4-24-2008

Proceedings Work-In-Progress Session of the 14th Real-Time and Embedded Technology and Applications Symposium, 22-24 April, 2008 St. Louis, USA

Ying Lu
University of Nebraska-Lincoln, ying@unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

http://digitalcommons.unl.edu/csetechreports/1

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.
Proceedings
Work-In-Progress Session
of the 14th Real-Time and Embedded Technology and Applications Symposium

22-24 April, 2008
St. Louis, USA

Organized by the
IEEE Technical Committee on Real-Time Systems

Edited by Ying Lu

© Copyright 2008 by the authors
The Work-In-Progress session of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08) presents papers describing contributions both to state of the art and state of the practice in the broad field of real-time and embedded systems. The 25 accepted papers were selected from 27 submissions. This proceedings is also available as University of Nebraska–Lincoln Technical Report TR-UNL-CSE-2008-0003, at


Special thanks go to the General Chairs – Scott Brandt and Frank Mueller and Program Chairs – Chenyang Lu and Christopher Gill for their support and guidance. Special thanks also go to the Work-In-Progress Program Committee Members – Zonghua Gu, Kyoung-Don Kang, Xue Liu and Shangping Ren for their hard work in reviewing papers.

Ying Lu

Work-In-Progress Chair
14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)
# Table of Contents


F. Muhammad, B. M. Khurram, F. Muller, C. Belleudy, M. Auguin, *Precognitive DVFS: Minimizing Switching Points to Further Reduce the Energy Consumption.* 9


D. Zhu, A. Ş. Tosun, *Adaptive Path Scheduling for Mobile Element to Prolong the Lifetime of Wireless Sensor Networks.* 17

D. Luong, J. S. Deogun, S. Goddard, *Feedback Scheduling of Real-Time Divisible Loads in Clusters.* 21

J. S. Deogun, S. Goddard, *Developing New Models to Reason about Time and Space.* 25

D. Zöbel, *A Compositional Transformation to Bridge the Gap between the Technical System and the Computational System.* 29

C. Bartolini, E. Bini, G. Lipari, *Slack-based Sensitivity Analysis for EDF.* 33

A. M. Picu, A. Fraboulet, E. Fleury, *On Frequency Optimization for Power Saving in WSNs.* 37

R. Staudinger, *Towards Automatic Translation to Temporally Predictable Code.* 41

L. Qiu, N. Chen, S. Ren, *Checkpointing Implementation for Real-time and Fault Tolerant Applications on RTAI.* 45


G. Modena, L. Abeni, L. Palopoli, *Providing QoS by Scheduling Interrupt Threads.* 53

A. Anta, P. Tabuada, *On the Benefits of Relaxing the Periodicity Assumption for Control Tasks.* 57

J. Shamsi, M. Brockmeyer, *Mapping Overlay Networks for Real-Time Applications.* 61


H. Aysan, S. Punnekkat, R. Dobrin, *Adding the Time Dimension to Majority Voting Strategies.* 69

W. Wiles, G. Quan, *An Experimental Model for the Verification of Dynamic Voltage-Scaling Scheduling Techniques on Embedded Systems.* 73


M. Wilson, R. Cytron, J. Turner, *Partial Program Admission by Path Enumeration.* 97

Lan Yao, Fuxiang Gao, Xiuli Cui and Ge Yu
Collage of Information Science and Engineering
Northeastern University
Shenyang, China
{yaolan, gaoxfuxiang, cuixiuli, yuge}@ise.neu.edu.cn

Chao Shang
Collage of Software
Northeastern University
Shenyang, China
Shangchao@ise.neu.edu.cn

Abstract

Emerging applications such as forest fire monitoring have increasing demands on WSN to transmit data in real-time. In order to ensure real-time data transmission, it requires that the operating system of a node should schedule tasks in real-time. TinyOS is one of the most popular operation systems that support multifarious applications. However, its FIFO scheduling strategy does not guarantee requirements for hard real-time applications. A Two-Level Priority (TLP) Real-time Scheduling Strategy is proposed in this paper. Two tier priorities, static and dynamic, are designed and integrated in TinyOS task queue to guarantee the real-time task scheduling. We demonstrate this approach by a real-world case study: a WSN hardware node embedded with our task scheduling strategy is designed and implemented. The result demonstrates that our TLP real-time scheduling strategy performs efficiently in terms of packet throughput and task scheduling time.

1. Introduction

Different WSN practical applications require different level of real-time sensing data transmission. Some applications, such as environmental data monitoring in precision agriculture, tracking and monitoring of migratory birds, do not require too much about the real-time, While hard real-time scheduling is a demanding feature for many application domains. In real-world scenarios, real-time data transmission is critical to guarantee the quality of service. For example, forest fire monitoring and detection to invader for security.

The typical node system --TinyOS adopts non-preemptive FIFO task scheduling strategy. When the task queue is empty, the processor sleeps, and then the CPU is waken up to execute tasks by external events. Because all the tasks are equal, the priority of relative important or urgent task cannot be guaranteed. It is likely that over-load may even occur, which may cause the loss of task or the decrease of communication throughput. Thus the communication and operational efficiency based on this scheduling strategy of the whole system is limited.

This work is supported by the Major State Basic Research Development Program of China under grant (No. 2006CB303003).

Because TinyOS is a single task kernel, the resource utilization can be limited. Thus it is necessary to design a multi-task system. However, a multi-task system raises the issue of real-time task scheduling. In order to make sure the system is real-time, the task scheduling strategy based on preemptive priority is widely adopted. Research efforts have been contributed to the study of multi-task scheduling strategy based on priority. [1] expends TinyOS to multi-task scheduling, and adds multi-task scheduling function to improve the responsiveness of the system. [2] proposes a priority scheduling based on the deadline to improve the real-time capability of WSN system. [3] puts forward a task priority scheduling algorithm to improve the throughput of the over-loaded nodes and thus solve the over-load of local node packets.

All the above scheduling strategies improve some aspects of the original FIFO scheduling strategy, however they all have some shortcomings. For instance, the deadline-based priority algorithm only takes into account the real-time task. As a result, it is inadequate for the over-loaded situation. The task-based priority algorithm improves the node over-loaded issue; however it ignores the real-time requirements.

In order to solve the limitations discussed above in TinyOS and other typical algorithms, a Two-Level Priority (TLP) Scheduling Strategy is designed and implemented. This strategy enables TinyOS to respond to important and hard real-time tasks. We also provide a hardware system that realizes and embeds TLP. Our solution effectively prevents over-load in a node, and hence improves the overall communication efficiency of the system.

The structure of the paper is as follows: in section 2, the TLP scheduling strategy is introduced. The design and implementation of hardware system is given section 3. The performance of the TLP implemented on the hardware system is evaluated in section 4. Finally this paper concludes in section 5.

2. Design of TLP Scheduling Strategy

Communication routing task and local data processing task are divided according to different
functions of WSN tasks. Two relative static priorities 
(H, L) are given to these two types of tasks respectively, so that the over-loaded situation caused by the congestion of the local task emerging into the communication tasks can be processed.

Instead of the situation that a number of tasks share the same dynamic queue space caused by the FIFO scheduling strategy, different tasks occupy their own queues according to their levels. The task is running according to its queue space, and when the task is finished, the corresponding queue will be reclaimed and reallocated to new-arrival task. This makes the task processing more flexible.

At the mean while, based on the static priority, the dynamic scheduling strategy is adopted, called Earliest Deadline First (EDF). The task's dynamic priority is determined by the deadline and the running time of the task. The FIFO strategy is used when the tasks have the same priority. As a result, the capability of the running real-time is improved.

The TLP scheduling strategy includes task submission and task scheduling.

Task submission: when a new task arrives, the scheduling strategy judges whether the queue is full. If the queue is not full, as it is shown in Fig.1, it submits the new task to the tail of the queue, and sort the queue according to the dynamic priority. If the queue is full, as it is shown in Fig.2, it sorts all the tasks in the queue according to static priority (steady sort), and judge static priority and dynamic priority of the current task and the tail task. It inserts the high priority task into a proper place of the queue, and dismisses the current task. In Fig.1 and Fig.2, &\text{task3}_1 represents the third arrived task, which has high-level static priority; &\text{task2}_0 represents the second arrived task, which has low-level static priority. The task submission of TLP scheduling strategy is shown in Fig.3.

![Fig.1 Dynamic priority in half-full queue](image1)

![Fig.2 Static priority in full queue](image2)

![Fig.3 TOS_task_post](image3)

The hardware adopts Atmegal28L processor (details about this hardware will be introduced in section three), and takes the timer0 as the task-scheduling clock. At each time of the interruption, TLP updates the task's dynamic priority before schedule the task. The scheduling function chooses the task with the highest priority (the earliest deadline and the shortest running time) to execute. If the task is overdue, it will be dismissed, and a new head will be chosen. The timer0 is used in TinyOS as both the sleep/wake-up timer and the task-
scheduling timer. When the task queue is not empty, the timer0 is a task-scheduling timer, otherwise it is a sleep/wake-up timer. When the timer interruption arrives, TLP execute the EDF task scheduling. When the task queue is empty, CPU turns to the low-power sleep status. The task scheduling process is shown in Fig.4.

The TLP scheduling strategy effectively alleviates the concurrent overload situation, reduces the dropout rates, and thus improves the system throughput and communication efficiency.

3. Design and Implementation of Hardware System

The node hardware adopts modular structure design. It consists of data processing modules, wireless communication module, power supply module and sensor module.

The node hardware system should satisfy the following conditions: small volume; low power with sleep mode enabled; high integrity and fast speed.

BEGIN
Initialization; Close interruption;
If (cur_task_is_end())
{ if (queue_isnot_empty())
  {update_queue_param(); exchange_stacks();
   get_head_task(); move_head_pointer();
   while (cur_task_overtime())
     { discard_cur_task(); get_head_task();
       move_head_pointer();}
   if (interrupted())
     { save_states(); run_cur_task();
       open initialization; return 1; }
  }
  else {change_scheduling_to_sleep();
    open interruption; return 0; }
  else {update_cur_param();
    update_queue_param();
    if (curtask.dy_pri>headtask.dy_pri )
      { run_cur_task();open initialization;
        return 1;}
    else {savestates();
      exchange(&curtask,&hetask);
      while (cur_task_overtime())
        {discard_cur_task(); get_head_task();
         move_head_pointer();}
     if(interrupted()){save_states();
       run_cur_task();
       open initialization; return 1}} }
END

Fig.4 Task Schedule

3.1 Data Processing Module

The data processing module is the core part of the node. In order to meet the needs of low-power and small volume, we choose ATmega128L micro processor[4]. It has 4K byte EEPROM, 4K byte SRAM, 53 general I/O lines, 32 general work registers, real-time clock RTC, JTAG interface compatible with the IEEE1149.1 stander, and 6 energy-saving mode which can be selected by software.

SCM ATmega128L has limited data storage capability. So a manageable data storage chip is needed to store data.

512K serial FLASH AT45DB041B is used here to store data. Compared with common data memory, this chip has the features of low power consumption, small size, serial interface, and a simple external circuit. It is suitable for sensor nodes.

3.2 Wireless Communication Module

The wireless communication module should satisfy the requirements of low work voltage, low energy consumption and small volume.

We use RF CC2420 chip. It is a wireless transceiver module compatible with 2.4Hz IEEE802.15.4 stander[5]. It is provided with programmable output intensity and transceiver frequency. Its external circuit mainly includes crystal oscillator circuit, antenna and impedance matching circuit, interface circuit, and decoupling filter circuit. Its maximum transceiver rate is 250kbps. It guarantees the efficiency and reliability of short-distance communication.

In the wireless sensor node communication module, the selection and the deployment of the antenna will affect the quality of the whole wireless communication networks directly. This node RF chip CC2420 chooses metal inverted F PCD wired antenna and monopole antenna at the same time. The PCD wired antenna is a conductor printed on the circuit board, through which the antenna senses the airwaves and receives information.

3.3 Power Supply Module

The power supply module is a very important module in WSN. This module uses MAXIM company's MAX604 chip[6]. MAX604 chip has features including a low voltage difference, low energy consumption, and linear manostat to guarantee the stability of the system working voltage. As the time goes on, the voltage will decrease gradually. So it can not provide the system with a stable voltage. Thus, a MAX604 chip is added to the battery to keep the voltage at 3.3V.

The hardware structure of the node system is shown in Fig.5.
4. Performance Analysis

A TLP scheduling strategy is embedded in TinyOS and uploaded to nodes, which are implemented with ATmega128L and CC2420 in this paper. 100 nodes are deployed randomly in the 100m×100m monitoring area. It is evaluated and the following conclusions are derived from the empirical results. As it is shown in Fig.6, when FIFO strategy is used, with the increase of the local task running time, the number of data packets sent per second decrease sharply. While after the TLP scheduling strategy is adopted, the number of data packets sent per second stays in a stable amount. The result demonstrates that our scheduling strategy and its implementation improves the node throughout significantly.

5. Conclusions

The TLP scheduling strategy is proposed to divide task priority into two tiers: static and dynamic to guarantee the real-time task scheduling on TinyOS, and solves the limitation that TinyOS can not satisfy the hard real-time requirement. Based on the independent development of ATmega128L hardware node, TLP is implemented. Our empirical evaluations demonstrate the efficiency of this solution. It remains our future work to compare our solution with other scheduling strategy and to further validate our solutions to other hard real-time scheduling application.

References

Similarities between Timing Constraint Sets: Towards Interchangeable Constraint Models for Real-World Software Systems

Yue Yu and Shangping Ren
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{yyu8, ren}@iit.edu

Abstract—Traditionally, given two timing constraint sets, their relationship is defined by their timed trace inclusions. This approach only gives a boolean answer to if one set of constraints is contained within the other. In this paper, we first introduce a quantitative measure to describe the closeness or the similarity between two timing constraint sets. We intend to study the satisfaction bounds of similar timing constraint sets by similar timed systems. Such bounds will help improve the predictability of real-time systems in real-world applications and provide guidance for self-tuning systems.

I. INTRODUCTION

Software for real-world systems inevitably has to operate in an unpredictable environment and interacts with physical machineries. Hence, for most of these software systems, it is difficult and unrealistic to design and implement them in such a way that can be guaranteed to behave precisely as specified due to the following facts:

- **System Complexity** The ever increasing complexities of software systems have made guarantees of exact system behavior impractically expensive, if not impossible. For example, as pointed out by Lee [1], advances in computer architecture and software have made it difficult or impossible to estimate or predict the execution time of software. Moreover, networking techniques introduce variability and stochastic behavior.

- **Operating Environment** The intrinsically unpredictable nature of the environments in which software systems operate determines that even though software operates precisely as designed, its interactions with the outer world may not be totally expected. For example, [2] shows that several aircraft accidents have been attributed to “mode confusion”, where the software operated as designed but not as expected by the pilots.

- **Computational Intractability** From a theoretical point of view, achieving exactness in the verification of system properties is sometimes intractable. For example, [3] has shown that the satisfiability of a very simple class of real-time properties such as “every p-state is followed by a q-state precisely 5 time units later” turns out to be undecidable in a continuous model of time. On the other hand, several real-time logics are decidable under discrete approximations to the real time [4] or under interval timing constraints that prohibit infinite accuracy [5].

Therefore, basing our reasoning about systems’ timing properties and timing constraint satisfactions on precise information of real-world software systems is unpractical. Moreover, the traditional view of equivalence between timed systems and inclusion of timed trace set is hardly obtainable, neither sufficient. In fact, it is more practical and more accurate to allow impreciseness when modeling real-world systems. Similarity metrics have been studied recently [6], [7], [8], [9]. In parallel to the researches on similarities between timed systems, our ongoing studies focus on the similarities between timing constraint sets and their impacts on constraint satisfactions. The constraint similarity theories can be further applied to our non-intrusive, event-based feedback loop control system for enhancing legacy systems with self-protection features.

II. SATISFACTIONS OF SIMILAR CONSTRAINT SETS BY SIMILAR TIMED SYSTEMS

A. Similarities between Timed Systems

Timed systems model the sequence of system events and timing information of those events. However, since timed system models are approximations of the real world, achieving exactness in these models are unrealistic [10], [11].

Huang et al. [7] investigate the real-time property preservation between two similar timed state sequences (execution traces of timed systems), and extend the results to timed systems (a timed system is described and modeled by a set of timed state sequences). More specifically, the authors define the distance metric over timed state sequences $d_{\text{sup}}$, and the weakening function over real-time properties $R^{\mu}(\mu \in \mathbb{R}^+)$, such that

- **Relaxation Property of $R^{\mu}$**: For real-time property $\varphi$, $R^{\mu}(\varphi)$ is weaker than $\varphi$; and the larger $\mu$ is, the weaker is the real-time property $R^{\mu}(\varphi)$.

- **Robustness Property of $d_{\text{sup}}$**: Given two timed state sequences $\bar{\tau}$ and $\bar{\tau}'$ such that $d_{\text{sup}}(\bar{\tau}, \bar{\tau}') \leq \epsilon$, if $\bar{\tau}$ satisfies formula $\varphi$, the real-time property $R^{\mu}(\varphi)$ is preserved for $\bar{\tau}'$. 


The authors extend these results to concurrent real-time systems (with interleaving semantics) in [8]. However, in both papers, they do not provide algorithms to compute distances between systems, relying on system execution to estimate the bound.

Henzinger et al. [6] define quantitative notions of timed similarity and bisimilarity which generalize timed similarity and bisimilarity relations [12] to metrics over timed systems. The authors show that both the timed computation tree logic (TCTL) [13] and the discounted computation tree logic (DCTL) [14] are robust under the bisimilarity metrics, i.e., states similar under the metric satisfy specifications with similar timing requirements. They also give algorithms to compute the similarity distance between two timed systems modeled as timed automata to within any given precision.

B. Similarities between Timing Constraint Sets

In parallel to the works on similarity concepts for timed state sequences and timed systems, we study similarities between linear timing constraint sets which can be used to express timing requirements for timed systems. Traditional ways of comparing timing constraint sets search for exactness. Consider the following problem of timed trace inclusion.

Example 1: A timed trace of a set of real-time constraints can be represented as a timed data stream\(^1\) [15]. The set of all timed data streams satisfying a given set of real-time constraints is often infinite. However, it can be represented as a convex polyhedron in the affine space \(\mathbb{R}^n\) where \(n\) is the number of constrained event types. For example, Fig. 1 shows the trace polyhedron of the constraint set (1).

\[
\begin{align*}
  t(e_1) - t(e_2) &\leq 6, &\quad t(e_2) - t(e_1) &\leq 6, \\
  t(e_1) - t(e_3) &\leq 7, &\quad t(e_3) - t(e_1) &\leq 3, \\
  t(e_2) - t(e_3) &\leq 9, &\quad t(e_3) - t(e_2) &\leq 14
\end{align*}
\]

(1)

From Fig. 1, it is not hard to see that each plane representing a constraint is parallel to the vector \(z = (-1)x_1 + (-1)x_2 + (-1)x_3\), where vectors \(x_1, x_2,\) and \(x_3\) indicate time axes of independent events \(e_1, e_2,\) and \(e_3,\) respectively. Thus the circumscribed polyhedron is in fact a prism. In the figure, the pentagonal prism circumscribed by all but the plane representing the constraint \(t(e_3) - t(e_2)\) is included within that of (1) (bold lines), indicating that the computation restricted by constraint set (1) is more similar to (1) than computations restricted by constraint sets (1) and (3) are not mutually inclusive, constraint set (3) is more similar to (1) than (2). Therefore, computations restricted by constraint sets (1)

\[
\begin{align*}
  t(e_1) - t(e_2) &\leq 5, &\quad t(e_2) - t(e_1) &\leq 3, \\
  t(e_1) - t(e_3) &\leq 5, &\quad t(e_3) - t(e_1) &\leq 2, \\
  t(e_2) - t(e_3) &\leq 15
\end{align*}
\]

(2)

Now, consider another set of timing constraints:

\[
\begin{align*}
  t(e_1) - t(e_2) &\leq 5, &\quad t(e_2) - t(e_1) &\leq 3, \\
  t(e_1) - t(e_3) &\leq 5, &\quad t(e_3) - t(e_1) &\leq 2, \\
  t(e_2) - t(e_3) &\leq 15
\end{align*}
\]

To facilitate the discussion of trace inclusion, we show the planes of the constraints in the same affine space \(\mathbb{R}^3\) as in (1), and view the space axiometrically in the direction \(z = (-1)x_1 + (-1)x_2 + (-1)x_3,\) as shown in Fig. 2.

Fig. 2. Inclusion of two sets of timed data streams.

where the trace polyhedron of the constraint set (2) (light lines) is included within that of (1) (bold lines), indicating that the constraint set given in (2) is more stringent.

As mentioned in Section I, behavioral similarities is more practical than behavioral equivalence in real-world systems. Now, consider the computation restricted by the following set of timing constraints

\[
\begin{align*}
  t(e_1) - t(e_2) &\leq 5, &\quad t(e_2) - t(e_1) &\leq 7, \\
  t(e_1) - t(e_3) &\leq 5, &\quad t(e_3) - t(e_1) &\leq 2, \\
  t(e_2) - t(e_3) &\leq 10, &\quad t(e_3) - t(e_2) &\leq 5
\end{align*}
\]

(3)

The relationship between computations restricted by constraint sets (1) and (3) is illustrated in Fig. 3.

By comparing Fig. 2 and Fig. 3, we can see that although computations restricted by constraint sets (1) and (3) are not mutually inclusive, constraint set (3) is more similar to (1) than (2).

\(^1\)A timed data stream over an event set \(E\) is a pair \((a, \alpha)\) where \(a\) is a sequence with elements from \(E\) and \(\alpha\) is a monotonically increasing sequence with elements from \(\mathbb{R}^+ \cup \{+\infty\}\).
and (3) show more similarity than computations constrained by (1) and (2).

The proposed work on quantifying similarities between timing constraint sets is thus to

- define similarity metrics (e.g., percentage of intersection or maximum distance) that reflect observations and their geometric interpretations; and
- give efficient algorithms for calculating similarities under such metrics. Under some metrics, the problem can be intractable, e.g., calculating the percentage of intersection could have exponential cost. In these cases, approximation algorithms are needed.

Our previous study has shown that the set of timed data streams allowed by a set of real-time constraints does not change when constraints between all event pairs are replaced by implicit constraints derived by applying all-pairs shortest paths algorithms on the corresponding timing constraint graph. Based on this property, and the fact that the intersection of convex sets is still convex, an intersection between two sets of constrained timed data streams can be derived by forming the union of the constraint sets and applying all-pairs shortest path algorithms. Such intersections can be used for deriving a constraint set that satisfies both sets of constraints, and facilitating similarity comparisons of timing constraint sets. For example, in Fig. 3, the intersection of trace polyhedra of constraint sets (1) (bold lines) and (3) (light lines) is the hexagonal prism in dark gray; and similarity between (1) and (3) can be defined based on their closenesses to the intersection.

C. How Similarity Relations Commute

In Section II-A and II-B, we discuss similarities between timed systems and between timing constraint sets, respectively. These results, together with the existing results on satisfiability of timing constraints by timed systems, can be integrated to study the satisfaction bounds of similar timing constraint sets by similar timed systems. More specifically, assuming that timed systems $S_1$ and $S_2$ can be shown to differ by $\epsilon$ (by results in Section II-A), timing constraint sets $C_1$ and $C_2$ can be shown to differ by $\epsilon^2$ (by results in Section II-B), and $S_1$ satisfies $C_1$ with weakening function $R^\mu$ as mentioned in Section II-A$^2$, some interesting questions would be: (1) how does a replacement timed system $(S_2)$ satisfy the original constraint set $(C_1)$; (2) how does the original timed system $(S_1)$ satisfy a replacement constraint set $(C_2)$; and (3) how does a replacement timed system $(S_2)$ satisfy a replacement constraint set $(C_2)$?

