additional costs due to interferences on the cache which affect the pre-empted task(s). This is known as *cache-related pre-emption delay* (CRPD) and it cannot simply be subsumed into the execution time of the pre-empting task without potentially introducing significant pessimism. This is because CRPD is dependent on the pre-empting and pre-empted tasks and the point of pre-emption. Specifically, it is incurred when a pre-empted task resumes and no longer has the instructions or data that the task was using in cache, because the pre-empting task(s) evicted them from cache. It is therefore difficult to determine the exact CRPD because the delay will not be incurred at once. Instead, CRPD will be incurred as the task uses data and invokes instructions that were evicted by the pre-empting task(s) during the remainder of its execution. In addition to being highly variable, CRPD can be significantly larger than CSC. In a study of a large multicore platform, Bastoni *et al.* [15] found the CSC to be around 5-10μs in the worst case, with variation being down to the number of tasks and scheduling policy which would not be changed at runtime. In comparison, they found the worst-case pre-emption costs to be much greater and more varied than the CSC, specifically they varied between 1-1300μs depending on the cache usage and system load. Figure C shows an example pre-emption with a small amount of CSC occurring when switching tasks and a large amount of CRPD spread out during the execution of a task after being pre-empted.

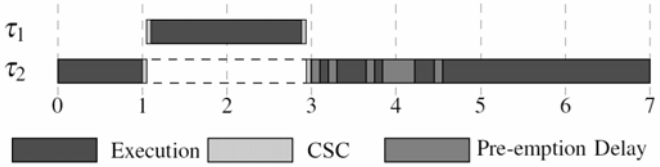

Figure D. Illustration of the effects of a pre-emption. CSC are incurred when switching tasks, and pre-emption delays are incurred during the remainder of a tasks execution after pre-emption as it accesses blocks that were evicted from cache during the pre-emption

As noted, the CSC is fairly constant and can be upper bounded and is therefore usually subsumed into the execution time of the pre-empting task. Figure E shows a revised version of Figure D with the CSC replaced by an increase to the execution time of task τ1.

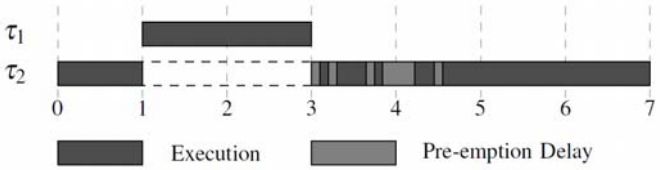

Figure E - Illustration of how the CSC can be subsumed into the execution time of the pre-empting task when compared to Figure D

2.3 CRPD Analysis for FP Scheduling

In this section, we review existing approaches for calculating CRPD when performing schedulability analysis for FP scheduling. To account for the CRPD when determining the schedulability of a taskset, a component is introduced into the response time analysis equation for FP, equation (a) , where represents the cost of a single pre-emption of task τi by task τj. This gives a revised equation for Ri as:

$$R_i^{\alpha+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^{\alpha}}{T_j} \right\rceil (C_j + \gamma_{i,j}) \qquad \text{(b)}$$

2.4 CRPD Analysis for EDF Scheduling

In this section, we review an existing approach for calculating CRPD when performing schedulability analysis for EDF scheduling. The EDF scheduling always schedules the job with the earliest absolute deadline first. Assuming negligible pre-emption costs, it is an optimal scheduling algorithm for a single processor. Any time a job arrives with an earlier absolute deadline than the current running job, it will pre-empt the current job. When a job completes its execution, the EDF scheduler chooses the pending job with the earliest absolute deadline to execute next. In the case where two or more jobs have the same absolute deadline, we assume the scheduler always picks the job belonging to the task with the lowest unique task index, see Figure F This has the benefit of minimising the number of pre-emptions. In the case where two task jobs have the same absolute and relative deadlines, it ensures that they cannot pre-empt each other. Furthermore, it ensures that after a pre-emption, the task that was pre-empted last is resumed first.

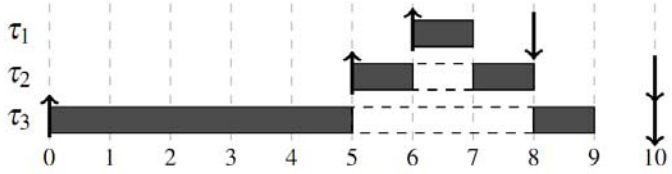

Figure F. Example schedule showing how the scheduler chooses which task should execute. Task τ3 is released at t = 0. At t = 5, task τ2 is released, pre-empting τ3 as although it has the same absolute deadline, it has a lower task index. At t = 6, task τ1 is released, pre-empting task τ2. At t = 7, τ1 completes, the scheduler then chooses to resume task τ2 as although it has the same absolute deadline as task τ3, it has the lower task index.

III CONCLUSION

The main contribution of this paper is a number of approaches for calculating *cache related pre-emption delay* (CRPD) in hierarchical systems with a global non-pre-emptive scheduler and a local pre-emptive fixed priority scheduler. This is important because hierarchical scheduling has proved popular in industry as a way of composing applications from multiple vendors as well as re-using legacy code. However, unless the cache is partitioned, these isolated applications can interfere with each other, and so inter-component CRPD must be accounted for, even if the cache is flushed after each global context switch.

REFERENCES

[1] J. Y.-T. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, vol. 2, no. 2, pp. 237-250, 682.

[2] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 7, no. 1, pp. 46-126, January 673.

[3] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Evaluation of Cache Partitioning for Hard Real-Time Systems," in *Proceedings 26th Euromicro Conference on Real-Time Systems (ECRTS)*, Madrid, Spain, 714, pp. 15-26.

[4] M. L. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," in *Proceedings of the International Federation for Information Processing (IFIP) Congress*, 141, pp. 807-813.

[5] J. Y.-T. Leung and M. L. Merrill, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters*, vol. 11, no. 3, pp. 115-118, 680.

[6] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptive Scheduling Hard-Real-Time Sporadic Tasks on One Processor," in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS)*, Lake Buena Vista, Florida, USA, 1990, pp. 182-190.

[7] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301-324, 1990.

[8] R. I. Davis and A. Burns, "Robust Priority Assignment for Fixed Priority Real-Time Systems," in *Proceedings 28th IEEE Real-Time Systems Symposium (RTSS)*, Tucson, Arizona, USA, 2009, pp. 3-14.

[9] Certification Authorities Software Team (CAST), "CAST-20 Addressing Cache in Airborne Systems and Equiptment," Position Paper 2003.

[10] A. Colin and S. M. Petters, "Experimental Evaluation of Code Properties for WCET Analysis," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 2003, pp. 190-199.

[11] R. Kirner and P. Puschner, "Transformation of Path Information for WCET Analysis During Compilation," in *Proceedings of the 13th Euromicro Conference of Real-Time Systems (ECRTS)*, 2001, pp. 29-36.

[12] T. Lundqvist and P. Stenstrom, "Timing Anomalies in Dynamically Scheduled Microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, Phoenix, AZ , USA, 1999, pp. 12-21.

[13] P. Puschner and C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs," *The Journal of Real-Time Systems*, vol. 1, no. 2, pp. 159-176, 1989.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, April 2008.

[15] A. Bastoni, B. Brandenburg, and J. Anderson, "Is Semi-Partitioned Scheduling Practical?," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, Porto, Portugal, 2011, pp. 125-135. [Online]. Extended version: http://www.cs.unc.edu/~anderson/papers/ecrts11-long.pdf