![Fig. 3. The trace polyhedra of constraint sets (1) (bold lines) and (3) (light lines), and their intersection (the dark gray region).](image)

![Fig. 4. Satisfaction of similar timing constraint sets by similar timed systems.](image)

III. APPLICATION: A NON-INTRUSIVE APPROACH TO ENHANCE LEGACY EMBEDDED CONTROL SYSTEMS WITH CYBER PROTECTION FEATURES

We plan to apply the timing constraint similarity theories to the event-based feedback loop framework proposed in [16]. The framework is designed to externalize the cyber attack-tolerant logic out of the controlled system to allow for easier conception, maintenance, and extension of attack-tolerant behaviors. Under this architecture, a controlled system is monitored and compared with a system model that represents the essential components and their relationship with the controlled system to determine the health of the system. Fig. 5 depicts the high-level view of our proposed architecture.

As shown in the figure, the newly added protection logic is separated from the existing controlled system and its activation is only through event observations. In addition, the observation, reasoning, and action schemes are separated into independent modules. Such architecture allows us to change and incorporate different observation interests, reasoning schemes, and action strategies without much modification to the controlled systems or other modules.

More specifically, the external layer contains three modules, i.e., Observation, Evaluation, and Protection modules. These three modules communicate with each other through standard interfaces. The Observation module observes events generated by the controlled system and maps them into a high level abstraction so that the Evaluation model does not have to...
be tied with a specific system or system specific events; instead, the information will be provided to the Evaluation module with high level abstractions to promote the separation of reasoning logics from individual systems. The Evaluation module is responsible for reasoning about the controlled system from the information provided by the Observation module and decides if the controlled system is behaving normally. The Protection module interfaces with the controlled system and impose protective constraints on the physical units to prevent potential catastrophe.

Now, assuming that the Evaluation module carries a static set of timing constraints $C_1$ to be satisfied by the controlled system, and the Protection module carries a dynamic set of timing constraints $C_2$ that constantly changes in order to adjust the timing behavior of the controlled system. The consistency between $C_1$ and $C_2$ can be guaranteed by ensuring that the two sets do not differ by more than $\epsilon$. The satisfaction bounds mentioned in Section II-C can be used to improve the predictability of the system: the controlled system and the adaptive constraints in the Protection module may change, the system’s timing behavior always stays within acceptable ranges (bounded by the bounds) from the desired behavior specified in the Evaluation module.

### IV. Conclusion

Quantifications of similarities between timed state sequences and between timed systems have been studied. This paper presents our ongoing researches on similarities between timing constraint sets. Our preliminary results show the following:

- inclusion and intersection relations of timely constrained trace sets can be derived by applying all-pairs shortest paths algorithms on the corresponding constraint graphs; and
- intersections of timely constrained trace sets can be used to derive similarity metrics between timing constraint sets.

Our future research aims at:

- define similarity metrics between timing constraint sets;
- give efficient algorithms for calculating similarities under the metrics;
- study the satisfaction bounds of similar timing constraint sets by similar timed systems; and
- apply the theoretical bounds to our event-based feedback loop control system so that the predictability of the system can be improved.

### References


Precognitive DVFS: Minimizing Switching Points to Further Reduce the Energy Consumption

Farooq Muhammad, Bhatti M. Khurram, Fabrice Muller, Cecile Belleudy, Michel Auguin
LEAT, University of Nice Sophia-Antipolis,
CNRS France
{muhammad, bhatti, fmuller, belleudy, auguin}@unice.fr

Abstract—Dynamic Voltage Scaling (DVS) has been a key technique in exploiting the hardware characteristics of processors to reduce energy dissipation by lowering the supply voltage and operating frequency. The DVS algorithms are shown to be able to make dramatic energy savings while providing the necessary peak computation power in general-purpose systems. However, the algorithm used to dynamically change the voltage and frequency introduces a lot of unnecessary switching points (points where frequencies are varied). Increase in switching points not only increases the power consumption of the system but waste of processor cycles also increases. We, in this paper, propose an approach that minimizes switching points and reduce the cost of switch. This approach also ensures timeline guarantees for real time tasks.

I. INTRODUCTION

Power considerations have become an increasingly dominant factor in the design of both portable and desk-top systems. Energy dissipated per cycle with CMOS circuitry scales quadratically to the supply voltage, $P = \alpha C L V_{dd}^2 f$.

where $\alpha$ is the switching activity, $C_L$ is the load capacitance, $V_{dd}$ is the supply voltage and $f$ is the frequency.

An effective way to reduce power consumption is to lower the supply voltage level of a circuit. It usually prolongs the battery life but at the same time, real time constraints of the application are not guaranteed and hence it reduces the throughput. Recent trends in embedded hardware support multiple voltage and clock settings at the processor level. DVS technology is used to dynamically scale the voltage and frequency of the processor to reduce energy consumptions and achieve optimal energy management for embedded systems.

In time-constrained applications, often found in embedded systems like cellular phones and digital video cameras, DVS presents a serious problem. Changing the operating frequency of the processor will affect the execution time of the tasks and may violate some of the timeliness guarantees. RTDVS (real time DVS) algorithms not only minimize the energy consumption of the system but they also provide timeliness guarantees. However these algorithms introduce unnecessary switching points. Increase in number of switching points augments circuit delays and changing the operating frequency of the processor consumes energy itself.

This paper proposes an approach which minimizes the switching points and hence further reduces the energy consumption. We propose to decrease the frequency of the processor only at those instants after which processor will go idle, if frequency is not decreased.

In next section, we present DVS and related work. Section 3 presents the system model. Section 4 describes our approach. Conclusions are made in section 5.

II. RELATED WORK

DVS enables systems to operate under dynamically varied supply voltages, forms the basis for total power consumption reduction [7,8,9,10,11,12,13,14,15,16,17,19,20]. Since dynamic power is a quadratic function of the voltage, reducing the supply voltage and, therefore, the processor speed can effectively minimize the dynamic power consumption.

In terms of reducing the overall energy consumption, many newly developed scheduling techniques, e.g. Irani et al. [3], Jejurikar et al. [1,2], Niu and Quan [4,5], and Yan et al. [6], are constructed based on the DVS schedule. For example, Yan et al. [6] proposed to first reduce the processor speed such that no real-time task misses its deadline and then adjust the voltage supply and body biasing voltage based on the processor speed to reduce the overall power consumption. Irani et al. [3] showed that the overall optimal voltage schedule can be constructed from the traditional DVS voltage schedule that optimizes the dynamic energy consumption.

Pillai et al [22] has proposed two approaches: first, cycles conserving DVS minimizes energy cost but it increases unnecessarily switching points, second, look-ahead approach reduces the switching points but complexity is high to analyse the deferred work and to calculate the slow down factor.

III. PRELIMINARIES

In this section, we introduce the necessary notations and formulate the problem.

A. System Model

A periodic task set of $n$ periodic and independent real time tasks is represented as $\tau = \{T_1, T_2, ..., T_n\}$. A 4-tuple $T_i = \langle P_i, D_i, C_i, B_i \rangle$ is used to represent static parameters of each task $T_i$, where $P_i$ is the period of the task, $D_i$ is the relative deadline, $C_i$ is the worst case execution time, and $B_i$ is the best case execution time for the task. $AET_i$ represents the Actual execution time of task $T_i$.

All tasks are assumed to be pre-emptive. Each invocation of the task is called a job and the $k^{th}$ job of task $T_i$ is denoted as $T_i^k$. The tasks are scheduled on a single processor which supports multiple frequencies. Every frequency level has a

This research was supported in part by French national project PHERMA (ANR-06-AF), and in part by French Ministry for Research and Higher Studies.
power consumption value and is also referred to as power state of the processor.

IV. APPROACH DESCRIPTION

Although real-time tasks are specified with worst-case computation requirements, they generally use much less than the worst case on most invocations. To take best advantage of this, a DVS mechanism could reduce the operating frequency and voltage when tasks use processor time less than their worst-case time allotment. When the task completes, actual processor cycles are compared with the worst-case execution time. Any unused cycles that were allotted to the task would normally (or eventually) be wasted, idling the processor. Instead of idling for extra processor cycles, DVS algorithms are used that avoid wasting cycles by reducing the operating frequency for subsequent ready tasks. These algorithms are tightly-coupled with the operating system’s task management services, since they may need to reduce frequency on task completion, and increase frequency on task release. These approaches are pessimistic as they reduce frequency of the processor right after the completion of the task (if \( C_i < AET_i \)) and increase the frequency of the processor again when recently finished task releases again for next instant. They assume that these extra cycles will be wasted if frequency is not decreased right after the completion of a task. It may, unnecessarily, increase the switching points. Switching from one frequency level to other frequency level takes processor time and uses system energy and hence increases the power consumption of the processor.

In this algorithm, we try to minimize the switching points. We propose to accumulate the cycles (\( C_i - AET_i \)) and don’t decrease frequency until a point after which these cycles will be wasted, idling the processor, if frequency of the processor is not decreased.

\[
C_{i_{\text{rem}}} = C_i - C_{i_{\text{completed}}}
\]

\[
t_i + C_{i_{\text{rem}}} < r_j^e
\]

\( C_{i_{\text{rem}}} \) represents the remaining time of execution of task \( T_i \) and \( r_j^e \) is the earliest release time of task \( T_j \), \( 1 \leq j \leq n \), after time instant \( t_i \) and \( C_{i_{\text{completed}}} \) represents the fraction of \( C_i \) of task \( T_i \) which has been executed until instant \( t_i \). At this time instant, the frequency of the processor is decreased to extend the execution of task \( T_i \) until the \( r_j^e \).

\[
\alpha = \frac{C_{i_{\text{rem}}}}{r_j^e - t_i}
\]

\( \alpha < 1 \)

\[
f_{\text{new}} = \alpha \cdot f
\]

B. Calculation of slow down (\( \alpha \)) factor:

Once appropriate switching point is identified, frequency of the processor is decreased by a factor of \( \alpha \) (\( \alpha < 1 \)) which is calculated in following way:

C. Processor idling

Actual execution time of task \( T_i \) may vary from \( B_i \) processor cycles to \( C_i \) processor cycles.
If frequency of the processor (for a single task $T_j$) is decreased considering that task $T_i$ will take $C_j$ time units will cause a lot of cycles unused if its $AET_j$ appears to be much smaller than its $C_j$ (Figure 3b). In worst case, processor may go idle for processor cycles $(C_j - B_j)/\alpha$.

1) Minimizing processor idle time:

Accumulation of cycles is used to decrease the frequency of one task. Frequency of the processor may be very low when remaining execution time of task and wasted time will be increased as well.

There is a need to define an approach such that wasted time (idle time) on processor is minimized. This paper proposes to calculate the slow down factor by considering $B_i$ instead of $C_i$.

![Figure 3: Idle time on the processor](image)

In this case slow down factor $\alpha$ is calculated as:

$$B_i^{rem} = B_i - C_i^{completed}$$

$B_i^{rem}$ represents the remaining execution time of task $T_i$ while considering that $T_i$ will take $B_i$ time units to execute.

$$\alpha = \frac{B_i^{rem}}{r^i - t_i}$$

$$f_{rem}^{new} = \alpha . f$$

This may cause task $T_i$ to miss its deadline if its $AET_i$ comes out to be more than $B_i$. To ensure timeline guarantees for task $T_n$ there is a need to increase the frequency back to normal value at time $t_i$ before the earliest release time $r^i$ of task $T_i$. This time is calculated by folding back a part of task $T_i$ that was crossing the $r^i$ (Figure 4b).

$$t_i = r^i - t_i - \left(\frac{C_i - B_i}{l - \alpha}\right)$$

t$_i$ represents a time after which frequency of processors is restored to normal value i.e. slow down factor=1.

This approach has one possible drawback which is the cost of switching frequency of processor from very low value to normal (i.e. $\alpha = 1$).

D. Gradual Increase in Frequency:

The main reason to change the frequency (increasing) in gradual steps is the DVFS switching cost, which includes both time and energy cost. Switching cost is proportional to the magnitude of the switch.

![Figure 5: Gradual increase in frequency of the processor](image)

We propose to change the frequency in such a way that switching cost is reduced. Switching cost is low when frequency of the processor is changed from a high value to low value but cost is high in case of transition from low to high frequency. Moreover, switching cost also depends upon the size of step (difference between the current value of frequency and next value of frequency). That’s why we have proposed to increase the frequency in gradual steps until frequency is restored to normal value (i.e. until slow down factor = 1). This approach is similar to the approach explained in the above section with only difference that frequency of the processor is decreased when there are two (or more (optional)/auto adaptive) tasks in the ready queue of the scheduler. Slow down factor for first task (higher priority first task $T_j$) is selected to be higher than that for second task $T_i$. To achieve this, more accumulated cycles are allocated to first task than to second task.

$$CY_f = \frac{C^{rem}_{1}}{C^{rem}_{1} + C^{rem}_{2}} \times \left( r^e - t_j - C^{rem}_{1} - C^{rem}_{2} \right) + 0.5 \frac{C^{rem}_{2}}{C^{rem}_{1} + C^{rem}_{2}} \times \left( r^e - t_j - C^{rem}_{1} - C^{rem}_{2} \right)$$

$$CY_s = 0.5 \frac{C^{rem}_{2}}{C^{rem}_{1} + C^{rem}_{2}} \times \left( r^e - t_j - C^{rem}_{1} - C^{rem}_{2} \right)$$

$CY_f$ represents the cycles allotted to higher priority task ($T_j$) and $CY_s$ represents cycles allocated to task $T_i$. 

$$\alpha_f = C^{rem}_{1} / CY_f$$

$$\alpha_s = C^{rem}_{2} / CY_s$$
Higher priority task $T_3$ (figure 5b) is allocated more cycles to keep process frequency lower during execution of task $T_4$ than that for task $T_2$. Frequency for task $T_4$ will be higher than that of $T_3$ and it will be restored to normal value ($\alpha=1$) at $r_j^T$.

V. CONCLUSIONS

In this paper, we have extended the approach of dynamic voltage and frequency scaling scheme for multiple-clock-domain processors. The fundamental difference between this scheme and prior online DVFS schemes is in terms of minimizing switching points.

In addition, we have proposed an extension to our own approach which provides a trade off between number of switching points and cost of a switch. In this approach, cost of switch is minimized at the cost of increase in switching points.

We have analysed (with some manual example) that 30% switching points are decreased as compared to existing approaches. In future we are planning to simulate the algorithm on CoFluent Design [21].

REFERENCES


Towards Exploiting the Preservation Strategy of Deferrable Servers

Reinder J. Bril and Pieter J.L. Cuijpers

Technische Universiteit Eindhoven (TU/e), Department of Mathematics and Computer Science,
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands
r.j.bril@tue.nl, p.j.l.cuijpers@tue.nl

Abstract

Worst-case response time analysis of hard real-time tasks under hierarchical fixed priority pre-emptive scheduling (H-FPPS) has been addressed in a number of papers. Based on an exact schedulability condition, we showed in [4] that the existing analysis can be improved for H-FPPS when deferrable servers are used. In this paper, we reconsider response time analysis and show that improvements are not straightforward, because the worst-case response time of a task is not necessarily assumed for the first job when released at a critical instant. The paper includes a brief investigation of best-case response times and response jitter.

1. Introduction

Today, fixed-priority pre-emptive scheduling (FPPS) is a de-facto standard in industry for scheduling systems with real-time constraints. A major shortcoming of FPPS, however, is that temporary or permanent faults occurring in one application can hamper the execution of other applications. To resolve this shortcoming, the notion of resource reservation [8] has been proposed. Resource reservation provides isolation between applications, effectively protecting an application against other, malfunctioning applications.

In a basic setting of a real-time system, we consider a set of independent applications, where each application consists of a set of periodically released, hard real-time tasks that are executed on a shared resource. We assume two-level hierarchical scheduling, where a global scheduler determines which application should be provided the resource and a local scheduler determines which of the chosen application’s tasks should execute. Although each application could have a dedicated scheduler, we assume FPPS for every application. For temporal protection, each application is associated a dedicated reservation. We assume a periodic resource model [11] for reservations. Conceivable implementations include FPPS for global scheduling using a specific type of server, such as the periodic server [5] or the deferrable server [12].

Worst-case response time analysis of real-time tasks under hierarchical FPPS (H-FPPS) using deferrable servers has been addressed in [1, 5, 6, 10], where the analysis presented in [5] improves on the earlier work. Based on an exact schedulability condition, we showed in [4] that the analysis in [5] can be improved for a deferrable server at highest priority when that server is exclusively used for hard real-time tasks. In this paper, we reconsider worst-case response time analysis. We show that improving the existing analysis is not straightforward, because the worst-case response time of a task is not necessarily assumed for the first job when released at a critical instant. For illustration purposes, we consider a specific class of subsystems $S$ and an example subsystem $S \in S$. The paper includes a brief investigation of best-case response times and response jitter.

This paper is organized as follows. In Section 2, we briefly recapitulate existing results for our class of subsystems $S$ and introduce our example subsystem $S \in S$. This example clearly illustrated the potential for improvement. We investigate response times and response jitter for our example in Section 3, and conclude the paper in Section 4.

2. A recapitulation of existing analysis

In this section, we briefly recapitulate existing analysis. We start with a description of a scheduling model for our class $S$ and present our example $S \in S$. Next, we recapitulate the analysis for a periodic resource model [11], a periodic server [5], and a deferrable server [4], which we illustrate by means of $S$. We conclude with an overview.

2.1. A scheduling model

We assume FPPS for global scheduling, and consider a class of subsystems $S$ consisting of an application with a single, periodic hard real-time task $\tau$ and an associated
server \( \sigma \) at highest priority. The server \( \sigma \) is characterized by a replenishment period \( T^0 \) and a capacity \( C^0 \), where \( 0 < C^0 \leq T^0 \). Without loss of generality, we assume that \( \sigma \) is replenished for the first time at time \( q^\sigma = 0 \). The task \( \tau \) is characterized by a period \( T^\tau \), a computation time \( C^\tau \), and a relative deadline \( D^\tau \), where \( 0 < C^\tau \leq D^\tau \leq T^\tau \). We assume that \( \tau \) is released for the first time at time \( q^\tau \geq q^\sigma \), i.e. at or after the first replenishment of \( \sigma \). The worst-case response time \( WR^\tau \) of the task \( \tau \) is the longest possible time from its arrival to its completion. The utilization \( U^\tau \) of \( \tau \) is given by \( \frac{C^\tau}{C^0} \) and the utilization \( U^\sigma \) of \( \sigma \) by \( \frac{C^\sigma}{C^0} \). A necessary schedulability condition for \( S \) is given by \( C^\sigma = \sum_{\tau \in S} \lceil \frac{C^\tau}{C^0} \rceil (T^0 - C^\sigma) \). (1)

2.2. An example subsystem

For illustration purposes, we use an example subsystem \( S \in S \) with characteristics as described in Table 1. Note

\[
\begin{array}{ccc}
T & D & C^0 \\
\sigma & 3 & C^\sigma \\
\tau & 5 & 2
\end{array}
\]

Table 1. Characteristics of subsystem \( S \).

that \( \tau \) is an unbound task [5], because its period \( T^\tau \) is not an integral multiple of the period \( T^\sigma \) of the server. In this section, we are interested in the minimum capacity \( C^\sigma_{\text{min}} \) for the various approaches, where \( C^\sigma_{\text{min}} = \min \{ C^\sigma | WR^\tau \leq D^\tau \} \). Given (1), \( C^\sigma_{\text{min}} \geq U^\sigma \cdot T^\tau = 1.2 \).

2.3. Analysis for periodic resource model

Based on [11], we merely postulate the following lemma. Without further elaboration, we mention that we can postulate similar lemmas for the analysis of \( S \) based on the abstract server model in [6] and deferrable servers in [10].

Lemma 1 Assuming a periodic resource model for \( S \), the worst-case response time \( WR^\tau \) of task \( \tau \) is given by

\[
WR^\tau = C^\tau + \left( \left\lceil \frac{C^\tau}{C^0} \right\rceil + 1 \right) (T^0 - C^\sigma). \tag{2}
\]

Given (2), we derive for our example \( S \) that the minimum capacity for a periodic resource model is given by \( C^\sigma_{\text{min}} = 2 \). For this capacity, we find \( WR^\tau = 4 \).

2.4. Analysis for a periodic server

Strictly spoken, our class of subsystems \( S \) does not satisfy the model described in [5], because that article assumes that every set of tasks associated with a server contains at least one soft real-time task. Fortunately, a periodic server provides its resources irrespective of demand. As a result, the soft real-time tasks of a task set do not hamper the execution of the hard real-time tasks with which they share a periodic server. The analysis presented in [5] therefore equally well applies to \( S \) in general and \( \sigma \) in particular. For an unbound task, we derive from [5] that \( WR^\tau \) is given by

\[
WR^\tau = C^\tau + \left\lceil \frac{C^\tau}{C^0} \right\rceil (T^0 - C^\sigma). \tag{3}
\]

Without further elaboration, we mention that (3) also holds for the analysis of \( S \) based on a deferrable server in [11]. Given (3), we derive that \( C^\sigma_{\text{min}} = 1.5 \), giving rise to \( WR^\tau = 5 \).

2.5. Analysis for a deferrable server

The following theorem for \( S \) has been formulated in [4] as a corollary of a central theorem.

Theorem 1 Consider a highest-priority deferrable server \( \sigma \) with period \( T^0 \) and capacity \( C^0 \). Furthermore, assume that the server is associated with a periodic task \( \tau \) with period \( T^\tau \), worst-case computation time \( C^\tau \), and deadline \( D^\tau = T^\tau \), where the first release of \( \tau \) takes place at or after the first replenishment of \( \sigma \). The deadline \( D^\tau \) is met when the respective utilizations satisfy the following inequality

\[
U^\tau = \min U^\sigma \leq 1. \tag{4}
\]

Note that (4) is a necessary and sufficient (i.e. exact) schedulability condition for both the task and the server. Further note that (1) and (4) are identical, implying that a deferrable server is optimal for \( S \) when \( D^\tau = T^\tau \).

According to Theorem 1, \( S \) is schedulable using a deferrable server with \( C^\sigma_{\text{min}} = U^\tau \cdot T^\sigma = 1.2 \). The worst-case response time \( WR^\tau \) of task \( \tau \) is a topic of Section 3.

2.6. Overview

Table 2 gives an overview of the minimum capacities \( C^\sigma_{\text{min}} \) and minimum server utilities \( U^\sigma_{\text{min}} \) for the various approaches for \( S \) that guarantee schedulability of task \( \tau \). The table includes the worst-case response time \( WR^\tau \) of task \( \tau \) as determined by the various approaches.

<table>
<thead>
<tr>
<th></th>
<th>( C^\sigma_{\text{min}} )</th>
<th>( U^\sigma_{\text{min}} )</th>
<th>( WR^\tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>periodic resource model [11]</td>
<td>2.0</td>
<td>5/6</td>
<td>4.0</td>
</tr>
<tr>
<td>abstract server model [6]</td>
<td>2.0</td>
<td>5/6</td>
<td>4.0</td>
</tr>
<tr>
<td>deferrable server [10]</td>
<td>2.0</td>
<td>5/6</td>
<td>4.0</td>
</tr>
<tr>
<td>periodic server [5]</td>
<td>1.5</td>
<td>1/2</td>
<td>5.0</td>
</tr>
<tr>
<td>deferrable server [11]</td>
<td>1.5</td>
<td>1/2</td>
<td>5.0</td>
</tr>
<tr>
<td>deferrable server (this paper)</td>
<td>1.2</td>
<td>2/5</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Table 2. A comparison of approaches for \( S \).
3. On response times and response jitter

We will now explore the example in more detail by considering the worst-case response time, best-case response time, and response jitter of task $\tau$ of $S$ as a function of $\phi^i$ for a deferrable server with a capacity $C^\sigma = 1.2$.

3.1. Worst-case response times

Because the greatest common divisor of $T^\sigma$ and $T^\tau$ is equal to 1, we can restrict $\phi^i$ to values in the interval $[0,1)$. As illustrated in Figure 3, $WR^\tau$ is equal to 4.4 and assumed for $\phi^i = 0$, i.e. when $\tau$ is released at the start of the period of the deferrable server $\sigma$. Hence, a critical instant [7] occurs for $\phi^i = 0$. Figure 1 shows a timeline with the executions of the server and the task for $\phi^i = 0$ in an interval of length 15, i.e. equal to the hyperperiod $H$ of the server and the task, which is equal to the least common multiple (lcm) of their periods, i.e. $H = \text{lcm}(T^\sigma, T^\tau)$. The schedule in $[0,15)$ is repeated in the intervals $[(hH, (h+1)H))$, with $h \in \mathbb{N}$, i.e. the schedule is periodic with period $H$. From this figure, we conclude that capacity deferral of $\sigma$ is a prerequisite for schedulability of $S$ with a capacity $C^\sigma = 1.2$, and $S$ is therefore not schedulable with a periodic server with that capacity. We observe that the worst-case response time of the task is assumed for the 2nd rather than the 1st job. Hence, we need to revisit the notion of active period [2] in the context of H-FPPS to take account of this fact.

3.2. Investigating best-case response times

Unlike worst-case response times, we cannot restrict $\phi^i$ to values in the interval $[0, \text{gcd}(T^\tau, T^\sigma))$, but have to consider values in the interval $[0, T^\sigma)$ instead. This is caused by the fact that the response time of $\tau$ in the start-up phase can be smaller than the response time in the stable phase, as illustrated for $\phi^i = 0.8$ in Figure 2. Although the relative phasing of the 1st job of $\tau$ at time $t = 0.8$ compared to the 1st replenishment of $\sigma$ is identical to that of the 4th job of $\tau$ at time $t = 15.8$ compared to the 6th replenishment of $\sigma$, the response time of the 1st job $R^\sigma_1 = 3.0$ and of the 4th job $R^\sigma_4 = 3.2$. These differences in response times are caused by the fact that the execution of the 1st job is not influenced by earlier jobs, whereas the execution of the 4th job is.

Figure 1. Timeline for $S$ with a release of task $\tau$ at the start of the period of the deferrable server $\sigma$. The numbers at the top right corner of the boxes denote the response times of the respective releases.

Figure 2. Timelines for $S$ with a first release of task $\tau$ at $\phi^i = 0.8$ using a deferrable server $\sigma$.

Figure 3. Worst-case response times of task $\tau$ as a function of the first release time $\phi^i$.

Figure 4. Best-case response time of task $\tau$ during its lifetime as a function of $\phi^i$. The dashed line shows the shortest response time in the stable phase.

The best-case response time $BR^\tau(\phi^i)$ of $\tau$ is shown in
Figure 4. The dashed line in this figure shows for which values of $\varphi^5$ the shortest response time in the stable phase is larger than the shortest response time in the start-up phase. From this figure, we draw the following conclusions. Firstly, the best-case response time under arbitrary phasing is 2.0, which is equal to the computation time $C^5$ of $\tau$. Secondly, if we only consider response times of $\tau$ in the stable phase, the shortest response time becomes 2.6. Finally, $BR^5(\varphi^5)$ is determined by the start-up phase for phasings $\varphi^5 \in (0.6, 2.6)$.

3.3. Investigating response jitter

The response jitter of task $\tau$ as function of $\varphi^5$ is defined as

$$RF^5(\varphi^5) = WR^5(\varphi^5) - BR^5(\varphi^5).$$  (5)

The response jitter $RF^5(\varphi^5)$ is illustrated in Figure 5. Notably, $RF^5(\varphi^5)$ is constant in the stable phase.

![Figure 5. Response jitter of task $\tau$ during its lifetime as a function of $\varphi^5$. The dashed line shows the response jitter in the stable phase.](image)

4. Conclusion

Based on an exact schedulability condition, we showed in [4] that existing worst-case response time analysis of hard real-time tasks under H-FPPS can be improved when deferrable servers are used. In this paper, we investigated that identified opportunity to exploit the preservation strategy of deferrable servers. To that end, we considered a specific example subsystem with (i) a server used at highest priority and (ii) a period of its task that is not an integral multiple of the period of its server. For our example, the utilization of the server can be significantly reduced when using a deferrable server rather than a periodic server or assuming a periodic resource model. Given these initial results, application of a deferrable server can be an attractive alternative for resource-constrained systems with stringent timing requirements for a specific application when no appropriate period can be selected for its associated server. Unfortunately, improving the existing analysis turns out to be non-trivial, because the worst-case response time of a task is not necessarily assumed for the first job when released at a critical instant.

Using the same example, we briefly investigated best-case response times and response jitter. Unlike existing best-case response times of tasks under FPPS [3, 9], we did not assume infinite repetitions towards both ends of the time axis. As a result, the best-case response time of a task is determined by a start-up phase for specific phasings of the task relative to the server. When the start-up phase can be ignored, the best-case response time becomes larger and, correspondingly, the response jitter becomes smaller.

Improved response time analysis of H-FPPS using deferrable servers is a topic of future work, and we are currently re-investigating the notions of critical instant and active period in this context.

References

Adaptive Path Scheduling for Mobile Element to Prolong the Lifetime of Wireless Sensor Networks

Dakai Zhu and Ali Şaman Tosun
University of Texas at San Antonio
San Antonio, TX 78249
{dzhu,tosun}@cs.utsa.edu

Abstract

Mobile elements, which can traverse the deployment area and convey the observed data from static sensor nodes to a base station, has been introduced for energy efficient data collection in wireless sensor networks (WSNs). However, most existing solutions only calculate a single path for the mobile element, which may lead to quick energy depletion for sensor nodes that are far away from the path. In this paper, for real-time data collection in a WSN with one mobile element, we study the adaptive path scheduling problem for prolonging the lifetime of the WSN. Here, multiple paths are planned and the mobile element follows the paths in turn to balance the energy consumption on individual sensor nodes, thus to extend the WSN’s lifetime. We first illustrate the problem with one motivational example. Then, for cases where the movement of the mobile element is restricted (e.g., straight lines), we propose and analyze the optimal solutions. For the general cases, we discuss the issues involved and speculate our future research directions.

1 Introduction

In the recent past, the popularity of wireless sensor networks (WSNs) has been manifested by their deployment in many real-life applications (e.g., habitat study [4] and ecology monitoring [7]). With potentially a large number of sensor nodes scattered in a region of interest, the main problem in WSNs is how to efficiently aggregate the data at each node to a base station, which has the computational power to store and process all the collected data [1, 2]. Note that, sensor nodes are generally battery powered and it is hard (if not impossible) to replace those batteries after their deployment. Therefore, developing energy efficient data collection schemes is ultimately important.

In conventional WSN deployments, the data collection is normally achieved by using a multi-hop data forwarding mechanism. Here, for the nodes that are far away and cannot reach the base station in a single hop, the data will be relayed by the near to base station neighbors [1]. However, in this scheme, the energy budget for the nodes that are close to the base station will be quickly depleted due to their high data transmission activities and the lifetime of the WSN is rather limited.

To address this problem, mobile elements, which can move around the deployed field and convey the data from each sensor node to the base station, have been proposed [5, 6, 10]. The main problem in this scheme is how to control the mobility of the mobile elements for efficient data collection while satisfying various constraints (e.g., before buffer is full on each sensor node [6]). More recently, considering the constraint that the mobile element may not be reachable from every sensor node, the hybrid approaches that combine multi-hop and mobile elements have been studied [3, 9, 8]. Here, the data is first aggregated locally using multi-hop schemes to some rendezvous points. Then, the mobile element visits these points to pick the data up [9].

Note that, in the existing studies involving mobile elements, only a single path is calculated for each mobile element and the same path is followed repeatedly during data collection [3, 9]. However, such a solution with a single path may lead to uneven energy depletion rate for sensor nodes in WSNs. For instance, in WSNs where the mobile element collects data from each node directly, the nodes that are far away from the path will use up their energy budget quickly leading to limited lifetime for such WSNs.

In this paper, for real-time data collection in WSNs with a single mobile element that collects data directly from each sensor node, we study the adaptive path scheduling problem. Different from the single-path solutions, the key idea is to calculate multiple paths for the mobile element. During data collection, the paths are followed in turn to balance the energy consumption on individual sensor nodes, thus to extend the lifetime of the WSNs.
2 System Models and Assumptions

In this section, we first present the system models and state our assumptions. The WSN considered consists of \( n \) static sensor nodes that are deployed in the field, one base station and one mobile element. The position for the node \( N_i \) (\( i = 1, \ldots, n \)) is given as \((x_i, y_i)\), which is assumed to be known. The base station is located at \((x_0, y_0)\). Departing from the base station, the mobile element needs to travel through the field, collect data from each sensor node directly and return to the base station for conveying the collected data and recharging within a given time\(^1\) \( T \).

It is well-known that, for wireless communication between two nodes with distance \( d \), the transmission power \( P \) needed can be modeled as:

\[
P = \alpha d^\beta
\]  

(1)

where \( \alpha \) and \( \beta \) are system dependent parameters. Suppose that the mobile element follows a travel path \( PH \) during one round of data collection, the amount of energy consumed by node \( N_i \) for transmitting data to the mobile element can be calculated as \( E_i = P_i \cdot t = \alpha d_i^\beta t \), where \( d_i \) is the shortest distance from \( N_i \) to \( PH \) and \( t \) is transmission time. Assuming that the sensor nodes have the same sampling rate, the amount of data collected at each node will be the same during any time interval \( T \) and \( t \) will be a constant. The maximum transmission range at the maximum power level \( P_{\text{max}} \) is assumed to be \( d_{\text{max}} \), which limits the maximum distance from any node to the path of the mobile element.

Therefore, to minimize the energy consumption at each node, it is desired for the mobile element to visit the location of each and every sensor node. However, due to the time limitation \( T \), the length of \( PH \) will be limited by \( L = S \cdot T \), where \( S \) is the constant moving speed of the mobile element. Note that the lifetime of WSNs is limited by the node(s) consuming the highest amount of energy.

\(^1\)The time may be limited by the buffer size on sensor nodes, or the energy budget of the mobile element.

With the goal of maximizing the lifetime of the WSN, in this work, we study the path planning problem for the mobile element. Different from previous work, we focus on adaptive path scheduling, where multiple paths will be planned and are followed in turn by the mobile element to balance the energy consumed at each node.

3 One Motivational Example

We first illustrate the problem with one example, where 8 sensor nodes are placed on a 4 \( \times \) 4 grid field as shown in Figure 1(a). Here, the base station is located at \((0, 0)\) and the mobile element needs to follow the grid on the field. Suppose that the grid size is 1 and the path length limit of the mobile element is 10. It can be easily seen that it is not possible for the mobile element to visit each and every sensor node during one round of data collection.

<table>
<thead>
<tr>
<th></th>
<th>( PS_1 )</th>
<th>( PS_2 )</th>
<th>( PS_3 )</th>
<th>( PS_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_1 )</td>
<td>0.3t</td>
<td>0.3t</td>
<td>0.3t</td>
<td>0.3t</td>
</tr>
<tr>
<td>( N_2 )</td>
<td>0.3t</td>
<td>12.3t</td>
<td>9.3t</td>
<td>8.3t</td>
</tr>
<tr>
<td>( N_3 )</td>
<td>0.3t</td>
<td>0.3t</td>
<td>6.3t</td>
<td>8.3t</td>
</tr>
<tr>
<td>( N_4 )</td>
<td>0.3t</td>
<td>0.3t</td>
<td>3.3t</td>
<td>4.3t</td>
</tr>
<tr>
<td>( N_5 )</td>
<td>0.3t</td>
<td>0.3t</td>
<td>3.3t</td>
<td>4.3t</td>
</tr>
<tr>
<td>( N_6 )</td>
<td>12.3t</td>
<td>12.3t</td>
<td>9.3t</td>
<td>8.3t</td>
</tr>
<tr>
<td>( N_7 )</td>
<td>24.3t</td>
<td>12.3t</td>
<td>9.3t</td>
<td>8.3t</td>
</tr>
<tr>
<td>( N_8 )</td>
<td>12.3t</td>
<td>12.3t</td>
<td>9.3t</td>
<td>8.3t</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>48.3t</td>
<td>48.3t</td>
<td>48.3t</td>
<td>48.3t</td>
</tr>
</tbody>
</table>

Table 1. Energy consumed by the sensor nodes for transmitting data during 12 rounds of data collection with different sets of paths.

Suppose that a path \( PH_1 \) is calculated as shown in Figure 1(b). For illustration purpose, we assume that \( \alpha = 1 \) and \( \beta = 2 \). Moreover, the transmission energy for the nodes on the path is assumed to be negligible. For the schemes with the single path \( PH_1 \), after 12 rounds of data collection, the
energy consumption of each node for transmitting the data is shown in the second column (i.e., labeled as $PS_1$) in Table 1. Here, we can see that node $N_7$ consumes much more energy than other nodes.

Instead of always following the same path, we may calculate two paths ($PH_1$ and $PH_2$, as shown in Figure 1(c)) and the mobile element can follow them alternatively. In this case, the energy consumption of each node for 12 rounds of data collection is shown in the third column (labeled as $PS_2$) of Table 1. Suppose that the WSN can be operated until the first node uses up its energy, using two paths can effectively double the lifetime of the WSN as that of the single path option. Note that, the total energy consumed by all the nodes is the same as the previous case.

The lifetime of the WSN can be further improved when more paths can be exploited. The case with four paths is shown in Figure 1(d) and the fourth column of Table 1. Note that the paths may be followed differently by the mobile element for better performance. For the four paths in Figure 1(d), if $PH_1$ and $PH_2$ are followed once while $PH_3$ and $PH_4$ are followed twice in sequence, the corresponding energy consumption of the nodes is shown in the last column in Table 1. Here, the lifetime of the WSN can be tripled compared to that of the single path option. Again, the total energy consumption for all nodes is the same.

Therefore, to maximize the lifetime of the WSN, instead of minimizing the overall energy for all nodes [9], we should focus on minimizing the energy consumption on individual nodes. Another interesting observation from this example is that, for the nodes that are close to the base station (e.g., $N_1$), their energy consumption is much less since most paths pass by or are close to such nodes.

4. Adaptive Mobile Element Path Scheduling

Let’s first formally state the adaptive mobile element path scheduling problem with the assumption that the WSN can operate until the first node dies. For a given WSN with $n$ sensor nodes and one mobile element, finding the set of paths $PS = \{PH_1, \ldots, PH_k\}$ for the mobile element to:

$$\text{Minimize } \left( \max_{v_i} \frac{E_i \sum_{j=1}^{k} \alpha(d_j^i)^\beta}{k} \right)$$

subject to

$$|PH_j| \leq L, \forall j$$

$$d_i^j \leq d_{\text{max}}, \forall i \forall j$$

where $E_i$ is the average energy consumption for node $N_i$ for one round of data collection; $|PH_j|$ stands for the length of path $PH_j$ and $d_i^j$ is the minimum distance from node $N_i$ to path $PH_j$. $k$ is the number of paths to be calculated.

4.1 Restricted Paths

In some applications, the movement of the mobile element may be restricted [5]. In what follows, suppose that the mobile element can only move horizontally (i.e., in x-direction) along a straight line. We need to find the optimal path location (i.e., y-coordinate) for the mobile element, which will communicate with each node when they are vertically aligned.

For the case of $k = 1$ (i.e., a single path is used), $E_i$ will reach its maximum value at the sensor node(s) with maximum and/or minimum $y$ coordinates. Therefore, to minimize the maximum value of $E_i$, the optimal location for all paths will overlap at $Y_{opt}$. The proof is omitted due to space limitation.

**Theorem 1** Suppose that the movement of the mobile element in a WSN is restricted along the x-direction, the location of the optimal path for the mobile element to minimize the highest energy consumption among all nodes is $Y_{opt}$. ■

4.2 Unrestricted Paths

The problem of finding the general paths is similar to the traveling salesman problem (TSP) with neighborhood and is expected to be NP-hard. In this work, we focus on two different heuristic approaches for solving the problem. Denoted by shrinking path planning (SPP), one approach first constructs the complete path for the TSP which visits all nodes (the computational efficient MST approximation can be used). Then, nodes are removed from the path (with the constraint of Equation 4) one by one until the path length satisfies Equation 3. Starting from the opposite direction, the growing path planning (GPP) approach first finds a partial path by solving the TSP with a subset of nodes. Then the partial path is extended to make sure that the distance from the path to the remaining nodes satisfies Equation 4. If the path is still within the limit, the path can be further extended to reach the remaining node as close as possible.

Focusing on GPP approach, in what follows, we discuss both offline and online heuristic schemes.

**Offline Planning for k Paths:** To find out $k$ fixed paths offline, we can calculate them independently or iteratively. For the independent scheme, we first divide the $n$ nodes into $k$ seed subsets with each subset having $\lceil \frac{n}{k} \rceil$ seed nodes in it. This guarantees that each node serves at least as a seed node in one subset. For each of the seed subsets, a path will be calculated following the GPP approach, which will pass by the node in the subset while getting as close as possible to other nodes.

| 19 |
For a subset of seed nodes, suppose the initial partial path obtained is \( PH_i \). The detailed steps for adding a node \( N_j \) into the path are explained below. Depending on which point on the path \( PH_i \) has the minimum distance to \( N_j \), there are two cases.

The first case is shown in Figure 2(a), where the point is on one edge of \( PH_i \): \( N_i, N_{i+1}, N_{i+2}, N_{i+3}, \ldots \). If the distance from \( N_j \) to the edge \((N_{i+1}, N_{i+2})\) is no more than \( d_{\text{max}} \), we will ignore the node \( N_j \) during the first phase of extending \( PH_i \). Otherwise, the path has to be first extended to the point \( N'_j \), such that \( d(N_j, N'_j) \leq d_{\text{max}} \). Here, the path length will be increased by \( \delta = d(N_{i+1}, N_j) + d(N_j, N_{i+2}) - d(N_{i+1}, N_{i+2}) \). After extending the path \( PH_i \) to node \( N'_j \), the closest point from \( PH_i \) to node \( N_j \) can be illustrated as the second case in Figure 2(b).

Suppose that, after incorporating all the remaining nodes in the first phase, the current path length is \(|PH_i|\). If \(|PH_i| > L\), the construction of \( PH_i \) fails. Otherwise, during the second phase, we can further extend \( PH_i \) to get as close to node \( N_j \) as possible while making sure \(|PH_i| \leq L\). In the second phase of extending \( PH_i \), if \(|PH_i| + \delta \leq L\), path \( PH_i \) can be extended to include node \( N_j \) by adding edges \((N_{i+1}, N_j)\) and \((N_j, N_{i+2})\) while removing edge \((N_{i+1}, N_{i+2})\). Otherwise, if \(|PH_i| + \delta > L\), we can partially add node \( N_j \) by extending the path to a virtual node \( N'_j \). And the path \( PH_i \) will be \( \ldots, N_i, N_{i+1}, N'_j, N_{i+2}, N_{i+3}, \ldots \). The details on how to calculate the position of \( N'_j \) is omitted due to space limitation. So as the discussion on extending path \( PH_i \) for the second case shown in Figure 2(b).

For the iterative scheme, we can construct the first path by randomly selecting some seed nodes. After that, the selection of the seed nodes for constructing the \( i^{\text{th}} \) path will depend on the energy consumed by the nodes in the first \((i - 1)\) paths. Nodes that consumed the highest amount of energy have higher priority for being selected as seed nodes.

**Online Adaptive Path Planning/Scheduling:** Instead of using \( k \) predetermined paths, new paths can be computed on the fly at runtime. Using the same idea in the offline iterative scheme, a new path can be calculated based on the remaining energy consumption of the nodes. To amortize the cost of path computation, each path can be used for \( R \) rounds of data collection. Or, a new path is calculated whenever the remaining energy ratio between the sensor with most energy and least energy is above a certain threshold \( \tau \).

## 5 Conclusion and Future Work

Existing approaches using mobile elements for data collection in WSNs normally plan a single path, which may lead to quick energy depletion for sensor nodes that are far away from the path. In this paper, we introduce the adaptive path planning/scheduling problem, where multiple paths are planned and followed in turn to balance the energy consumption on individual sensor nodes, thus to extend the WSN’s lifetime. For cases with restricted movement of the mobile element, one optimal solution is analyzed. For general cases, different approaches to find multiple paths are discussed, where nodes with higher energy consumption are more likely to be on the constructed paths.

For our future work, we will consider cases where the lifetime of WSN can last until multiple nodes die. Moreover, adaptive path scheduling for hybrid schemes with multi-hop data forwarding will be studied.

## References


Abstract

Quality of Service (QoS) provisioning for divisible loads in clusters can be enabled using real-time scheduling theory, but is based on an important assumption: that the scheduler knows the execution time of every task in the workload. Information from production clusters, however, shows that estimated execution times of tasks are often inaccurate. Most of the work on scheduling divisible loads on clusters is based on this information, and therefore maybe of limited use when applied in practice. In this paper, we present our ongoing work to develop an EDF (earliest deadline first) scheduling algorithm with a feedback mechanism that is able to solve this problem. The objective of the new algorithm is to provide QoS provisioning of divisible loads when estimated execution times of tasks are inaccurate.

1 Introduction

Scheduling of arbitrarily divisible loads represents a problem of great significance for cluster-based research computing facilities such as the U.S. CMS (Compact Muon Solenoid) Tier-2 sites [5]. One of the management goals at the University of Nebraska-Lincoln (UNL) Research Computing Facility (RCF) is to provide a multi-tiered QoS scheduling framework in which applications “pay” according to the response time requested for a job [5].

Previous work on Quality of Service (QoS) provisioning for divisible loads in a cluster computing environment, however, is based on an important assumption: the scheduler needs to know the execution time of every task in the workload in advance. Scheduling decisions may be inefficient if this information is not accurate. Estimation of task execution time is a hard problem not only in real-time systems but also in general cases [6]. Although much work has been done to improve this estimation, there are always uncertainties in task execution times. In distributed systems, this problem becomes even harder because a task might be executed on multiple processors, and communication time should also be considered [1, 7]. Usually, the estimated task execution time is provided to the scheduler along with other task parameters. In most cases, this estimation is the worst-case task execution time, which is obtained empirically or based on expert knowledge of the task. Users who work with clusters tend to overestimate this value “just in case” their job runs longer.

We studied one year’s worth of logs for production jobs submitted to the Red and PrairieFire clusters1 at the University of Nebraska-Lincoln (UNL). We found that among jobs that finish successfully on both Red and PrairieFire clusters, the average execution times are only 9% and 18% of the estimates respectively. In Table 1, we show the number of overestimated and underestimated jobs. According to the current practice, most of the jobs exceeding their estimated execution times are killed. Log information shows that about 91% of such jobs on PrairieFire and 98% on Red, are killed, though these jobs consist of only 3% to 5% of the total number of jobs in a cluster.

<table>
<thead>
<tr>
<th>Number of jobs</th>
<th>Red</th>
<th>PrairieFire</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jobs run longer than estimated</td>
<td>6103</td>
<td>1370</td>
</tr>
<tr>
<td>Jobs run less than estimated</td>
<td>188545</td>
<td>26193</td>
</tr>
<tr>
<td>Jobs that finish on time</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jobs that are killed</td>
<td>5963</td>
<td>1240</td>
</tr>
<tr>
<td>Total</td>
<td>194648</td>
<td>27563</td>
</tr>
</tbody>
</table>

Table 1. Job statistics from two real clusters

QoS provisioning for divisible loads involves three components: an admission controller that decides to accept or reject an incoming task, a scheduler that schedules and partitions admitted tasks into subtasks, a dispatcher that sends the partitioned subtasks to the processors at their scheduled

---

1Red is a 111 node production-mode LINUX cluster, with each node containing two dual core Opteron 275 processors. PrairieFire is a 128 node production-mode LINUX cluster, with each node containing two (single core) Opteron 248 processors.
times. The scheduler makes decisions based on task parameters, such as execution time and deadline. If a task is admitted, it will be placed into the pending queue as a collection of subtasks and later dispatched by the dispatcher. One problem with this model is that once the schedule for a task (and its subtasks) is set, it is not changed. If nodes become available before the scheduled task start time, they are not used. The cluster processing capability is, therefore, wasted. Another problem is that the scheduler does not know how long a task will run after it runs past its allocated time. So, such tasks are generally killed to enforce the schedule. Task killing is, however, undesirable because the time the cluster spends on killed tasks is completely wasted.

We want to achieve the following goals when designing a real-time divisible load scheduling algorithm when execution times of tasks are different from their estimate. First, unused idle time when task finishes earlier than expected must be utilized, so that the system utilization is increased, and we can accept more tasks. Second, overrun tasks are killed only if necessary, i.e., when they cause other tasks to miss their deadline. Task real-time constraints should be guaranteed as long as their execution times are not underestimated. The new scheduling algorithm will be compared with the previous approaches by using simulations as well as experiments on a real cluster.

2 Task and System Models

To develop our scheduling algorithm, we use the same task and system models adopted in [2, 3, 4].

Task Model. A divisible task $T_i$ is denoted by the tuple $T_i = (A_i, \sigma_i, D_i)$ where $A_i$ is arrival time, $\sigma_i$ is data size and $D_i$ is relative deadline of the task. A workload consists of a set of independent tasks. A task is arbitrarily divisible, which means it can be partitioned into a set of subtasks, each of which processes a portion of the data. We use the vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ to denote the data distribution of a task where $n$ is the number of processing nodes assigned to such a task, and $\alpha_i$ is the data fraction allocated to the $i^{th}$ subtask, which means $\alpha_i \sigma$ unit of data is assigned to subtask $i$. We have $0 < \alpha_i \leq 1$ and $\sum_{i=1}^{n}(\alpha_i) = 1$.

System Model. The system consists of a cluster with a head node, denoted $P_0$, connected to $N$ processing nodes, denoted $P_1, P_2, \ldots, P_N$, via a switch. Every processing node in the cluster has the same computational capability and the same bandwidth on its link to the head node. We call such a cluster homogenous, as opposed to a heterogeneous one where computation and transmission capabilities of processing nodes are different from each other. The head node does not participate into the computation but takes the role of the admission controller, the scheduler and the dispatcher. By assumption, data transmission from the head node cannot be done in parallel. Only one processing node can receive data from the head node at a time.

Applying divisible load theory, transmission and computation time of a task is represented by a linear model. The transmission and computation time of $\sigma$ data units is given by $\sigma Cms$ and $\sigma Cps$. $Cms$ represents the time to transmit a unit of workload from the head node to a processing node. $Cps$ represents the time to compute a unit of workload on a single processing node.

3 Algorithms

3.1 Divisible Load Scheduling with Feedback

To develop our algorithm, we adapt the EDF-DLT algorithm [2]. The primary idea of EDF-DLT is to model a homogeneous cluster as heterogeneous and dispatch subtasks at the estimated available time of a processing node, so that the idle time in a cluster node can be better utilized. Recall that $P_1, P_2, \ldots, P_n$ denote $n$ homogenous processors. Assume node $P_i$ could start processing task $T$ at time $r_i$, for $i = 1, 2, \ldots, n$. We call $r_i$ the available time of $P_i$. It is either the time $P_i$ is released by a previous task or the time task $T$ arrives, whichever is latest. The $n$ nodes are ordered by their available times: $P_1$ is the earliest at time $r_1$ and $P_n$ the latest at time $r_n$.

Let $\hat{E}$ denote the task execution time when DLT is applied. $Cps_i$ and $Cms_i$ denotes the unit processing cost on node $P_i$ and $Cms$, denotes the unit transmission cost. Then, as shown in [2], for the heterogeneous model, we have the following:

\[
Cps_i = \frac{\hat{E}}{Cps} \\
Cms_i = \frac{\hat{E}}{Cms}.
\]

Tasks in a workload have the same $Cms$ and $Cps$ values, which are the estimated time to transmit and compute a single data unit of a task. The actual values, however, may differ from the estimated values.

When a task $T_i$ arrives, the scheduler calculates the minimum number of nodes to be assigned to $T_i$ so that it does not miss its deadline. As shown in [2], the execution time of a task, denoted by $\hat{E}$, is given by Equation (3),

\[
\hat{E}(\sigma, n) = \sigma Cms + \frac{\prod_{j=2}^{n} X_j}{1 + \sum_{i=2}^{n} \prod_{j=2}^{n} X_j} \sigma Cps
\]

where

\[
X_i = \frac{Cps_{i-1}}{Cms + Cps_i}, \text{ for } i = 2, 3, \ldots, n
\]

and the minimum number of nodes assigned to a task is given by:

\[
\tilde{n}_{min} = \left[ \frac{\ln \gamma}{\ln \beta} \right]
\]
where
\[ \gamma = 1 - \frac{C_{ms}}{A + D - r_n} \]  \hspace{1cm} (6)
and
\[ \beta = \frac{C_{ps}}{C_{ms} + C_{ps}} \]  \hspace{1cm} (7)

The data distribution vector is given as
\[ \sigma_1 = \frac{\sigma}{1 + \sum_{i=2}^{n} \prod_{j=2}^{i} X_j} \]  \hspace{1cm} (8)
and,
\[ \sigma_i = \frac{\prod_{j=2}^{i} X_j \sigma}{1 + \sum_{i=2}^{n} \prod_{j=2}^{i} X_j}, \text{ for } i = 2, 3, \ldots, n \]  \hspace{1cm} (9)

The results from [2] show that EDF-DLT is one of the best known scheduling algorithms for real-time divisible loads in clusters. This algorithm assumes that the estimate of task execution time is correct. However, if the actual values of \( C_{ms} \) and \( C_{ps} \) do not match the user’s estimate, tasks would either finish earlier or run past their estimated execution time. Since there is no feedback mechanism incorporated in the above algorithms, the scheduler has no means of knowing about these situations. This leads to idle time that is not utilized or tasks being killed because their allocated time expires.

We propose DLSwF, a DLT-based scheduling algorithm with a feedback mechanism, to handle these cases. Its goal is to better utilize the processing nodes and minimize the number of tasks that are killed. We use the following definitions to describe how DLSwF works:

- A task is said to “underrun” if its execution time is smaller than the estimated value. Most of the tasks on real clusters fall into this category. A task that underruns is called an underrun task.

- A task is said to “overrun” if its execution time is larger than the estimated value. A task that overruns is called an overrun task.

The general process of the DLSwF algorithm is shown in Pseudocode 1. It is based on four events in the system. The NewTaskEvent is invoked when a task arrives. We use the function AdmissionControl to check if we can accept the task or not. If it is accepted, this function generates the data distribution and the schedule for the task.

Due to the feedback module, the system is able to detect and handle the two events: OverrunTimerEvent and TerminationEvent. The first event is invoked when a subtask does not finish at its expected completion time. The second event is invoked when a subtask finishes its execution. The mechanism to handle these two events are described in Section 3.2.

### Pseudocode 1 DLSwF(Event)

```plaintext
1: if Event is NewTaskEvent then
2: call AdmissionControl to decide whether the task can be admitted or not
3: call GenerateSchedule to partition the task if it is admitted
4: else if Event is OverrunTimerEvent then
5: handle overrun and update nodes status
6: else if Event is TerminationEvent then
7: update nodes status
8: else if Event is DispatchTimerEvent then
9: //this event is handled by DispatchTask()
10: end if
11: call DispatchTask()
12: return
```

The DispatchTimerEvent is invoked when a subtask in the dispatching queue to be submitted.

After processing any of these events, the system invokes the DispatchTask function. This function is to dispatch a subtask in the dispatching queue, if any, to a processing node in the cluster. After dispatching a subtask, it will reset the DispatchTimer to the time when the next subtask should be submitted.

### 3.2 Handling Overrun and Underrun Tasks

Since the scheduler is not clairvoyant, it cannot know if a task underruns/overruns until its subtasks finish. Therefore, if a task overruns, it will be difficult for the scheduler to estimate the termination time of such a task in order to schedule the next tasks correctly. The nodes occupied by overrun tasks are considered to be blocked, or to have estimated finish times at \( \infty \). An overrun task can therefore severely affect the acceptance of new tasks and result in accepted tasks missing their deadlines.

Common practice on real clusters is to kill overrun tasks, the EDF-DLT algorithm also uses such an approach to ensure overrun tasks do not cause other tasks to miss deadlines or new tasks to be rejected. However, killing an overrun task is costly because the time the system has spent on that task is wasted and the task would have to be resubmitted later. Thus, our algorithm tries not to kill overrun tasks if it is avoidable. Still, deadlines of tasks that do not overrun should not be missed.

In the DLSwF algorithm, an overrun task is allowed to continue to run as long as it does not: (i) cause any already accepted task to miss its deadline or (ii) prevent a new task from being accepted.

Condition (i) says that when a task overruns, it should not cause any other tasks to miss their deadline, otherwise, the overrun tasks will be killed. Condition (ii) says that if a new task can only be accepted with the nodes occupied
by the overrun tasks then overrun tasks will be killed. Intuitively, this method works well in the case where the system is not heavily loaded. But when the system is very busy, the algorithm cannot prevent overrun tasks from being killed. If the two conditions are enforced, an admitted task will not miss its deadline unless it overruns.

The HandleOverrun function is described as follows. Assume that an overrun task occurs at time \( t \), we need to gather the following information in order to handle the situation:

- \( N_{OR} \): Number of nodes that have an overrun subtask.
- \( D_T \): Number of subtasks waiting to be dispatched at time \( t \).
- \( N_{AV} \): Number of available nodes at time \( t \).

It may be noted that \( N_{OR} > 0 \), \( D_T \geq 0 \) and \( N_{AV} \geq 0 \), since it is assumed that at least one overrun task exists.

Based on \( D_T \) and \( N_{AV} \), we evaluate the available time \( t' \) of blocked nodes to ensure that the schedule is being enforced. In other words, we need to determine when these nodes must finish their jobs. There are two cases:

- **Case 1:** \( 0 \leq D_T \leq N_{AV} \).
  - In this case, there are subtasks that must be dispatched at time \( t \), and sufficient nodes are available. Therefore, overrun tasks can continue to execute.

- **Case 2:** \( D_T > N_{AV} \).
  - In this case, a sufficient number of nodes are not available. However, we see that all subtasks do not start at the same time and thus some have to wait until others finish their data transmission. Therefore, if we order the subtasks in increasing order of their start time, we can let the overrun jobs continue to run until the \( k \)th subtask starts, with \( k = D_T - N_{AV} \).

As opposed to the overrun case, the solution for underrun tasks is relatively straightforward. The system knows immediately when a task underruns because of the feedback mechanism, i.e., TerminationEvent is detected before the expected completion time of a task. Therefore, it is able to update nodes status and if there is a pending task in the dispatching queue, this task will be dispatched immediately.

### 4 Conclusions and Future Work

In this paper, we address the problem of inaccuracy in the estimated execution times in the context of real-time divisible load scheduling. We present an approach to identify and handle overrun and underrun tasks. QoS and real-time constraints of the system are enforced by integrating the feedback mechanism into the scheduling algorithm. Our algorithm is expected to significantly improve the system performance with different levels of uncertainty in tasks execution time. We plan to consider the following issues when developing the algorithm: (i) applying historical knowledge of the workload to improve the admission control of the scheduler and (ii) detecting failure nodes in the cluster and reconfiguring the scheduler when nodes are added/removed from the cluster.

### References


Developing New Models to Reason about Time and Space

Jitender S. Deogun and Steve Goddard
Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE 66588-0115
{deogun,goddard}@cse.unl.edu

Abstract

Cyber-physical systems (CPS) tightly integrate physical processes with cyber-control and monitoring. The difference between CPS and traditional embedded systems lies in the degree of integration of software systems with physical systems, the scale and complexity of the integrated systems, and the reliance on sensing, computation, and actuation via networks.

New scientific foundations for specifying, designing, and implementing CPS will be needed before such systems will be true integrations of software and physical systems. As a first step in that direction, we extend and augment the notion of time bands, introduced by Burns et al., with space bands. We then briefly introduce the concept of Sigma bands, which are defined by the product of time and space bands. This proposed framework enables the formal specification of temporal and spatial properties and introduces tools for reasoning about activities that span multiple resolutions of time and space.

1 Introduction

A cyber-physical system (CPS) is a collaborative system of computing, sensing and actuating devices integrated with physical systems. In many such systems, correctness will be defined in terms of temporal and spatial properties. A primary problem experienced in building today’s embedded systems is that different portions of the system reason about time and/or space in different scales. A simple example in the temporal domain is the system clock. The hardware is capable of providing sub-nanosecond resolution. However, system time is generally kept at a 10 millisecond resolution. The resolution available to the application may be even more coarse-grained. A problem arises when different software modules interact, while operating at different temporal resolutions. This problem will become significantly worse in a CPS.

Burns et al. have identified this problem and defined the notion of time bands [1, 2]. Time bands provide a framework for reasoning about observable activities and events within a time granularity that is consistent with the activity of interest. They argue that time should be a central tenet of complex systems that model or reason about dynamic behavior, and provide a formalization of time bands and case studies that demonstrate the use of their time band framework.

We observe that similar problems arise when software and physical subsystems interact in the spatial domain, and with applications whose processing or correctness is dependent on both temporal and spatial conditions at the same time. The problem is not a trivial task of scaling temporal or spatial units at software interfaces. The challenge is in identifying and enumerating the bands required and then formalizing the abstractions needed to map activities from one band to another, while retaining the appropriate level of detail. Intuitively, this is similar to the scaling concept used in geographic information systems (GIS): as we zoom in on an area, more detail is revealed; and as we zoom out, less detail is presented.

As a first step in establishing scientific foundations for specifying, designing, and implementing CPS with temporal and spatial requirements, we extend and augment the notion of time bands, introduced by Burns et al. [1, 2], with space bands. We then briefly introduce the concept of Sigma bands, which are defined by the product of time and space bands. This proposed framework enables the formal specification of temporal and spatial properties and introduces tools for reasoning about activities that span multiple resolutions of time and space.

2 Model Framework and Formalization

A particular CPS, such as a smart environment for a hospital, could be specified and designed using the time band model described in [1, 2]. This method, however, will not be able to capture the dynamic spatial aspects of the envi-
environment, or the degree to which spatial constraints affect temporal aspects of the system. In such an environment, the system may be required to track and localize (i.e., in the spatial dimension) mobile assets, such as blood bags or wheel chairs, as well as personnel with varying degrees of accuracy and criticality. Moreover, localization in the spatial dimension may help the system adapt to the dynamic environment in the temporal dimension. Consider, for example, a scenario in which a hospital patient needs emergency attention. It will take less time for a secondary physician that is making rounds on the same floor to reach the patient than the primary physician who went to the cafeteria to get a cup of coffee.

A framework for specifying, designing, and implementing such a CPS must support both spatial and temporal properties. In addition, these scenarios indicate that the area of interest where the system must respond in a time critical fashion lies at the intersections of different, possibly independent, dimensions. The proposed concept of Sigma bands is developed to capture such design complexities.

In the remainder of this section, provide a brief overview of the time band model, as defined by Burns and Baxter in [2], but with two proposed modifications. Next, we introduce our concept of a space band model, which is inspired by the time band model. Finally, we propose the concept of our two-dimensional Sigma band model.

2.1 Time Band Model

The notion of a band is used in [2] to “define a strict temporal level in any system description.” This notion leads to time band specifications of a system that highlight the temporal structure of the system. That is, the time bands force a vertical temporal axis of system design onto a flat description. The functional properties of the system can be modeled at different levels of time band abstractions. The time band model is based on the following central concepts: Time-Band (t), Activities, Events, Precedence Relations, Clocks, Mappings, Granularity, and Behaviors. Each of these notions has associated algebraic properties that are used to formalize the model. Burns and Baxter provide a complete and formal specification of their model in [2] using Z notation. Due to space limitations, we refer the reader to [2] for more background on their time band model.

In this paper, we propose a generalization of the time band model by making two fundamental sets of changes in the model. The first is related to time bands and activities. The second change is related to durations and events. We briefly describe these changes next. Please note, however, that we present these extensions and the formalization of the space band model using set theory notation rather than Z notation in an effort to make the material more accessible to a wider audience. We do so, we believe, without losing expressibility or introducing ambiguity.

Following the notation of [2]: let $T$ denote a time band model; $B$ denote the set of time band identifiers; and $A$ denote the set of all instances of activities.

**Time bands and Activities:** A time band has a unique unit of time that is determined by its granularity. Figure 1 shows three different time bands with different granularities. An activity is a process or task that consumes time. Following [2] it may be noted that all changes in states of a system occur within activities. Unlike [2], however, we assert that any activity can be associated with one or more bands and can dynamically change its band. We propose this change because we believe the restriction of an activity to a single band artificially restricts the time bands that can be defined for a system.

Allowing an activity to be associated with more than one band, may result in the duration of an activity spanning more than one bands. The primary reason for this change is that we allow an activity to consist of a number of sub-activities where each sub-activity is associated with a unique band. Thus, under our proposed change, an activity $a$ is considered to be an ordered composition of one or more sub-activities $a_i$. An activity is said to be active in a band if one or more of its sub-activities are associated with the band.

Unlike [2], we assume that a band is associated with a possibly empty set of activities. This change is made to afford the dynamic nature of a CPS in which not all specific activities are known in advance, but the general notion of the activity is known. An example might be the tracking of objects in the environment, with the tracking rate being defined by the rate at which the object moves and the path the object takes.

![Figure 1. Three different time bands.](image-url)
Formally, let \( \mathcal{A} \) be a set of activities. Following our definition of an activity, \( a \in \mathcal{A} \), being an ordered composition of one or more sub-activities, \( a_i \), we have \( a = a_1, a_2, a_3, \ldots a_k \). Letting \( \mathcal{B} \) be the set of time band identifiers, as defined above, there exists some function \( \text{tband} \) such that an activity maps to a nonempty set of bands.

\[
\text{tband} : \mathcal{A} \rightarrow \mathcal{B}
\]

Thus, \( \forall a \in \mathcal{A}, \text{tband}(a) = \{ b \in \mathcal{B} : \text{tband}(a_i) = b \} \), for some subactivity, \( a_i \), of \( a \).

An activity is said to be associated with a timeband if one or more of its subactivities are associated with this timeband. It follows that there exists some function \( \text{activity} \) such that each time band is associated with some, possibly empty, set of activities that have one or more subactivities associated with this band.

\[
\text{activity} : \mathcal{B} \rightarrow \mathcal{A}
\]

Thus, \( \forall b \in \mathcal{B}, \text{activity}(b) = \{ a \in \mathcal{A} : \text{tband}(a_i) = b \} \), for some sub-activity, \( a_i \), of \( a \).

**Durations and events:** An activity, \( a \), has a length or duration, \( \delta(a) \), associated with it. In this paper we use length and duration interchangeably for the time bands domain. The length of an activity can be expressed in terms of the smallest granularity band to which it is associated or a combination of two or more consecutive granularity bands. To simplify the presentation, let us assume that three granularities will suffice. An activity of zero length is called an event. Let \( \mathcal{E} \) denote the set of all events in an application domain. More formally, we have

\[
\delta : \mathcal{A} \rightarrow \mathbb{N}, \ \mathbb{N} \times \mathbb{N}, \ \mathbb{N} \times \mathbb{N} \times \mathbb{N}, \text{where} \ \mathbb{N} \text{is set of natural numbers including zero.}
\]

\[
\mathcal{E} = \{ E \in \mathcal{A} : \delta(E) = 0 \}
\]

It may be noted that an event is an atomic activity and cannot be divided into sub-activities within a given band, though an event may map to an activity in a band with finer temporal granularity. Thus, an event corresponds to a unique band and \( \text{events}(b) = \{ E \in \mathcal{E} : \text{tband}(E) = b \} \) defines the set of events associated with the time band \( b \).

### 2.2 The Space Band Model

Let \( S \) denote a space band model. The formalization of the space band model is based on six basic notions: Space-Band (\( s \)), Feasible Path, Occurrence, Ruler, Granularity, Mappings and Behavior. These notions are somewhat akin to the time band notions.

1. **Space-Band (\( s \)):** A space-band (\( s \)) is defined by its granularity and determines the units of space for \( S \). In Figure 2, we show three different space bands, first with granularity of a yard, second that of a foot, and third that of an inch. In a domain where space is the only parameter under consideration, a system is composed of a partially ordered finite set of space-bands.

2. **Feasible Path:** A feasible path is a virtual line, connecting two objects, that does not cross any obstructions in the domain of the application. A feasible path is an ordered composition of feasible sub-paths where each sub-path is defined with respect to a specific space band, \( s \), and has a length (or distance) measured in terms of that space band. A feasible path is said to be associated with a space band \( s \) if one or more of its sub-paths are defined with respect to \( s \).

3. **Ruler:** Rulers are abstractions of measurement that define spatial frames of reference within a band. The units of the ruler are bounded below by the granularity of the band. Thus, measurements in a band are given in the units of length (or distance) of the band. The ruler of a band determines how precisely distance can be measured in the band.

4. **Position:** A position is a feasible path of zero length (or distance) in the ruler of the band.

5. **Mappings:** A mapping maps the positions (feasible paths of zero length) in a space band to feasible paths of possibly of zero length, in other space bands.

6. **Behaviors:** A behavior is a set of feasible sub-paths within a space band. The behavior in a space band give a partial specification of the system with respect to that band.
In this model, precision is defined as the minimum spatial distance between adjacent positions that can be recorded and stored. A unit of a feasible path is the shortest distance measurable in the units of space by the ruler for that band, which might be greater than the system precision.

For lack of space, we only give a sample of the formalism for the space band model.

**Space bands and Feasible Paths:** We let \( \mathcal{P} \) denote the set of all feasible paths and \( \mathcal{B} \) denote the set of space band identifiers. Following our definition of a feasible path, \( p \in \mathcal{P} \), being an an ordered composition of feasible sub-paths, \( p_i \), we have \( p = p_1, p_2, p_3, \ldots p_k \).

There exists some function \( sband \) such that each feasible path maps to a nonempty set of space bands.

\[
\text{sband} : \mathcal{P} \rightarrow \mathcal{B}
\]

Thus, \( \forall p \in \mathcal{P} \), \( sband(p) = \{ b \in \mathcal{B} : sband(p_i) = b, \text{ for some subpath, } p_i, \text{ of } p \} \).

Similarly, there exists some function \( feaspath \) such that each space band is associated with a possibly empty set of feasible paths.

\[
\text{feaspath} : \mathcal{B} \rightarrow \mathcal{P}
\]

Thus, \( \forall b \in \mathcal{B} \exists feaspath(b) = \{ p \in \mathcal{P} : sband(p_i) = b, \text{ for some sub-path, } p_i, \text{ of } p \} \).

**Lengths and Positions:** A path, \( p \in \mathcal{P} \), has a length, \( \text{length}(p) \), associated with it. The length of a path can be expressed in terms of the smallest granularity band to which it is associated or a combination of two or more consecutive band granualities. To simplify the presentation, let us assume that three granualities will suffice. A path of zero length is called a position. More formally, we have

\[\text{length} : \mathcal{P} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}, \text{ where } \mathbb{N} \text{ is set of natural numbers including zero.}\]

Let \( \pi \) be the set of all positions in the system. Then,

\[\pi = \{ p \in \mathcal{P} : \text{length}(p) = 0 \}\]

Similarly, there exists some function \( \text{position} \) such that each space band is associated with a possibly empty set of positions.

\[\text{position} : \mathcal{B} \rightarrow \pi\]

Thus, \( \forall b \in \mathcal{B} \) \( \text{position}(b) = \{ p \in \pi : \text{sband}(p) = b \} \).

### 2.3 Sigma Band Model

We now introduce the concept of our two-dimensional Sigma band model as a product of time and space bands. The Cartesian product of a time band model \( \mathcal{T} \) and a space band model \( \mathcal{S} \) generates a two dimensional Sigma band model \( \Sigma \), defined as \( \Sigma = \mathcal{T} \times \mathcal{S} \). The degrees of dependence in the \( \Sigma \) band model is two. The sigma band model is based on five basic notions: Sigma-Band (\( \sigma \)), Area_of_interest, Impulse, Region, and Granularity. These notions carry the somewhat different meanings as compared to the notions of time and space bands.

1. **Sigma-Band (\( \sigma \)):** A sigma-band is represented by its granularity and describes its units as an area defined as \( t \times s \). The finite set of \( \sigma \) bands constructs a system using partially order relations.

2. **Area_of_interest:** An area_of_interest is set of activities on feasible paths within the \( \sigma \) band. The area_of_interest may reflect the state changes and effects on a system environment, depending upon its selection. A movement \( \mathcal{LM} \) is an area covered by set of activities on feasible paths, \( A \times \mathcal{P} \).

3. **Impulse:** An impulse is an area_of_interest of zero duration and zero distance in a specific sigma band.

4. **Region:** Regions are abstractions of an area_of_interest within a specified band \( \sigma \). A region is an abstraction of nonempty and countable infinite sequence of impulses.

The formalization of Sigma band model is omitted for lack of space and will be presented later in a complete paper.

### 3 Conclusions and future work

In this working paper, we describe on going research in developing new models for reasoning about space and time. We present a generalized concept of time bands and propose a new concept of space bands. We also introduce a new concept of two dimensional Sigma bands that integrate time and space bands. The formalism of the framework presented may be used to capture complex interactions between the time and space dimensions. We are currently working on developing complete formalisms and properties of Sigma and space band models.

### References


A Compositional Transformation to Bridge the Gap between the Technical System and the Computational System

Dieter Zöbel
Institut für Softwaretechnik
Fachbereich Informatik
Universität Koblenz-Landau
Email: zoebel@uni-koblenz.de

Abstract

The majority of embedded applications with real-time constraints monitor and control a technical system. The correct behavior of such a system typically is described in terms of the technical system. In contrast the embedded hard- and software operates on an image of the technical system which is prone to deviations and delays. Therefore a compositional transformation is proposed which maps assertions specifying the behavior of the technical system to the program level conditions which guarantee for those assertions.

1 Introduction

Like any other human being a scientist is shaped by the community which he or she belongs to. The view on a problem scope and the way to find a solution is deeply inspired by the paradigms which are the common tenet to the respective communities. This phenomenon can be observed when scientists cooperate on a common subject area, e.g. electrical engineers and computer scientists. But even within scientific communities there are disparities of views, e.g. the time-triggered community and the event-triggered community inside the real-time community.

The subject area of embedded applications with real-time constraints is prone to this phenomenon. On one hand there is the hard- and software which builds up the computational system. On the other hand there is the physicality of the technical system which has to be monitored or controlled. The former is more in the focus of computer scientists, the latter more in the focus of engineers. They all have the common motivation to design and implement safe embedded applications by using mature engineering techniques.

Reflecting the state of the art in developing time-critical embedded applications there exists a myriad of mature but isolated techniques for certain questions which are relevant in the design process. One technique to be cited in this context is the worst case execution time analysis (WCET) which supplies indispensable parameters to enforce real-time properties. The execution times of processes are input to any real-time scheduling algorithm which itself builds upon a process oriented paradigm of programming. As a consequence of the variety of isolated techniques, various authors state that there is a strong need for holistic approaches, integrating the diversity by the establishment of a few essential paradigms. Such an approach cannot be a new level of abstraction on top of the existing techniques [3]. Instead it requires more or less a start from scratch.

As desirable as such a holistic approach may be, it is a long term option at the moment. In contrast, short term options have to be far more modest in that they should bridge the gaps which still exist between mature but isolated techniques. Furthermore, they should identify chains of techniques and tools which are able to support certain development processes for embedded applications. The structuring elements of this approach are the interface definitions between the joints of the chain. In its modesty this approach reveals where there are versatile techniques and tools, where they are weak, or where they are missing at all. Additionally it has to be noticed that high level techniques and tools which use certain assertions pretend that in turn these assertions can be easily propagated to lower levels of abstraction. Shortcomings of this kind can be observed for modelling languages and adjacent verification tools which apply to the technical system. So, it may be that the correctness of a system is proved using basic assertions about the technical system. However, the common verification techniques neglect that for completeness a profound sub-chain of techniques and tools which are needed to derive these basic assertions from lower level abstractions. Often those assertions have to be derived arduously from the level program code [7].
The following two sections show the basic ideas of a transformation technique which is able to bridge the gap between verification techniques applied to the technical system and the techniques of real-time programming applied to the computational system. The next sections presents a case study applying this bridging technique to a standard real-time application. Finally, there is a conclusion assessing the technique introduced and an outlook to further research efforts on this topic.

2 Bridging the gap

In the scope of real-time scheduling basic techniques towards the formulation of real-time conditions have been adopted from modelling techniques originally applied to database systems. This centers around the term consistency which in addition to a value based definition in the scope of database systems requires certain extensions referring to the time this data was created and the aging of this data when being used by real-time processes (see [1] and [5]). The two decisive definitions – absolute and relative temporal consistency – bound the absolute and relative time since the data has been taken from the technical system.

A generalization of this approach to determine real-time conditions distinguishes between the technical system, represented in terms of real-time entities, and its observation, namely real-time images [4]. A relation, called temporal accuracy, is defined for assigning the real-time image to some real-time entity within a bound history. Based on this knowledge the worst case error when utilizing this real-time image is estimated and can be taken into account for decisions which have to be made by the real-time process.

This paper wants to give a brief sketch, how the consequent extension of these approaches cited above results in a surplus value which consists in bridging the gap between a certain assertion $I$ necessary for the correct operation of the technical system and the coded control action $CA$ corresponding to the following program fragment:

```c
if (Condition) Action;
```

To explain the approach in more detail the viewpoint of a programmer developing a time-critical embedded application is adopted here. This viewpoint is program-centric in that the values of variables are processed and evaluated for decision making. Particulary in the scope of embedded systems several questions emerge from this view and unsettle the programmer:

- How precise is the value of a variable in correlation to the technical system?
- From which instant of time with respect to the technical system does the value of a variable stem?

Program code written under these circumstances is aggregated to processes which build up the computational part of the embedded system. These processes are executed concurrently following some real-time scheduling policy. Even though there is a profound theory of scheduling behind, the question remains what is the right control action $CA$ to satisfy property $I$ in the technical system.

![Figure 1. The reference architecture of an embedded system consisting of computational system which monitors and controls the technical system.](image)

3 Transformation of value domains

The computational system monitors and controls the technical system. Let $x$ and $y$ be physical entities of the technical system. Later in the case study $x$ will be the fuel level of a tank and $y$ a pump which can be switched to refuel the tank. The entities $x$ and $y$ have corresponding value domains $V_x$ and $V_y$. Typically sensors and actuators as in figure 1 introduce deviations in value and cause time delays. Additionally the infrastructure and the application processes of the computational system are responsible for further delays. Therefore an invariant property $I$ cannot be directly used as $Condition$ in the respective program fragment. Instead a transformation has to be applied which takes into account all deviations and delays:

$I \preceq C$

The operator $\preceq$ correlates value domains on one hand belonging to the technical system on the other hand to the computational system. E.g. in the following case study the

- At which instant of time a decision will be made by the program in execution?
- At which instant of time will the decision made by the program take effect in the technical system?
values \( V'_x \) satisfying invariant \( I \) are correlated to these observed values \( OV'_x \) of the computational system which guarantee for the validity of the invariant.

This correlation can be computed in a compositional way, which step by step takes into account all deviations and delays. E.g. the first step is build up by the sensor relation which models the falsifying behavior of the sensor:

\[
SR_x \subseteq V_x \times OV_x
\]

For some set of observed values \( OV'_x \subset OV_x \) it should be known which physical values may have caused them via the sensor:

\[
DOM(SR_x, OV'_x) = \{ v_x | (v_x, ov_x) \in SR_x \land ov_x \in OV'_x \}
\]

Analogously there is a respective relation \( AR_y \) on the actuator side.

A further operator, which is needed, has to predict what may happen in the technical system. This is captured by mapping \( TS \):

\[
TS_x : 2^{V'_x} \times \Delta T \rightarrow 2^{V_x}
\]

Let \( ov_x \) be some image value of the fuel level processed in the computational system. Applying the following mapping it is possible to derive all values \( V'_x \subset V_x \) which may have been read once before by the sensor system and after some delay are processed by some process as image value \( ov_x \). To compute what is put in the sentence above as after some delay includes the possible time interval starting from the earliest to the latest time this value may stem from. This time interval \([t_{\text{early}}, t_{\text{late}}]\) must include the processing time and henceforth depends on the policy of real-time scheduling.

\[
\bigcup_{\Delta t \in [t_{\text{early}}, t_{\text{late}}]} TS_x(DOM(SR_x, \{ov_x\}), \Delta t)
\]

Unfortunately this is not the operational structure which is needed from the viewpoint of program development. In typical applications the requirements in terms of \( I \) are given and the control action \( CA \), particularly the Condition has to be coded. So, the inversion of the formula above is needed which is explained in detail in [7].

4 A case study: Controlling the fill-level of a tank

To illustrate the transformation to find the correct Condition we refer an example of the fuel tank mounted near the jet engine in an airplane [2]. Let us assume that the fill-level of this tank should by guarantee never be less than some value:

\[
I \equiv v_x \geq 50 l
\]

Because the fuel of this tank is steadily consumed by engines there is a pump to refill this tank from other tanks. The status of the pump is determined by:

\[
pump_{on} \equiv v_y = 1
\]

As any other technical system our fill-level control system suffers from a lot of time- and value-dependant imprecisions. Control is possible only if some knowledge is available about the lower and upper bounds of these imprecisions. Let us assume to have the following knowledge:

- The vendor of the fuel-level measurement system guarantees that the value \( ov_x \) never deviates more than \( \pm 10\% \) from the value \( v_x \).
- The fill-level sensor is an independent device. When read by the process which executes \( CA \) the age of the fill-level value is somewhere between 10\( ms \) and 50\( ms \).
- Fuel is steadily consumed from the tank, with minimum consumption of \( 0.1l/s \) and in peak situations up to 20\( l/s \). So \( v_x \) is perishable between these largely differing rates.
- Process \( i \) responsible for the fulfillment of \( I \) is preemptive and periodic within the interval \( \Delta p_i = 150ms \).
- Finally the reaction by the actuation system has to be modelled. Here the assumption is that from setting \( OV_y \) until the instant of time that the pump is running lasts up to 350\( ms \). Conversely, there is no reaction at all of the pump before 70\( ms \).

This allows to calculate the lower and upper bounds for:

\[
\begin{align*}
t_{\text{early}} &= 10ms + 70ms = 100ms \\
t_{\text{late}} &= 50ms + 2 \times 150ms + 350ms = 700ms
\end{align*}
\]

Now we can derive Condition reversing the formula mentioned in the end of the last section.

1. We determine the set \( V''_x \subseteq V_x \) for which \( I \) holds:

\[
V''_x = \{ v'_x \in V_x | v'_x \geq 50l \}
\]

2. Next we determine those values \( V'''_x \) which have been sensed in some past \( t - \tau, \tau \in [t_{\text{early}}, t_{\text{late}}] \) and still satisfy \( I \) at time \( t \). From the deliberations above we know that any decision effecting the pump is based on sensed fill-levels \( v_x \) in the interval:

\[
100ms \leq \tau \leq 700ms
\]

So, the fill level minimally shrinks by

\[
100ms \times 0.1l/s = 0.1l
\]
and maximally increases by

\[ 700\text{ms} \times 20l/s = 14l \]

Taking into account the highest decrease we find \( V''_x = \{ v''_x \in V_x | v''_x \geq 64l \} \). This guarantees that after the longest evolvement of the technical process without control action the tank will still have \( v_x \geq 50l \).

3. Calculating \( OV'_x \) those values have to be included which by deviations of the sensor only can stem from \( v_x \in V''_x \). Since \( V_x \) contains scalar values and the imprecision is proportional to \( v''_x \) it suffices to concentrate on border values. So, we look for the smallest \( ov''_x \) such that the corresponding values \( v''_x \) are in \( V''_x \) and find this border value by multiplying the border value from above with the highest deviation:

\[ 64l \times 1.1 = 70.4l \]

In terms of relation \( SR_x \) we can assert that whenever \( ov''_x \geq 70.4l \) then all \( v''_x \) for which \( (v''_x, ov''_x) \in SR_x \) are elements of \( V''_x \).

4. We code the condition \( (OVx \geq 70.4) \) which finally guarantees that \( I \) holds under all value- and time-dependent imprecisions. Hence, the resulting transformation reads:

\[ (v_x \leq 50l) \Rightarrow (OVx \geq 70.4) \]

The control application presented in this case study, though it is still rather simple, demonstrates the essential steps to gain the right \( Condition \) to fulfill the specification \( I \). Different from our demonstration we have often the case that the periods are not known. Instead, there we may be interested in those functional dependencies which which determine the period (which is also a deadline here). So it may be that we code \( (OVx \geq 80) \) and ask for the consequences with respect to the periods. This situation can in turn be solved by using the period as parameter in the equations above, e.g instead of \( 14l \) we compute the maximum decrease as a function of \( \Delta p_i \):

\[ (400\text{ms} + 2 \times \Delta p_i) \times 20l/s \]

Continuing this calculation we obtain that the period should be less than \( 368\text{ms} \). This demonstrates the degree of freedom available with this canonical approach.

5 Conclusion and outlook

First of all this approach wants to be understood as a consequent and sophisticated enhancement of those papers which already have modelled value- and time-dependent imprecisions of real-time systems (e.g. [1], [4] or [6]). Even at the lowest level of abstraction – the coding of statements which interfere with the technical system to be controlled – the topics of scheduling and verification can be combined.

So, on one hand there is the verification at the level of programs. Here a property e.g. that \( Condition \) is evaluated in any period can be proved. This asserts that the correct \( Action \) is executed if necessary. On the other hand there is the verification at the level of the technical system. Here the basic assertion \( I \) regarding the fuel level is input for the deduction of higher level properties like the aeronautical stability of the plane. In this context the transformation \( I \models C \) bridges a gap between two important sub-chains of techniques and tools. At the same time the transformation has a upper interface in a value domain \( V \) and a lower level interface in the value domain \( OV \) which makes it independent of the lower and higher level verification tools.

The essential disadvantage so far is that the transformation has to be performed manually. Consequently, those relations and mapping that compose the transformation have to be identified and elaborated to generic building blocks which permit the automated derivation of correlations of the value domains. This would enhance both the top down design of embedded applications and the bottom up adaption and tuning of system parameters as it is needed in the scope of sensitivity analysis.

References


Slack-based Sensitivity Analysis for EDF

Cesare Bartolini, Enrico Bini, Giuseppe Lipari
Scuola Superiore Sant' Anna, Pisa, Italy
{cbartolini, e.bini, lipari}@sssup.it

Abstract—Real-time systems are characterized by several non-functional properties which are used to describe the temporal behaviour. Traditional schedulability analysis allows to determine whether the timing requirements are going to be met or not. On the other hand sensitivity analysis is also capable to measure the admissible variation to the non-functional properties. This is extremely important in practice since the non-functional parameters are often determined with a large margin of uncertainty.

The purpose of this paper is to lay a basis for the sensitivity analysis for EDF which is comprehensive of the three basic properties (computation times, deadlines and periods) using a common methodology.

I. INTRODUCTION

Real-time systems are generally constrained by timing requirements, and the key issue for designing such systems is predictability. Real-time theory allows the designer to know in advance whether a system will be able to fulfill its constraints. Clearly, this analysis requires a specific model for the system, based on some non-functional properties. These properties are the values which can be used to carry on a schedulability analysis.

This analysis faces two main problems. The first one is that computing these parameters can be difficult. Therefore, the estimate might be too distant from the reality. Then, one question is: What is the admissible range of variation?

Even if the parameter estimates are very accurate, there may still be other similar problems during the lifetime of the system. New releases, revisions, added features might all introduce some extra load on the system. But adding extra load might or might not exceed the system’s capacity. So the second problem is: How much could the non-functional properties be stretched before the system exceeds its feasibility limits?

The Sensitivity Analysis tries to address these questions. It is a relatively recent branch of real-time research which studies the amount by which task parameters can be modified remaining within the boundaries of feasibility.

This research aims at fully developing a methodology for EDF sensitivity analysis. However, the work is at its early stages, because, while the analytical expressions have been identified, there are still many issues to be addressed to reduce the complexity of the algorithm.

A. Related work

In the last years, there has been a growing interest in Sensitivity Analysis. The widespread usage of embedded systems with real-time properties, and the contextual need to reduce production costs, has created a new major branch in real-time research, aimed at maximizing the utilization of the processor. Initial researches focused on static priority algorithms [9], [6].

Some work has also been done on EDF schedulers, with particular attention to deadlines. For example, Balbastre et al. [2] and Hoang et al. [8] propose two solutions, based on the Processor Demand Criterion [3], [4]. Both proposals aim at finding the minimum deadline.

From the deadlines’ perspective, Bini and Buttazzo [5] developed a work aimed at describing the region of feasible deadlines. Although quite complex, it proposes a very elegant theory.

Some work on computation times has been done by Balbastre et al. in [1]. However, in that article the authors pose several constraints on the structure of the tasks, and the resulting expressions are quite complex. To the best of our knowledge, no additional work on EDF WCET sensitivity has been done.

Some preliminary analysis on EDF periods was done by Buttazzo et al. [7]. The problem of period sensitivity is that the feasibility test for EDF schedulers [4] requires to check a condition in a set of values which are dependent on the periods, up to the order of magnitude of the hyperperiod (see Section II). If the task periods change, both the set and the limit vary in a way which is not easily predictable.

The proposal of this preliminary work is to lay the basis for a sensitivity analysis which can be applied to all three parameters of a task, using a methodology which is uniform in the three cases. In particular, in this research we analyze sensitivity on the computation times using a methodology which is analogous to the one shown in [6] for fixed-priority schedulers, then follow along the same line (inversion of the feasibility condition) for the other two parameters.

II. EDF FEASIBILITY TEST

We assume that our system is running a set $\Pi$ of $N$ real-time tasks scheduled by EDF. The tasks are denoted by $\tau_1, \ldots, \tau_N$. Every task $\tau_i$ is characterized by a worst-case execution time $C_i$, an activation period $T_i$, and a relative deadline $D_i$.

Since our main purpose is to variate the task parameters, while still allowing the task set to be scheduled. Our analysis starts from a necessary and sufficient schedulability test [4].

Theorem 1 (from [4]): The task set $\Pi$ is schedulable by
EDF if and only if
\[ \sum_{k=1}^{N} \frac{C_k}{T_k} \leq 1 \tag{1} \]
\[ \forall L \in \text{dlSet}, \sum_{k=1}^{N} \left( \left\lfloor \frac{L - D_k}{T_k} \right\rfloor + 1 \right) C_k \leq L \tag{2} \]
where \( \lfloor \cdot \rfloor \) denotes \( \max \{ \cdot , 0 \} \), dlSet denotes a proper set of absolute deadlines defined as follows
\[ \text{dlSet} = \{ D_i + kT_i | \tau_i \in \Pi \land D_i + kT_i \leq H \land k \in \mathbb{N} \} \]
and \( H = \text{GCD}(T_1, \ldots, T_N) \) (often called hyperperiod of the task set in the literature).

By reversing this condition, we may stretch a feasible task set to its limit while preserving schedulability, or, in the case of a non-feasible task set, it is possible to discover the minimum amounts by which it should be relaxed to attain feasibility.

III. SENSITIVITY ANALYSIS

We are interested in the study of the variations of the following parameters for each task \( \tau_i \):

- the worst-case execution time (WCET) \( C_i \)
- the relative deadline \( D_i \)
- the period \( T_i \).

The purpose of sensitivity analysis is twofold: It can either be used to increase the processor load of a system with low CPU utilization up to the maximum affordable for the given task set, or it can reduce it so that an overloaded system becomes schedulable. We are currently focusing our research on tweaking a single property at a time.

For practical reasons, it is useful to introduce an addition expression, \( \nu_i \), which represents the number of instances of task \( \tau_i \) which can occur in the time window \( L \)
\[ \nu_i(L) = \left( \left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1 \right) \tag{6} \]

Additionally, the classical expression for processor utilization will be used throughout the paper:
\[ U = \sum_{k=1}^{N} \frac{C_k}{T_k} \tag{7} \]

A. Worst-case execution times

The objective is to have a schedulable task set when \( C_i \) is substituted with \( C_i + \Delta C_i \), so the maximum \( \Delta C_i \) which allows the task set to be scheduled is found. This analysis requires both Equations (1) and (2) to be fulfilled. From Eq. (1) we have:
\[ \frac{C_i + \Delta C_i}{T_i} + \sum_{k \neq i} \frac{C_k}{T_k} \leq 1 \Rightarrow \Delta C_i \leq (1 - U)T_i \tag{3} \]

The previous expression also shows that, if the task set without task \( \tau_i \) is already overloaded, then it is impossible to find a feasible solution (in that case, \( C_i + \Delta C_i \) would become less than zero).

The inversion of Equation (2) with respect to \( C_i \) is immediate.
\[ \forall L \in \text{dlSet}, \nu_i(L)(C_i + \Delta C_i) \leq L - \sum_{k \neq i} \nu_k(L)C_k \tag{8} \]

In this expression, it is possible that \( \nu_i = 0 \). This means that \( \tau_i \) will not be executed in the time window \( L \), so this value of \( L \) does not provide any useful information for sensitivity on this task. Therefore, such a situation can be excluded, allowing to remove the comparison with 0 in the following passage:
\[ \forall L \in \text{dlSet}, C_i + \Delta C_i \leq \frac{L - \sum_{k \neq i} \nu_k(L)C_k}{\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1} \Rightarrow \]
\[ \Delta C_i \leq \min_{L \in \text{dlSet}} \left\{ \frac{L - \sum_{k=1}^{N} \nu_k(L)C_k}{\left\lfloor \frac{L - D_i}{T_i} \right\rfloor + 1} \right\} \tag{9} \]

Combining Equations (3) and (4), the maximum value for \( \Delta C_i \) can be found as follows:
\[ \Delta C_i = \min \{ \Delta C_i^{\text{max}L} , \Delta C_i^{\text{max}U} \} \tag{10} \]

WCET modifications can also be computed for more than one task at a time, by using an approach similar to the one described in [6]. Let \( d = (d_1, \ldots, d_N) \) be the direction in the C-space along which WCETs are to be modified. The new vector of worst-case computation times becomes \( C + \lambda d \), where \( \lambda \) is the value we are attempting to maximize.

Let \( b = \left( \frac{1}{a_1}, \ldots, \frac{1}{a_N} \right) \) and \( C = (C_1, \ldots, C_N) \) be the vectors of the task rates and the WCETs, respectively. The condition of Eq. (1) can be expressed as follows:
\[ b \cdot (C + \lambda d) \leq 1 \Rightarrow \lambda_{\text{max}U} = \frac{1 - b \cdot C}{b \cdot d} = \frac{1 - U}{b \cdot d} \tag{11} \]

In Equation (2), the only variables are the worst-case execution times \( C_k \). The condition becomes:
\[ \forall L \in \text{dlSet}, a(L) \cdot (C + \lambda d) \leq L, \]
where \( a(L) = (\nu_1(L), \ldots, \nu_N(L)) \) is a vector representing the fixed parameters. From this follows:
\[ \lambda_{\text{max}L} = \min_{L \in \text{dlSet}} \left\{ \frac{L - a(L) \cdot C}{a(L) \cdot d} \right\} \tag{12} \]

Combining the two,
\[ \lambda_{\text{max}} = \min \left\{ 1 - \frac{U}{b \cdot d} , \min_{L \in \text{dlSet}} \left\{ \frac{L - a(L) \cdot C}{a(L) \cdot d} \right\} \right\} \tag{13} \]

Clearly, if \( d_i = 1 \) and \( \forall k \neq i, d_k = 0 \), only task \( \tau_i \) will be modified, and in this case Equation (13) becomes identical to (5).

B. The S function

To perform the sensitivity analysis for periods and deadlines it is convenient to introduce the following auxiliary function:
\[ S_i(L) = \frac{L - \sum_{k \neq i} \nu_k(L)C_k}{C_i} \tag{14} \]

A few considerations over the \( S \) function are in order. \( \nu_k(L)C_k \) is an upper bound to the execution time of the task \( \tau_k \)
in the time frame. By summing all these contributions except the one for task \( \tau_i \), which is the task on which the algorithm is operating, the result is the total execution time of the task set without \( \tau_i \). This sum is then subtracted from \( L \), giving the time up to \( L \) available for \( \tau_i \). By dividing this time by \( C_i \) (which is now a constant), we obtain the maximum number of instances of \( \tau_i \) which may be run in time \( L \) without disturbing the other \( N - 1 \) tasks.

Therefore, \( S_i(L) \) is the maximum number of instances available for task \( \tau_i \) in the time window \( L \). Note that this is a fractional number, while a conservative integer will be required.

To have a successful feasibility test, the condition in Equation (2) may now be rewritten as follows:

\[
\forall L \in \text{dlSet}, \quad \nu_i(L) \leq S_i(L)
\]  

(10)

The 0 index in \( \nu_i \) does not provide any useful information. If the left member is 0 or less, then task \( \tau_i \) will not be executed in the \( L \) time frame, regardless of its parameters. Note that this is not true for the other tasks (which figure in \( S_i(L) \)). By removing the floor function from the left member, the condition may be rewritten as follows:

\[
\forall L \in \text{dlSet}, \quad \frac{L - D_i}{T_i} \leq S_i(L) - 1 \Rightarrow \\
\forall L \in \text{dlSet}, \quad \frac{L - D_i}{T_i} < [S_i(L)] \quad (11)
\]

The passage from Equation (11) to (12) is the pivot of this analysis. While variations of \( C_i \) do not introduce particular problems, the fact that \( D_i \) and \( T_i \) are contained within the floor function requires special care. Particularly, the inversion of the floor function (which in itself, from a mathematical point of view, is not immediate) will change the “less than or equal to” relationship to strictly less, with the consequence of excluding the boundary values from the feasible solutions.

This condition can then be used to evaluate the sensitivity of the parameters.

Equation (12) can be used as a starting point for computing the sensitivity of the two remaining parameters. However, they parameters have different behaviors with respect to this equation. In particular, changing them affects not only the dlSet, but in the case of the period even the hyperperiod \( H \), requiring the analysis to be carried out for a much greater number of values.

C. Deadlines

To stretch the task set to its feasibility limit, we will attempt to reduce \( \tau_i \)’s deadline by an amount \( \Delta D_i \) (so that it will be positive in the case of a feasible task set). Deadlines are subject to two separate and independent conditions: the one in Equation (12) is one, while the other requires that each deadline is greater than or equal to the WCET of its own task. The latter condition is very easy to express:

\[
\Delta D_i^{max} = D_i - C_i, \quad (13)
\]

Considering Equation 12, and modifying \( D_i \) by a quantity \( \Delta D_i \), we get the following:

\[
\forall L \in \text{dlSet}, \quad \frac{L - (D_i - \Delta D_i)}{T_i} < [S_i(L)] \Rightarrow \\
\Rightarrow \Delta D_i^{supL} = \min_{L} \{ [S_i(L)]T_i - L \} + D_i \quad (14)
\]

It should be noted that, while in Equation 4 the maximum value is included, and the computation time can be modified by \( \Delta C_i^{maxL} \) without compromising the schedulability, the same is not true for deadlines. Therefore, the deadline can be modified by a value \( \Delta D_i < \Delta D_i^{supL} \).

The problem with Equation (14) is that when deadlines are changed the dlSet set changes, too. For this reason, the most conservative solution is to generate the whole \( S_i(L) \) function up to the hyperperiod and find the minimum value. This solution is not particularly taxing from a computational point of view, and works correctly, at least with integer values (fractional values introduce extra issues which will not be covered due to space constraints).

Combining the two expressions, the final condition is \( \Delta D_i < \Delta D_i^{supL} \land \Delta D_i \leq \Delta D_i^{maxL} \). Note the difference in the equal sign, which is included in the second condition but not in the first.

A negative value for \( \Delta D_i^{supL} \) or \( \Delta D_i^{maxL} \) means that the deadline must be increased by such a value to make the system feasible.

D. Periods

In this last situation, the objective is to reduce the period of task \( \tau_i \) by an amount \( \Delta T_i \). Periods, like the other parameters, are subject to two different conditions. One is the usual condition in Equation (2), while the other one is related to processor utilization. The latter one is the easiest to evaluate (some passages will be skipped):

\[
\Delta T_i^{maxU} = T_i - \frac{C_i}{1 - \sum_{k \neq i} \frac{C_k}{T_k}} = \frac{T_i^2(1 - U)}{(1 - U) + C_i}. \quad (15)
\]

This expression (it is especially clear in the intermediate form) also shows that, if \( \sum_{k \neq i} \frac{C_k}{T_k} \geq 1 \), then the task set is already overloaded even without task \( \tau_i \), and changing its period will not be sufficient to make the system schedulable. The same methodology used for deadlines is valid for reducing the period of task \( \tau_i \) by a value \( \Delta T_i \).

\[
\forall L \in \text{dlSet}, \quad \frac{L - D_i}{T_i - \Delta T_i} < [S_i(L)] \Rightarrow \\
\Rightarrow \Delta T_i^{supL} = T_i - \max_{L} \left\{ \frac{L - D_i}{[S_i(L)]} \right\} \quad (16)
\]

Periods introduce even more difficulties than deadlines. If the period of task \( \tau_i \) changes, then not only does the dlSet change, but the hyperperiod as well. For this reason, a conservative approach would require to test all values up to \( H^* = \text{GCD}(H, T_i - \Delta T_i) \), where \( H \) is the hyperperiod.
TABLE I
PROPERTIES OF THE SAMPLE TASK SET

<table>
<thead>
<tr>
<th>Task</th>
<th>C_i</th>
<th>D_i</th>
<th>T_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ_1</td>
<td>10</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>τ_2</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>τ_3</td>
<td>2</td>
<td>100</td>
<td>6</td>
</tr>
</tbody>
</table>

Overall, the sensitivity expression for periods is the combination of the two previous expressions:

\[ \Delta T_i < \Delta T_i^{\text{supL}} \land \Delta T_i \leq \Delta T_i^{\text{maxU}}. \quad (17) \]

IV. EXAMPLE

In this section, a small example of application of the proposed methodology will be shown. The sample system is made up of three tasks, with properties shown in Table I. This task set has a hyperperiod \( H = 672 \) and a utilization \( U \approx 0.93 \).

As can easily be verified using the test summarized in (1) and (2), this task set is schedulable on an EDF scheduler. However, it has some unused processor capacity which might be exploited.

By applying sensitivity on worst-case execution times, the results are shown in Table II, with values rounded to the lower second decimal digit. The table eminently shows that neither of the two tests is sufficient by itself, and both must be executed to find the actual limit for \( \Delta C_i \). It can be immediately verified that the \( \Delta^{\text{max}} \) computed by the algorithm are the actual maximum values by which execution times can be increased without compromising the system’s schedulability.

TABLE II
PARAMETER SENSITIVITY

<table>
<thead>
<tr>
<th></th>
<th>( \tau_1 )</th>
<th>( \tau_2 )</th>
<th>( \tau_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Delta C_i^{\text{maxL}} )</td>
<td>2.19</td>
<td>0.47</td>
<td>0.41</td>
</tr>
<tr>
<td>( \Delta C_i^{\text{maxL}} )</td>
<td>1</td>
<td>0.33</td>
<td>0.76</td>
</tr>
<tr>
<td>( \Delta C_i )</td>
<td>1</td>
<td>0.33</td>
<td>0.41</td>
</tr>
<tr>
<td>( \Delta D_i^{\text{supL}} )</td>
<td>6</td>
<td>1</td>
<td>98</td>
</tr>
<tr>
<td>( \Delta D_i^{\text{supL}} )</td>
<td>3</td>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td>( \Delta T_i^{\text{supL}} )</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \Delta T_i^{\text{supL}} )</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Deadline and period sensitivity require computing the \( S \) function as described in Section III-B. For extra information, a graph displaying a close-up of the function is shown in Figure 1.

Both deadline and period sensitivity are shown in Table II. It is important to remember that, when the lower value is the one tagged with \( \text{max} \), it is included in the possible solutions, while when it is the \( \text{sup} \) one, it is not, and the next lower amount should be used.

V. CONCLUSIONS

In this short paper, an analytical methodology for sensitivity analysis has been proposed. Although quite easy to implement in an algorithm, the methodology suffers from some problems which will be addressed in the future. In fact, while the WCET sensitivity is quite complete, the problem with the other two parameters is that it is necessary to exhaustively test all possible values for \( L \) (even moreso for periods) to find the minimum possible value. The problem is even greater with rational numbers, since in this case the results are affected by the granularity of the increment used for \( L \).

The first and main thing to address in this research, as a consequence, is related to the structure of the \( S \) function. If a pattern can be identified for it, it might be possible to restrain the set of values which must be tested.

A second approach to reduce the complexity of the proposed solution would be to understand how the \( \text{dlSet} \) changes with respect to variations of the periods and the deadlines. This would allow to know up front which will be the critical values of the \( S \) function after the parameter change.

Another possible development of this work would be to introduce release jitters in the analysis, and possibly find expressions for maximizing jitter, too.

REFERENCES

On Frequency Optimization for Power Saving in WSNs

Andreea Maria Picu
INRIA ARES
69621 Villeurbanne, France
andreea.picu@insa-lyon.fr

Antoine Fraboulet
INSA de Lyon/INRIA ARES
69621 Villeurbanne, France
antoine.fraboulet@insa-lyon.fr

Eric Fleury
ENS Lyon/INRIA ARES
69634 Lyon, France
eric.fleury@inria.fr

Abstract

One of the most challenging problems in wireless sensor networks (WSNs) research is energy management. We propose two concepts aiming at saving power in low duty cycle applications. We first suggest a methodology for using hardware timers effectively. Then, we provide a way to calculate microcontroller (µC) configurations with various clock frequency setpoints, while respecting several types of constraints imposed on these frequencies, e.g., by other components of the µC, by protocol specifications, by external factors. Our evaluation shows that this approach can respect constraints while saving as much as 11.12% of energy when compared to a popular WSN operating system (OS).

1. Introduction

In recent years, embedded sensor networks have found their way into a wide variety of applications and systems with very diverse requirements and characteristics: disaster relief, environment monitoring, emergency medical response and home automation. However, in the collective conscience, the definition of sensor networks hardly changed since the early days of their military applications. This definition no longer holds for the civilian application areas mentioned above. Given the general trend towards diversification, a design space, rather than a definition, is now needed. Sensor networks should be conceived differently for groups of similar applications based on their characteristics and constraints with respect to the design space. Only then will WSNs truly be application-oriented.

Many WSN projects are currently using generic models based on popular OSes like TinyOS [8] or Contiki [2]. However, few of them have discussed the importance of specific models for sensor network programming and reconfiguration until now. Although it has non-negligible benefits, delegating this problem to generic frameworks often suffers from several drawbacks: no support for application professionals, failure to use and/or manage hardware efficiently, reductive energy management etc. Our work addresses these last two related issues.

Energy is a vital resource for mobile computing and there is unanimous consensus that advances in battery technology and low-power circuit design cannot, by themselves, meet the energy needs of future mobile systems. This is why energy management strategies must be developed for all levels: component, system, network, application etc. Schemes for power saving in WSNs often address communication protocols, but in order to account for the unique needs of each application, a global approach to the optimization of energy consumption is essential.

To provide a basis for application-specific energy administration, we discuss application-driven frequency scaling and enhanced hardware timers utilization. We present a software tool using a simple representation of the µC to configure the platform such that user and/or application timing requirements are satisfied and that power drawn from the battery is minimum.

2. Related Work and Motivation

Our work builds on the observation that generic WSN OSes use one unnecessarily high and fixed frequency, while the hardware supports several variable and much lower frequencies. Reducing the operating frequency will reduce the power dissipation linearly. However, in embedded systems, such as sensor nodes, frequency scaling is a delicate operation. Numerous features depend on and constrain clock frequency (e.g., components of the µC, protocols, applications), therefore reconfiguration will be needed if the frequency is changed. Time management, in particular, will be deeply affected by frequency scaling.

Current power saving mechanisms. There are generally multiple clocks in a µC and, except for the CPU, all other elements must choose from a set of clocks. These clocks themselves are the result of multiplexing several clock generators. Reducing the power consumption by scaling the frequency of the clocks will affect the entire platform. Some peripherals, like timers, are not easy to manage.

---

*This research was supported in part by the European Union under the 6th Framework Programme, Information Society Technologies project WASP (IST-034963) and Life Sciences, Genomics and Biotechnology for Health project MOSAR (LSHP-CT-2007-037941).
even with constant clock frequency. This is why embedded systems often use only one hardware timer to realize a list of software timers, even though many hardware timers are available. Scaling clock frequencies in a µC requires a mechanism to control all parts affected by scaling.

The frequency scaling technique reduces the processor clock frequency, allowing the processor to minimize the energy dissipation linearly. This technique saves energy even when it is not advantageous to go into Low Power Mode (LPM) at the expense of reduced performance. Although dynamic voltage scaling renders the lowest energy dissipation for most µCs, it is not always dramatically better than using a combination of dynamic frequency scaling and LPMs, which is much less expensive to implement [3]. Moreover, reducing power dissipation will have a significant positive impact on battery capacity, as shown in [9] and [12]. Frequency scaling is also essential if we plan to use voltage scaling in the future. Due to rapid advances in µC technology, we expect voltage scaling to be available for chips used in WSNs before long.

Many of today’s WSN OSes claim to be low power but they only consider LPMs for instructions. Similarly, previous studies on frequency scaling are limited to the core processor or µC and not at the circuit level or at most at the OS level. In a typical embedded system, the processor is attached to various peripherals, e.g., timers, serial ports etc. Few efforts have been made around peripheral integration for low power, even though a complete platform integration is essential in embedded systems.

**Time management in WSN OSes.** A critical part of any OS is a reliable and efficient timer service. In WSNs, application timer rates vary from a few events per week to sampling rates of 10 kHz or even higher. Ideally, hardware timers would run at the same frequency as application timers. However, WSN OSes only use one high frequency hardware timer to generate all the required application timers. Moreover, µCs provide two to four hardware timers, only one of which is used by OSes (see Figure 1).

Recent developments, like the abolition of the timer tick, largely improved time management in OSes in general. The trend extended to embedded real-time OSes with the release of TiROS [11]. Although it solves the problem of the trade-off between decent timer resolution (with increased tick frequency) and low power consumption, it still fails to fully use hardware capabilities. Full usage of hardware timers would reduce processing due to time management to a minimum.

To conclude, past efforts concerning frequency scaling and/or time management concentrate on hardware or OS. However, hardware only takes into account the past of the application and the OS handles the present. Only the application itself can really improve power consumption, since it has information about the future.

### 3. Frequency Optimization in WSNs

To optimize the interaction between hardware and software, we worked through several steps, illustrated in Figure 2 (code generation has not been addressed yet). First, we developed a novel timer allocation algorithm, since timers are one of the key µC subsystems in reducing operating frequency. We then used this algorithm to place constraints on hardware timers. These initial constraints allow us to obtain all valid hardware configurations, simply by walking the frequency optimization graph and applying the constraints associated with each vertex to the set of solutions. Both schemes are described in the following sections.

#### 3.1. Timer Management

The allocation of software timers to hardware timers is an important factor in determining the minimum frequency at which the µC can operate. As explained above, WSN OSes assign all software timers to one clock or hardware timer. The frequency is often very high (e.g., 2 MHz for TinyOS 2.x’s timer) relative to its optimal value, in order to accommodate a decent timer resolution. Our contribution is an allocation scheme that will calculate the minimum frequency required to provide all the timers for applications and the OS, while spreading these logical timers throughout the available hardware timers. In short, we switch from the approach presented in Figure 1 to the one in Figure 3.

The aim of our algorithm is to partition the set of software timers \( f_s \) into as many subsets as hardware timers \( f_h \) available. It must do so in a way that minimizes hardware timer frequencies. This is a set partitioning NP-complete optimization problem, that we solved using an
adaptation of Jensen’s algorithm [7]. Our evaluation shows that this algorithm gives far better results than less sophisticated heuristics, e.g. a greedy algorithm. We therefore obtain constraints on the hardware registers of μC timers from user and application constraints.

3.2. Frequency Optimal Configurations

Frequency scaling implies a lot of reconfiguration if we want to continue satisfying user and application requirements. This is why a hardware reconfiguration tool is essential for our project. This tool needs two inputs: a detailed description of relevant hardware and user and/or application requirements translated into constraints on hardware registers. The latter is provided by our timer management algorithm above. The former is presented in the following.

Hardware Description. Although we chose the TI MSP430 for this study, we do not make any assumptions on the μC or on the OS, hence generality is preserved. For the purpose of our analysis, we split the μC into several blocks, corresponding to subsystems sharing the same clock. Our blocks are roughly the equivalents of the μC’s peripherals as presented in [5]: Basic Clock Module, Timers A and B, ADC’s 10 and 12, Flash Controller and USART.

Our description includes the part of the hardware that is relevant to our study as a directed connected acyclic graph, in which source vertices are clock sources and sink vertices are usually frequency division registers. Since our reconfiguration tool only deals with clock frequencies, we represent only those registers that have a direct impact on hardware configurations. In the case of a divider: for all child configurations, multiply the clock frequency by the value of the divider and add that value to the configuration.

Computing Optimal Configurations. Hardware configurations consist of register values: one unique value for each register per configuration. Using our annotated graph, we calculate possible hardware configurations in the following way: we use the depth-first search algorithm to walk the graph in post-order. This allows us to start with sink nodes and work our way up to source nodes (which are all clock sources), while adding an increasing number constraints on the clock frequency on the way. When the walk is complete, we obtain a list of possible clock frequencies and the associated μC configurations. Constraints can be easily added or removed by accessing the graph structure. The traversing operation is different for each type of vertex. For example, in the case of a divider: for all child configurations, multiply the clock frequency by the value of the divider and add that value to the configuration.

Given a hardware description, and user and/or application timing requirements, the reconfiguration tool will generate frequency-optimal hardware configurations. At compile time, this allows any application to have a small set of configurations, each with its own clock frequency, and to freely switch among them. Once the possible configurations are calculated for each application, the programmer can include code in the application or in the operating system (e.g., under the form of a service), that will switch from one hardware configuration to another.

While the offline character of our optimization scheme may be seen as a drawback, it is consistent with the spirit of embedded systems and WSNs, in which applications are very simple and fully determined in advance. Most WSN OSes are designed for a single application tightly coupled with the OS and have one fixed and largely predetermined hardware configuration. Our tool allows for multiple prede-
3.3. Evaluation

Our goal is to achieve energy savings in WSNs by optimizing the frequency of the $\mu$C for each application. One important consequence of frequency optimization is that it avoids unnecessary wake-ups from LPMs. As a preliminary evaluation of our scheme, we compared the amount of energy consumed by a simple application in TinyOS 2.x and the same application using our optimization scheme.

As shown in Figure 5, we consider an application that sends a temperature sample every $\Delta$ time units. In an ideal situation, the device would not wake up from LPM between two packet transmissions: $\Delta = \delta$. However, in TinyOS, $\delta \simeq 1$ s for all values of $\Delta \geq 1$ s. Moreover, the hardware timer is configured such that the overflow of the timer’s counter will also issue an interrupt, regardless of the value of $\Delta$ and $\delta$. This counter overflows every $\simeq 2$ s (16-bit counter driven by 32 kHz crystal) and while one may think the 1 s interrupts will mask the overflows, this is not true. In reality, the 1 s alarm lacks accuracy, therefore both interrupts are received and the interrupt handlers executed with a small LPM time in between. The inaccuracy is due to the fact that periodic timers are only periodic in software: the hardware timer is reset after each interrupt. This results in an average error of about 2.35% (over several duty cycles).

We manage to improve timing in two manners: a) eliminate the maximum number of useless wake-ups within the limits of available hardware, b) use periodic hardware timers to minimize error. The maximum time a $\mu$C can go without waking up is dependent on the width of its timer’s counter and the minimum timer frequency. For the usual TI MSP430 configuration (ACLK on 32 Khz quartz crystal, 16-bit counter and clock dividers at their maximum), this time is 128 seconds. Therefore, when $\Delta > 128$ s, we will have $\Delta > \delta$ even with optimal frequency.

To calculate the energy consumed in both cases, we used component data sheets [4], [6], measures from the WSIm hardware platform simulator [1] and the performance overview presented in [10]. Our results are presented in Table 1. As expected, there is no improvement for high duty cycle applications ($\Delta \leq 1$ s), but it rapidly increases to reach $\simeq 11.12\%$ for low duty cycle applications.

### Table 1: Energy saved as compared to TinyOS

<table>
<thead>
<tr>
<th>$\Delta$</th>
<th>Useless IRQs per $\Delta$</th>
<th>Inevitable IRQs per $\Delta$</th>
<th>Energy Saved (%)</th>
<th>Avg. Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sec</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.5553</td>
</tr>
<tr>
<td>30 sec</td>
<td>44</td>
<td>0</td>
<td>5.598</td>
<td>2.5554</td>
</tr>
<tr>
<td>1 min</td>
<td>39</td>
<td>0</td>
<td>7.4550</td>
<td>2.5554</td>
</tr>
<tr>
<td>15 min</td>
<td>1.349</td>
<td>7</td>
<td>10.7706</td>
<td>N/A</td>
</tr>
<tr>
<td>30 min</td>
<td>2.699</td>
<td>14</td>
<td>10.9432</td>
<td>N/A</td>
</tr>
<tr>
<td>1 hour</td>
<td>5.399</td>
<td>28</td>
<td>11.0315</td>
<td>N/A</td>
</tr>
<tr>
<td>1 day</td>
<td>129.599</td>
<td>674</td>
<td>11.1174</td>
<td>N/A</td>
</tr>
<tr>
<td>1 week</td>
<td>907.199</td>
<td>4724</td>
<td>11.1206</td>
<td>N/A</td>
</tr>
<tr>
<td>1 month</td>
<td>3887.999</td>
<td>20249</td>
<td>11.1210</td>
<td>N/A</td>
</tr>
<tr>
<td>1 year</td>
<td>46,655,999</td>
<td>242,999</td>
<td>11.1211</td>
<td>N/A</td>
</tr>
</tbody>
</table>

4. Conclusion and Current Work

Our work introduces two complementary methods to reduce energy consumption in WSNs. The goal is to save energy while facilitating the collaboration between a very rich hardware platform and the user or the application. Ongoing work deals with further developments of the reconfiguration software: creating a code generator, including other $\mu$C peripherals in the frequency optimization graph.

A second activity targets a more thorough evaluation of our work. This includes testing it on real sensing devices and better illustrating its progress over current schemes. For example, the preliminary performance evaluation does not illustrate the improvement of our scheme over a well-parametered tickless WSN OS. In a tickless OS, a list of timers is ordered according to their expiration time and the device sets one hardware timer to the nearest deadline. Our scheme avoids unnecessary processing caused by periodic timers by using all available hardware timers.

### References

Towards Automatic Translation to Temporally Predictable Code

Robert Staudinger
University of Salzburg
Department of Computer Science
5020 Salzburg, Austria
rstaudinger@cs.uni-salzburg.at

Abstract

Contemporary Microprocessors are highly optimised towards average case performance using caches and branch prediction. While these features provide considerable speedups they come at the price of predictability. However, for real-time applications with timing precision requirements in an order of magnitude close the CPU’s clock frequency, tight prediction of WCETs (worst case execution times) is indispensable. We are proposing a conceptual model and an assembly transformation strategy to turn code with nested conditional control structures into code with a flat flow of control. This so-called single-path code facilitates the prediction of timing behaviour, ideally causing only an negligible overall slowdown. To overcome the burden of writing a full fledged compiler, we are designing our transformation to be applied post-pass, with full support for any optimisations conducted during the preceding compilation stage.

1 Introduction

Moore’s law does not go past embedded systems. CPUs of all architectures and dimensions are constantly superseded by more powerful successors. However, an unfortunate side-effect anent to the domain of time-critical applications is, that more contemporary microcontrollers tend to expose increasingly non-deterministic behaviour regarding individual instruction latencies. This can largely be attributed to the hierarchical memory model with regard to a program’s data, and pipelined execution pertaining to its code. We focus on the latter issue: when branch prediction fails, the pipeline of fetched and decoded instructions has to be flushed and refilled before program execution can proceed. Since branches can – and often will – depend on input data passed to a program at runtime, it is even theoretically impossible to correctly predict them in all and any cases. This poses a problem for hard real-time systems, where tight prediction of a program’s timing behaviour is indispensable.

The traditional approach to overcome this problem is to exactly model the hardware in question and apply path analysis to determine worst case timing scenarios [4]. However, a correct simulation of CPU intrinsics, including behaviour like instruction latencies and cache effects, is very specific to the model in question, and thus tied to considerable effort.

In the light of the complexities adherent to prediction of a program’s worst case execution time (WCET), Puschner proposed the Single-Path Approach [9] to timing-aware algorithms. The essence of this concept is to transform the code from control dependence to data dependence [1], removing conditional branches, and thus eliminating the non-determinism they are inducting. Single-path algorithms use predicated instructions to conditionally execute code instead of branching.

Research on predicated execution has extensively been conducted to increase performance on high-end processors [6]. Their high clock frequencies depend on long pipelines, which in turn increases the performance impact of pipeline stalls. An approximative rule of thumb for conditionally executed sequential blocks of code is, that predicated execution is favourable over branching, if the time required to execute the block is shorter than the time required to recover after a pipeline stall. Predicated instructions propagate through the pipeline just like their unconditional counterparts, but – the depending on CPU architecture – the execute and/or writeback stages are not executed but swapped for NOPs if the associated boolean predicate is false. Consequently any actual side effects caused by the execution of the instruction in question are impeded.

While the algorithms presented in [9] require manual adoption of source code, we are interested in automatic translation to single-path code. Rather than implementing

---

*This project has been supported by the Austrian Science Fund, project No. P18913-N15
for automatic translation of arbitrary programs to their assembly level, to allow for building upon already optimised code.

This paper is structured as follows. Section 2 introduces the predicate stack model we conceived for single-path execution of nested control flow graphs (CFGs) and illustrates the transformation using a real-world example. Section 3 presents how we are mapping the model to the ARM instruction set. Section 4 outlines preliminary experimental evaluation of this work in progress, and Section 5 gathers first conclusions and outlines future work.

2 The Predicate Stack Model

In the context of this paper, by referring to conditional blocks of code we are only identifying strictly forward conditional ones. We denote a block \( b_i \) being forward conditional if it does not have a backwards edge to the immediate predecessor block \( b_{i-1} \). Using this criterion we can sort out conditional blocks induced by loop constructs. For a more thorough discussion treating the reconstruction of CFGs from assembly code we refer to [2].

For automatic translation of arbitrary programs to their semantically equivalent single-path counterparts we introduce the notion of a predicate stack. The elements on the predicate stack mirror the nesting of conditional code blocks in the CFG. Conditional branches push onto the predicate stack, the associated join-nodes pop from it. Conditional code executes taking into account the topmost element on the predicate stack.

With regard to the model described in this section either alternatives are equivalent. If a block is already relying on predicated execution (e.g. introduced by an optimising compiler), what is left to do for the translation step is allocating the respective condition register on the predicate stack.

For the purpose of illustrating the transformation strategy and run-time execution mechanism we are using the bubble sort algorithm, also used in [11]. Figure 1 reproduces the source code exactly as used in our experiments. Furthermore Figure 4 (a) shows a terse, simplified CFG, (b) depicts the counterpart CFG after translation to single-path code. The utilization of the predicate stack can be read off at the right of sub-figure (b). Bsort is built around a single conditional block, there is no further nesting. Hence only one predicate is needed to indicate whether the code is actually executed or just passed through the CPU’s pipeline without side-effects.

The predicate in question (denoted \( p_0 \)) depends on the result of the comparison (Fig. 4, Block 3'). Thus the transformation process has to insert the predicate allocation accordingly. The operations of subordinate Block 4’ are predicated with \( p_0 \). Finally \( p_0 \) is revoked in Block 5’, before the conditional is tested again.

Obviously, in the general case of a nested conditional block \( b' \) within a block \( b \), the predicate associated to \( b' \) always depends on the the predicate of the surrounding block \( b \), as code within a disregarded branch must never be executed. Therefore each predicate that is pushed on top of a non-empty predicate stack has to be combined with the current top element at program execution time using logical \textbf{and} (c.f. Figure 3).

Figure 1. Bubble-Sort Algorithm in C

```
01 void bsort (int a[], int n) {
02     int i, j, t;
03     for (i = n - 1; i > 0; i --) {
04         for (j = 1; j <= i; j++) {
05             if (a[j - 1] > a[j]) {
06                 t = a[j];
07                 a[j] = a[j - 1];
08                 a[j - 1] = t;
09             }
10         }
11     }
```

Figure 2. Transformation Algorithm

```
01 procedure transform (block, predicate) begin
02     for each op in block do
03         rewrite_predicate (op, predicate);
04     if b := get_subordinate_block (op) then
05         p := push_predicate (op);
06         transform (b, p);
07     pop_predicate ();
08     end if
09     loop
10 end
```

Figure 3. Predicate Stack Manipulation

```
01 procedure push_predicate (op) begin
02     new_pred := get_predicate (op);
03     if stack_is_empty () then
04         stack_push (new_pred);
05     else
06         cur_pred := stack_top ();
07         stack_push (cur_pred & new_pred);
08     end if
09     end
```

Figure 2 presents the recursive algorithm used to transform a program’s CFG (constructed from assembly code) into single-path code. For the sake of brevity and clarity
the special casing for the entry block, which is not associated with a predicate by definition, is omitted. The algorithm transforms each operation in the block to use the assigned predicate (Line 3). In `rewrite_predicate()` two different cases have to be considered. (i) The instruction does not yet have an assigned predicate, in which case it is simply added. (ii) The processed instruction is already predicated as a result of optimisations done by the compiler, the predicate has to be rewritten to use the one currently on top of the predicate stack. In case the CFG forks to subordinate blocks, a new predicate – associated with the currently processed operation – is allocated on top of the predicate stack. The `transform()` procedure recurses to process the new block before the predicate is removed from the stack (Lines 4-7). This results in a depth first traversal of the CFG.

3 Mapping to the ARM Architecture

We are implementing the model proposed in the previous section on an ARM architecture\(^1\) due to the significance this CPU family has for embedded systems appliances. More specifically we are using an XScale PXA255 ARMv5 CPU on a Gumstix Connex board\(^2\).

ARM opcodes fully support predicated execution, therefore the translation of instructions is straightforward. The opcodes in question either have to be rewritten to their predicated counterparts, or in case they are already predicated by virtue of compiler optimisations (e.g. using `-O3` for gcc), the predicate has to be swapped for the respective one topping the predicate stack.

For the representation of the predicate stack at runtime we are using the condition flags provided by the program status register (PSR). They can be directly read and written using the `mrs` and `msr` opcodes. Four of the status bits (Negative, Zero, Carry, Overflow) are read- and writeable in user mode and can thus immediately be used as predicates\(^3\). This limits the maximum intraprocedural nesting depth of conditional block to a value of four, an acceptable value for code targeted at time critical systems given that loop constructs do not stress the predicate stack.

Possibilities to support even deeper nesting include extending the predicate stack to also use the status bits defined as unused by the ARM manual (a total of eight flags) and swapping out lower parts of the predicate stack to the program stack.

4 Experimental Evaluation

In order to gain experimental evidence regarding the methodologies outlined in this paper we have made an attempt to reproduce the results from [10] on the hardware platform described in Section 3. In particular we looked at the Bubble Sort algorithm, the benchmarks were compiled with gcc-3.4.5 in order to exercise them on the Gumstix only using minimal bare-metal configuration, restricted to serial I/O drivers and timing infrastructure.

By inspecting the assembly code generated when using

\(^1\)http://www.arm.com/documentation/Instruction_Set/index.html
\(^2\)http://docwiki.gumstix.org/Basix_and_connex
\(^3\)http://www.arm.com/documentation/Instruction_Set/index.html
aggressive ("O3") optimisation we observed, that that the algorithm is not suitable for single-path conversion, because gcc already heavily relies on predicated instructions instead of branches. Further investigations showed that conditional blocks up to about five statements in the C source code are almost always compiled to predicated instructions.

Hence the preliminary conclusion we draw is, that many of the well known sorting algorithms with tight loops and brief conditional blocks are unsuitable for post-pass transformation when compiled with full optimisation using gcc for ARM. We are thus looking to conduct measurements on application code, as it is not always possible to express domain-specific programs as elegantly as the discussed examples. In particular we will be looking at the controller loop of the JAviator quadrotor UAV\(^4\).

## 5 Conclusion and Future Work

In this work in progress paper we have introduced the notion of a predicate stack and presented a conceptual model for single-path execution of predicated code. Furthermore we have outlined an assembly transformation algorithm that translates arbitrary programs to single-path code. We are aware that unconditional single-path transformation is a brute force approach when applied to domain-specific programs rather than well-behaved and optimised algorithms. Nevertheless studying the behaviour of such programs with regard to single-path execution is an important direction we are setting out for further work. Also our current effort is constrained to intraprocedural transformations, further work is required to look at single-path execution from an intraprocedural point of view. Moreover, we need to collect experience regarding the behaviour of single-path code in the context of full blown embedded systems, rather than isolated benchmarks [12].

Finally we acknowledge that single-path execution is only one among a number of orthogonal issues towards improved WCET analysis and predictability. Software managed caches (often referred to as “scratchpad memory”) and fine-grained control over CPU subsystems (like for example I-Cache locking [3]) are posing interesting challenges, all the more when combined with single-path execution, as presented in this paper.

## 6 Acknowledgements

The author would like to thank Harald Röck for perpetually providing insight regarding ARM assembly intrinsics and Horst Stadler for helping with the experimental evaluations in the course of this effort.

\(^{4}\text{http://javiator.cs.uni-salzburg.at}\)

## References


Abstract
Checkpointing Rollback Recovery protocol is often used to provide fault tolerance for real-time applications. However, existing checkpointing implementations support only non-real-time applications as the checkpointing overhead is usually not deterministic. In this paper, we present an implementation of the checkpointing scheme with the Real-Time Application Interface (RTAI) supported by Linux, where services provided by the real-time operating system makes the checkpointing overhead, including the time to place a checkpoint and the time to recover the system from a failure, predictable.

1. Introduction
Checkpointing Rollback Recovery (CRR) is one of the popular temporal redundancy techniques used to achieve fault tolerance in real-time systems [1]. However, as performing a checkpointing also takes time and consumes resources, we must take into account the checkpointing overhead to better predict the satisfaction of constraints in real-time applications.

There are two main functions in a CRR protocol that need to be implemented, i.e., the checkpointing function where checkpoints are taken periodically and the recovery function where systems are recovered from faults by rolling back to previous checkpoints. Previous work in checkpointing implementation, such as [2-4], normally accomplish these two functions by utilizing multi-threaded processes on general purpose operating systems, where main function thread has to be blocked by checkpointing and recovery threads frequently. However, implementations built on non-real-time OS do not provide deterministic preemption and inter-process communication, because a kernel space thread cannot be interrupted by other kernel space threads or by user space threads. The OS kernel is “locked” once a kernel function is executing. This usage of locks introduces non-deterministic latencies for both checkpointing and recovery tasks, which are not tolerable in real-time applications.

2. Library Implementation with RTAI

2.1. RTAI
The Real-Time Application Interface (RTAI) modifies the general purpose Linux kernel so that the patched operating system can use the Interrupt Abstraction approach to add deterministic real-time characteristic. Specifically, with an additional Interrupt Abstraction layer on top of general purpose Linux, RTAI can intercept hardware interrupts before they go to the Linux kernel. RTAI then apply real-time scheduling policies to decide which task shall be run first. Comparing with general purpose Linux, RTAI’s task scheduler uses fully preemptive scheduling based on a fixed-priority scheme and hence provides predictable behavior for hard real-time tasks.

Another nice feature of RTAI is that it provides a technique named LXRT which allows users to develop and run hard real-time tasks in user space using the same API that is provided in kernel space RTAI. This practice makes the development, debug and test of real-time applications much easier than in the kernel mode. This is the method that we use in this paper to implement the checkpointing scheme on RTAI.

2.2. Checkpointing Scheme in RTAI
For each real-time application running on the RTAI Linux, which is called “main function” in this paper, there are two associated tasks, i.e., the checkpointing
task and the recovery task. Fig. 1 and 2 give the work flows of these two tasks, respectively.

As depicted in the Fig. 1 and 2, these two tasks are performed through cooperation of four main modules: a checkpointing module, a fault detection module, a fault recovery module, and a main function module.

All the modules are implemented as real-time tasks supported by RTAI preemption and real-time scheduling services. Since the scheduling is based on priorities, the assignments of priorities on different modules need to be carefully considered. In our library priorities are set as below with higher numbers representing higher priorities.

<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Function</td>
<td>Priority 1</td>
</tr>
<tr>
<td>Checkpointing</td>
<td>Priority 2</td>
</tr>
<tr>
<td>Fault Recovery</td>
<td>Priority 2</td>
</tr>
<tr>
<td>Fault Detection</td>
<td>Priority 3</td>
</tr>
</tbody>
</table>

Table 1. Real-Time Tasks and Their Priorities

2.3. Implementation

Our checkpoint library is developed in user space with LXRT. Under LXRT, these tasks can be conveniently coded and tested in user space, and at the same time benefit from the real-time characteristic. The implementation is based on the deterministic preemption ability offered by the RTAI. With the RTAI scheduler, real-time tasks with higher priority will be able to preempt lower-priority tasks, and hence have deterministic timing behaviors.

The first development step is to use the API’s provided by LXRT to create each function module as a real-time task associating with a priority specified in Table 1. Specifically, we use two RTAI functions: \texttt{rt_task_init_schmod} and \texttt{rt_make_hard_real_time} to create a real-time task. There are two things happening after these two functions are called. At first, a task is created and is assigned a priority. In LXRT, however, SCHED_OTHER is the standard Linux default scheduler performs non-preemptable and non-priority scheduling on tasks. So the second function is to switch the scheduling to SCHED_FIFO, which is intended for special and time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with SCHED_FIFO are assigned static priorities in the range from 1 to 99, which means that when a SCHED_FIFO process becomes runnable, it will immediately preempt a running SCHED_OTHER process or a SCHED_FIFO process of lower priority [5]. A FIFO (first in, first out) policy is applied to processes of the same priority. Preempted SCHED_FIFO processes remain at the head of their priority queue and resume execution again once all higher-priority processes become blocked, which obviously can help us to predetermined our running order and realize real-time performance.

As described in Section 2.2, we have four tasks running concurrently in a system. The main function is then created as a real-time task with priority 1, which means it is the lowest priority and can be preempted by other higher priority tasks. In order to perform the checkpointing functionality depicted in Fig. 1, we create a checkpointing task with priority 2. Meanwhile, since a checkpoint will be taken periodically, we need to set a real-time timer and make the checkpointing task as a periodical real-time task by calling the function \texttt{start_rt_timer} to start a real-time timer, and then \texttt{rt_task_make_periodic} to make the timer a periodical one. Then when the time reaches the period, the timer wakes up the checkpointing task. There are two possible situations when the checkpointing task is up: (1) when the current running task is the main function. Since the checkpointing task has higher priority, it preempts the running main function and start taking checkpoint. After the checkpoint is taken, another function \texttt{rt_task_wait_period} will be called such that the checkpointing task will be sent back to sleep and wait for the next coming period. The real-time scheduler will then resume the execution of the main function; (2) if the current running task is the fault detection or fault recovery. Since the checkpointing task has lower priority, the scheduler will simply block the task until the higher priority tasks finish.

To achieve fault recovery, we need to create two real-time tasks, i.e., the recovery task with priority 2 and fault detection task with priority 3. The fault detection
task is also periodic. When the timer reaches the fault detection interval, the fault detection task preempts all running tasks and sends a “keep alive” signal to the main function. If no response is received, it will report a fault occurrence by sending an RPC signal to the fault recovery task and then block itself.

Different from the checkpointing and the fault detection tasks, the fault recovery task is event-driven instead of time-driven. Specifically, it starts as an infinity loop and waits for a fault event. When the recovery task receives “fault occurrence” signal from the fault detection task, it calls the function `rt_task_resume` so that the real-time scheduler put it in the front of the running queue for execution. The task will read the previous checkpoint from the persistent storage, and recover the application state accordingly. After the recovery procedure finishes, the recovery task then calls `rt_task_suspend` function to suspend itself again in the infinite loop, until the next fault occurrence event arrives.

It is worth noting that the checkpointing frequency has impacts on system performance. In particular, more frequent checkpointing speeds up the recovery when failures occur, and therefore improves the system availability and accelerates the execution time. However, checkpointing also takes time and consumes resources. It increases the fault-free execution time and can jeopardize the satisfaction of timing constraints. The checkpointing task hence may need to communicate with non-real-time Linux processes to receive adaptive checkpoint interval information. For instance, a central controller located in a remote process may decide the proper checkpoint interval and send the value to the checkpointing task through communication network. The challenge for adaptive checkpoint interval in real-time application is that we need to guarantee that new checkpoint interval can be applied to the application and be effective within predictable time.

RTAI provides a set of real-time Inter Process Communication (IPC) mechanisms that can be used to transfer and share data between tasks in both the real-time and Linux user space domains. These mechanisms include real-time FIFO’s, mailboxes, semaphores, and RPC’s (Remote Procedure Calls). In this implementation, we use the real-time FIFO for checkpointing task to receive massage from normal Linux tasks.

Specifically, when the checkpointing task is resumed by the periodical timer and before it takes a checkpoint, it checks the real-time FIFO queue to see if there is a message indicating the change of checkpoint intervals. If a new checkpoint interval is detected, the checkpointing task finishes saving its current checkpoint first and then calls function `next_period`. This function resets the time which will be the caller periodical task’s next running period. Since the checkpointing task can be guaranteed to obtain the CPU periodically, the adaptive checkpoint intervals are hence able to be applied within a deterministic time range. In fact, if a checkpoint reset message is in the FIFO queue, and the previous checkpoint interval is \( Y \), the new value will be effective in no later than \( 2Y \) time.

Fig. 3 gives an overall architecture of our implementation on RTAI.

3. Experiment Results

The experiment settings are as follows: a Pentium Dual Core 1.6GHz CPU and 1GB RAM. The system is running on a Federal Core Linux with kernel version 2.6.18 and an RTAI 3.4 patch. In our experiments, we develop a simple application that adds 1 to the current values starting from 1 until we force it to terminate. The checkpointing operation is hence to save the current accumulation value into a file, and the recovery operation is to retrieve the checkpoint (previous accumulation value) and continue adding values to that.

The first experiment is to show that the time to take a checkpoint is predictable in our implementation. To test it in a stress environment, we create “disturbing” threads in the background. Specifically, when the checkpointing task starts executing, we run various number of normal Linux dummy threads (priority 0) and lower priority real-time dummy threads (priority = 1) in the following order: first, we test the checkpointing overhead with no disturbing thread. We then test by separately increasing the number of normal Linux thread by 10 and real-time thread by 1. Next, we simultaneously increase the number of normal Linux thread by 10 and real-time thread by 1. Lastly, we increase the number of normal Linux thread by a larger amount 30. We repeat the experiment and adopt the average values.
From the results in Table 2, we can see that in spite of the disturbing threads running in the background, the time to take a checkpoint remains almost the same with a changing range per disturbing normal Linux thread increase less than 0.2% and per real-time thread increase less than 2.5%, and hence is in a predictable range. This is due to the deterministic preemption and priority-based scheduling provided by the RTAI.

<table>
<thead>
<tr>
<th>Checkpointing Time</th>
<th>Number of Normal Linux Thread</th>
<th>Number of Real-time Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 ms</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40 ms</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>41 ms</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>42 ms</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>44 ms</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>46 ms</td>
<td>60</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2. Checkpointing Overhead

The second experiment is to measure the overhead of recovering a fault. In this experiment, we create another task named fault generator. This task periodically (every 20 ms) produces an artificial fault to be fed to the fault detection task and trigger recovery task.

<table>
<thead>
<tr>
<th>Recovery Time</th>
<th>Normal Linux Thread Number</th>
<th>Real-time Thread Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>33 ms</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>33 ms</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>34 ms</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>34 ms</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>35 ms</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>37 ms</td>
<td>60</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3. Recovery Overhead

From the results in Table 3 we can see that the recovery time does not change much (less than 0.2% for per normal Linux thread and 1.5% for per real-time thread) even after adding 60 normal disturbing threads in the background.

Our last experiment is to show the time it takes to apply a new checkpoint interval on a running application. We create an adaptive checkpointing task that put a message into the real-time FIFO queue, and measure the time between the message entering to the queue and the new checkpoint interval being effective. We change the checkpoint interval in 5 different ways, repeat the test on each of the adaptation, and record the average value.

<table>
<thead>
<tr>
<th>Previous Checkpoint Interval</th>
<th>Next Interval</th>
<th>Checkpoint Switching Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>48</td>
<td>44 ms</td>
</tr>
<tr>
<td>48</td>
<td>60</td>
<td>42 ms</td>
</tr>
<tr>
<td>60</td>
<td>80</td>
<td>44 ms</td>
</tr>
<tr>
<td>80</td>
<td>100</td>
<td>41 ms</td>
</tr>
<tr>
<td>100</td>
<td>120</td>
<td>42 ms</td>
</tr>
</tbody>
</table>

Table 4. Checkpoint Interval Switching Overhead

The results in Table 4 indicate that the adaptive checkpoint intervals can be applied dynamically and be effective within a deterministic time frame.

4. Conclusions and Future Work

In this paper we implement the Checkpoint Rollback Recovery scheme in RTAI real-time operating system. The preemptable interrupt service provided by the RTAI makes the checkpointing overhead predictable, so that the checkpointing scheme is feasible to be applied in real-time applications to provide fault tolerance. The experiment results performed on a real system indicate that the checkpointing overhead are close to constants.

Our future work is to extend this work to distributed environment, where global system states are maintained through synchronized checkpointing protocols. The deterministic synchronization overhead hence needs to be guaranteed by utilizing real-time-aware inter-process techniques.

References
A 2000 frames / s programmable binary image processor chip for real time machine vision applications

A. Loos, D. Fey
Institute of Computer Science, Friedrich-Schiller-University Jena
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
\{loos,fey\}@cs.uni-jena.de

Abstract

Industrial manufacturing today requires both an efficient production process and an appropriate quality standard of each produced unit. The number of industrial vision applications, where real time vision systems are utilized, is continuously rising due to the increasing automation. Assembly lines, where component parts are manipulated by robot grippers, require a fast and fault tolerant visual detection of objects. Standard computation hardware like PC-based platforms with frame grabber boards are often not appropriate for such hard real time vision tasks in embedded systems. This is because they meet their limits at frame rates of a few hundreds images per second and show comparatively long latency times of a few milliseconds. This is the result of the largely serial working and time consuming processing chain of these systems. In contrast to that we designed an application-specific instruction processor chip which exploits massive parallelization of often used image preprocessing algorithms to minimize computation times. To get a feasible image resolution of 320 x 240 pixels at processing frame rates up to 2000 frames per second we realized an image processor on a semi-custom 0.18 \( \mu \text{m} \) pure logic CMOS platform. The paper presents the architecture, the performance parameters of the designed processor chip and some simulation test results.

1 Motivation and introduction

The motivation to present that paper emerges from a firm tendency to substitute PC-based standard machine vision systems with smaller and faster embedded components (e. g. smart cameras), what is currently a world-wide on-going ambitious research topic [1],[2],[3]. One way to manage that is to use application specific integrated circuits (ASICs) as basic platform. The advantages of ASIC based components confront with their main weakness: the inflexible and fixed instruction set. To meet that we present a so called ASIP (application specific instruction set processor) which combines the flexibility of a GPP (General Purpose Processor) with the speed of an ASIC.

Figure 1. Image processing flow

Before we explain some details of our processor architecture we point out the characteristic data processing flow of the considered machine vision system in which our processor chip will work. Figure 1 illustrates a generic procedure of the embedded machine vision environment. In the beginning a real scene is captured by a CMOS imager and converted to digital values (1). Afterwards the gray scaled image is segmented (2) and we receive a raw binary image representation. The quality of the image is improved by applying e. g. morphological filter operations (3). In the next step we will calculate diagonal, vertical and horizontal projections of the binary image (4). This means simply that pixels are counted what can be perfectly parallelized. This information can then be used in the next step to calculate the object’s centroid and orientation, what is one of the most
important tasks in industrial image processing. For step 1 we can use a commercial CMOS sensor as well as a sensor which was especially designed in our project. The steps 2 and 5 are solved by hardwired algorithms implemented in a low-priced medium class FPGA (field programmable gate array). The steps 3 and 4 are calculated in our ASIP chip which is the core of our embedded real time vision system and which we present in this paper.

The rest of the paper is organized as follows. Chapter 2 presents the ASIP chip architecture and the primary system parameters, chapter 3 shows some application examples. The fourth chapter presents the currently working process. Finally, some conclusions are given.

2 Chip architecture

2.1 Overview and general chip features

The designed architecture of our ASIP is a result of a precise analysis of the needed performance and the algorithms to solve image processing problems as described before. Our design strategy can be summarized as follows: don’t integrate as much as possible functionality but rather as much as even required. Figure 2 shows a block diagram of our ASIP chip architecture, its data paths (wide arrows) and its control paths (small black arrows). The main components are the processor array (core), the control unit, which contains a microprogram unit and the vertical and horizontal counter arrays. Using the capabilities of the micro program unit self-defined arbitrary morphological 3x3 operators can be loaded into the control unit. Therefore the possibilities to manipulate binary images are almost unlimited. The task of the vertical and horizontal counters located in each pixel row and pixel column is to realize the pixel counting for the projection operations as described above. Two standardized and user friendly interfaces (JTAG, SPI) serve for the connection to the chip’s outside world. These interfaces allow to load the microprogram or to chose simple, but otherwise time consuming algorithms (in the serial computing case) to remove disturbances in images, e. g. holes in objects or speckles on the background or to detect edges.

The input data can be read serially (slow mode) or via a 16 bit wide data bus (fast mode). No external memory modules are required, the chip has internal registers to store the entire image and all required temporary data. An exceeding feature is the fact, that the consumed processing time is not depending on the image size since only local operators are applied.

The chip is driven by a 40 MHz clock. As result, a single morphological operation performed on a 320 x 240 pixel image including data in/output only needs 250 $\mu$s. For even faster data in/output a ROI (region of interest) can be defined in steps of 20 pixels in horizontal and 32 pixels in vertical direction.

To solve general object criteria like centroids and object orientations a fast executable preprocessing method is realized on chip. This is performed, as mentioned above, by two parallel working counter arrays determine the three possible projections in horizontal, vertical, and diagonal direction. The output data can be a preprocessed image or the pixel projection values. The control of both the data in/output streams and the data processing within the core and the counter arrays is organized by a finite state machine which is part of the control unit.

A boundary scan chain located on the left chip side and closely assigned to the JTAG control module provides board level test purposes.

2.2 Processor core

During the design analysis it became obvious, that it is not recommended to create a full parallel circuit, where each image pixel has its own processing unit. This would result in too much chip area. Therefore we decided to design a mixture of a time and space multiplexing architecture, i.e. a group of pixels is serially processed by one PE and several of such PEs are working simultaneously. This requires to find a trade off between a fixed number of serially processed pixels by one PE and the resulting image operator latencies.

In order to compute the 76800 image pixels (320 x 240) on a strongly limited chip area of 25 mm$^2$ within a reasonable
processing time, we fixed that parameter to 16 pixels per processign element (PE) ([4]). As result the processor core consists of 4800 PEs. The architecture of one is shown in Figure 3.

One PE has three synchronous clocked 1 bit registers and a small combinatorial network (LU, logical unit) to compute its own new state depending on the own pixel value and the value of the eight neighbored pixels in each calculation step. That means that all PEs in the core are linked with each other by a local X-network. The LU allows to carry out the basic boolean functions and, or, and not. Their controllable interconnections allow to perform the mentioned binary morphological operations. The result storage and feedback supports to process the image data in multiple cycles. Therefore the number of a certain set of morphological operations can be subsequently combined depending on the image processing problem.

Due to the time multiplexing approach further hardware resources for shifting data and for storing pixel data and temporary results are closely attached to each PE. That basic structure is shown in Figure 4 (modules 1 to M represent the additional shift and store resources). The shift resource is necessary to shift step-by-step pixel data to the neighbored modules and to the horizontal and vertical pixel counters. Fifteen of those multiplexed PEs form a closed ring to process one image row (maximum: 240 pixels). The alignment of the 320 row structures one upon the other builds up a vertical array and ensures the massive parallel vertical interconnection to the neighbored units.

Figure 3. Schematic structure of one processing element

3 Application example

In table 1 the result of a functional simulation of our ASIP prototype for a programmed segmentation operation is shown. An image of a vehicle rim (a representation of a typical industrial scene) is captured by an image sensor and segmented with a fixed threshold. In addition the image is cut by the in-built ROI functionality to the required image section (1 and 2a). The removal of some disturbances can be performed by the designed circuit (temporary images 2b - 2d). The overlay of the preprocessed image, which is the ASIP output, and the original gray scale image is quite good (3).

Table 1. Application example of a rim inspection (Image 1 by courtesy of V&C GmbH)

One complete procedure including image input, image processing and image output needs only 250 µs what is a pretty good value to fulfill industrial inspection tasks in real-time. Furthermore we carried out simulations in which the
centroid and the orientation of objects is calculated in combination with the FPGA using the projections which are determined by the ASIP-internal horizontal and vertical counters. When we assume a time of further 250 µs (a spare of 10000 clock cycles per image at 40 MHz) to calculate the physical moments we are able to process two thousand images in each second.

4 Working progress

4.1 Finished work

The basic design works of the processor core architecture date back to 2006. The VHDL models were tested and validated in the first half of 2007. The RTL and layout synthesis of the complete ASIC design (see Figure 5) including all modules was performed in fall 2007. There were some difficulties to fit in the design into the chip area (one tile of 5 mm x 5 mm dimension). Due to the lack of vertical space the core supply pads could not be placed in a regular orientation. After finishing the validation of the physical design constraints (distances of wires, geometries of internal structures) the tape out was carried out in November 2007. We received the manufactured chips in February 2008.

![Figure 5. Image processor chip (GDSII view)](image)

4.2 Pending work steps

Unfortunately we could not use a standard package offered by the fab to enclosure the circuit. The reason is that too many bond pads are located on the left circuit edge. Therefore the processor chips has to be bonded and packaged by a project’s co-partner specialized in electronic packaging. We prefer a COB (chip on board) technology, in which the chip is directly fixed and bonded onto the base circuit board.

The next steps would be testing activities and a prototype realization. The integration of the chip into a smart camera system for industrial purposes is intended in fall 2008.

5 Conclusions

In machine vision object detection and classification is a common application, e. g. to inspect automated product pipelines. The algorithms to segment images, to identify certain objects out from a set of objects known in advance, and to detect their position and their orientation within few milliseconds with cheap hardware is both an economically important and technically challenging task. To meet that we designed a massively parallel programmable ASIP processor chip which is suited for the integration in small embedded vision systems fulfilling real-time tasks. We solved this by a microprogrammable parallel on-chip architecture which allows e. g. the programming of fast image segmentation operations. This programmable structure is supported by additional counter resources to extract certain features like the moments of zeroth, first and second order to compute rapidly, area or centroid resp. orientation of detected objects with a throughput of up to 2000 images per second. The reason to choose exactly the chip area of 25 mm² is that only fixed tiles of 5 mm x 5 mm area units were supported by the mask house at the used chip technology node. The pixel resolution of later manufactured chips may be larger without that constraint. The generic formal description of the architecture would support this.

Acknowledgments

This work is supported by the local government of Thuringia, Germany, Ministry of economics, technology and work (TMWTA).

References


Providing QoS by Scheduling Interrupt Threads

Gabriele Modena, Luca Abeni, Luigi Palopoli
University of Trento
Trento - Italy
gabriele.modena@gmail.com, luca.abeni@unitn.it, palopoli@dit.unitn.it

Abstract

This WiP describes some preliminary results obtained when experimenting with the priorities of IRQ threads in a real-time version of the Linux kernel. IRQ threads allow to schedule interrupt handlers so that their interference on real-time activities can be controlled. However, the experiments presented in this paper indicate that fixed priority scheduling does not provide enough flexibility for finding a trade-off between real-time performance and throughput, and we argue that reservation-based scheduling is needed.

1 Introduction

Real-time scheduling theory has traditionally dealt with the problem of scheduling the CPU so that the execution of a set of concurrent tasks can meet some timing constraints. The kind of real-time constraints considered range from hard real-time constraints (requiring strict and deterministic execution guarantees) to soft real-time constraints (for which occasional violations can be tolerated and probabilistic performance guarantees are required). Moreover, different kinds of tasks (ranging from tasks characterised by fixed execution and inter-activation times to tasks described by stochastic descriptions) have been analysed, and scheduling algorithms have been modified to address high variabilities in the task sets.

However, the CPU is not the only type of resource that needs to be shared between the various applications running in a system. Very often, real-time tasks need to interact with IO devices (e.g., acquiring data from sensors, or sending computation results to actuators) or with other nodes through a network link. The need for a real-time I/O creates problems of challenging complexity, which cannot be mitigated by simply using suitable real-time scheduling algorithms for the CPU.

Some recent pieces of work have shed some light on a largely underestimated problem: a real operating system kernel needs some CPU time to exchange data with hardware devices [3, 6]. For instance, it is completely useless to precisely schedule a device (e.g., a disk) if the kernel is not able to find enough CPU time to manage the incoming data. And, the CPU time spent by the kernel for handling the device must not be accounted to real-time tasks that do not use such a device (causing deadline misses) This means that a really coordinated strategy for the scheduling of different resources [9] is needed.

In this work in progress, we investigate how some recent patches for the Linux kernel permit to make IO activities schedulable, and we experiment with different priority assignments verifying that fixed priority scheduling does not provide enough flexibility for controlling both the real-time performance and the throughput of real-time and non real-time applications coexisting in the same system.

2 Kernel Structure

As explained in the introduction, to schedule other resources than the CPU the OS kernel needs to consume CPU time in handling hardware interrupts coming from the various devices providing the resources. To understand why the time spent serving interrupts can be a problem for real-time applications, consider the structure of a traditional kernel, in which hardware interrupts are generally served in two phases:

- a short Interrupt Service Routine (ISR) is invoked as soon as an interrupt fires and is responsible for acknowledging the hardware interrupt mechanism, postponing the real data transfer and processing to a longer routine, to be executed later;
- a longer routine (soft interrupt, or bottom half) is executed later to correctly manage the data coming from the hardware device.

ISRs generally execute with interrupts disabled, while soft interrupts always execute with interrupts enabled and are served when switching from kernel space (where ISRs run)
to user space (where user programs are executed). Therefore, soft interrupts can be preempted by ISRs.

Both ISRs and soft interrupts have a higher priority than user tasks, and can “steal” execution time from them. Such “stolen time” can be accounted in real-time guarantees by modelling it as a blocking time, and/or by modelling ISRs and soft interrupts as high priority tasks. This implies that a low-priority task can make a task set unschedulable by causing the generation of a large number of hardware interrupts.

This problem is generally solved in real-time kernels by scheduling the interrupt handlers: for example, the Real-Time Preemption patch (RT-preempt) [8] introduces real-time features in the Linux kernel and transforms ISRs and soft interrupts in kernel threads (the hard IRQ thread and the soft IRQ thread), that are schedulable entities handled by the kernel scheduler in the same way as user tasks (so, IRQ threads can have lower priorities than real-time tasks, and can be preempted by them). A real-time application that does not need to interact with a specific device can schedule its tasks in foreground respect to the device’s interrupt handlers, so that real-time tasks are not disturbed by the device’s interrupts.

This solution can present a slightly higher overhead, and requires a more careful synchronisation, but also has the advantage of permitting to correctly account the handler code in a real-time system (that is, the CPU time required to execute the handler can be correctly accounted in order not to break the system’s guarantees).

The possibility to schedule interrupt handlers (provided by IRQ threads) permits to give user-space real-time tasks higher priorities than interrupts, reducing the interference from hardware devices. However, it is still not clear how to assign priorities so that real-time and QoS guarantees are respected: although real-time theory provides tools for assigning priorities to real-time tasks (for example, by using the Rate-Monotonic - RM - priority assignment), there still are no reliable algorithms to properly assign priorities to IRQ threads.

Of course, it is easy to find priority assignments that provide good real-time performance in specific cases: for example, when real-time applications do not need to access a hardware device, the IRQ threads provided by RT-Preempt allow to reduce the interference caused by such a device. However, it is not easy to assign the tasks priorities when the real-time application depends on data coming from the device.

3 Scheduling the IRQ threads

Since there is not any theoretical model showing how IRQ threads affect devices throughputs and the performance of real-time tasks, we have assessed the effects of IRQ threads priorities through a set of experiments.

To evaluate the interactions between a set of periodic real-time tasks and a hardware device generating interrupts:

- a network card has been selected as an interrupt generating device because it is easy to generate a controlled load on it, and to measure the network throughput;
- a set of periodic periodic real-time tasks has been used to generate some time sensitive CPU load, and all the real-time tasks have been scheduled using real-time (SCHED_FIFO) priorities assigned according to RM;
- real-time performance have been quantified by measuring the latency [1] experienced by a periodic task. This latency is a good real-time performance metric, because it must be accounted in the admission test as a blocking time $B_i$, so high latency values risk to make unschedulable task sets that would be schedulable if kernel effects were not considered.

The impact of IRQ threads’ priorities has been measured by repeating the experiments with different priority assignments. In particular, the goal of these experiments was to check how manipulating the priorities of the interrupt threads allows us to control the real-time tasks’ latency and the network throughput.

To reduce the impact of external factors, the experimental setup is composed of two computers connected by a cross network cable. The cyclicstest program [5] has been used to measure the latency experienced by a real-time task with period 10ms, and the netperf program [4] has been used to generate a very high network load and to measure the throughput achieved by the network card. One of the two computer generates the network traffic by using a netperf client, while the other computer runs the netperf server together with the set of real-time tasks and cyclicstest. This second computer is an AMD K6-2@400Mhz running the 2.6.24-rc2-rt1 Linux kernel[7], and both the computers use a 100Mb Realtek ethernet card.

The priorities of the cyclicstest periodic task and of all other real-time tasks have been assigned according to RM, and the priorities of the IRQ threads serving the network card (the hard IRQ thread, and the softirq-net-rx thread - these two threads will be indicated as “networking threads”) have been varied from 1 (minimum priority) to 99 (maximum priority). To better

---

1. The schedulability of a real-time task set can be guaranteed by using an admission test, which is traditionally based on the execution times and periods of real-time tasks (utilization-based test, response time analysis, or time demand analysis). This admission test can be enhanced to account the blocking times, and/or by introducing in the admission test some high priority tasks modelling interrupt activities.

2. Note that we used a low-power computer by purpose, to better highlight the problems caused by the interrupt handlers.
<table>
<thead>
<tr>
<th>Priority</th>
<th>Maximum Latency</th>
<th>Net Throughput</th>
<th>95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → 49</td>
<td>98µs</td>
<td>37Mbps</td>
<td>3Mbps</td>
</tr>
<tr>
<td>50 → 79</td>
<td>94µs</td>
<td>38.3Mbps</td>
<td>1.6Mbps</td>
</tr>
<tr>
<td>80</td>
<td>148µs</td>
<td>76.6Mbps</td>
<td>1.2Mbps</td>
</tr>
<tr>
<td>81 → 99</td>
<td>164µs</td>
<td>72.25Mbps</td>
<td>2.1Mbps</td>
</tr>
</tbody>
</table>

Table 1. Real-Time latency and network throughput experienced assigning different priorities to the interrupt threads.

![Figure 1. Latency CDF for a non real-time Linux kernel running netperf.](image1)

![Figure 2. Latency CDF for a RT-Preempt Linux kernel running netperf.](image2)

To expose the effects of these two threads, netperf has been configured to use UDP packets composed by 600 bytes. To have a baseline value to be used as a reference for the following results, a vanilla Linux kernel (without IRQ threads) has been used in a first set of experiments, which resulted in the latency Cumulative Distribution Function (CDF) depicted in Figure 1. Although the probability of measuring a latency < 200 is high, the distribution function has a long tail, and the maximum measured latency is 70602µs (note the logarithmic scale on the X axis in the figure). The corresponding network throughput is about 80Mbps.

Figure 1: Latency CDF for a non real-time Linux kernel running netperf.

Figure 2: Latency CDF for a RT-Preempt Linux kernel running netperf.

While the achieved network throughput is reasonable, a worst-case latency of more than 70ms is not acceptable for a large number of real-time applications. Running the same experiments on a RT-Preempt kernel (without any tuning of the IRQ priorities) resulted in lower worst-case latencies, as shown in Figure 2 which compares the latency CDFs for an “-rt” and a vanilla kernel. Note that the CDF for the 2.6.24-rc2-rt1 ends before 100µs (so, the worst-case latency is less than 100µs), while the CDF for the vanilla kernel is truncated (as shown in Figure 1, it reaches 1 after 70ms). However, the network throughput for Preempt-RT went down to less than 50Mbps. To avoid this decrease in networking performance (while not renouncing to low latencies), we investigated the effects of tasks priorities the latency and throughput, through a new set of experiments.

Since some first experiments seemed to confirm that assigning the same priority to the hard IRQ thread and to the soft IRQ thread gives the best results, we decided to always assign priorities in this way. The experiments’ results showed four different possibilities for priority assignment:

1. the networking threads have the lowest priorities in the system. This includes all the priorities from 1 to 49 (50 is the default priority of all the IRQ threads);
2. the networking threads have priorities between 50 (the priority of all the other IRQ threads) and the priority of the periodic real-time threads (in particular, cyclicstest, whose priority is 80);
3. the networking threads have the same priority as cyclicstest;
4. the networking threads have the highest priority in the system. This include all the priorities ranging from 81 to 99.
Table 1 summarises the results obtained in the most relevant cases. In particular, it is possible to see that when the networking threads have priority from 1 to 49 (case 1), the latencies experienced by cyclictest are smaller than 100µs but the achieved network throughput is low.

The latencies and throughput measured in case 2 (network threads priorities between 50 and the priorities of the real-time threads) are basically equivalent to the ones measured in case 1.

When the networking threads are scheduled at priority 80, which is the same priority as cyclictest (case 3), the throughput measured by netperf increases to 76Mbps\(^3\), but the latency experienced by real-time tasks is increased by about 50µs.

Further increasing the networking threads priority (case 4) increases the latency, but has no positive effects on the network throughput.

Unfortunately, the increase in latency is not gradual, so it is not possible to assign tasks’ priorities to obtain a latency between 100µs and 140µs; in the same way, it is not possible to have a fine-grained control on the network throughput by only playing with priorities. Note that the throughput obtained by assigning to the IRQ threads priorities smaller than the real-time tasks priorities is very bad, and these priority configurations can be hardly considered useful.

The previous experiments show that fixed priority scheduling does not provide enough flexibility to control both real-time performance and hardware device throughput in an effective way. Hence, we argue that more advanced schedulers should be used for the IRQ threads; since the load of such tasks is highly variable and unpredictable (being generated by hardware interrupts which often do not follow any controlled arrival pattern), we believe that a scheduler allowing us to reserve a fraction of CPU time to IRQ threads would be more appropriate for scheduling them.

A first candidate for scheduling IRQ threads is the Completely Fair Scheduler (CFS) that has been recently introduced in the Linux kernel (and implements a form of Proportional Share scheduling). Unfortunately, some preliminary experiments seem to indicate that CFS is not yet able to provide latencies below 200µs (it is not clear if this is due to the CFS algorithm itself, or to implementation issues).

4 Future Work

This Work-in-Progress only reports preliminary results (which look very interesting, because they show that IRQ threads require scheduling algorithms more advanced than the traditional fixed priority one).

We are currently working on some experiments to check if CFS can be used to enforce temporal protection between IRQ threads and real-time applications running in user space\(^4\). We also plan to run some experiments using a Sporadic Server (which is included in the POSIX standard) to implement this form of temporal protection.

The temporal protection between tasks can also be obtained by using a reservation-based scheduler such as a CBS-based one \([2]\). Our prototype of CBS scheduler for Linux is compatible with the RT patch, and we are starting to use it for scheduling IRQ threads. We expect that the flexibility and guarantees provided by this scheduler will allow us to find a good trade-off between latency and throughput, but we have no numbers to show yet.

Finally, we plan to confirm the obtained results by using different interrupt-generating devices (for example, the hard disk controller) and different workloads.

After collecting a large amount of data through the previously described experiments, we aim developing a mathematical model allowing us to provide real-time and QoS guarantees by scheduling IRQ threads and to use the experiments results for validating the model.

References


---

\(^3\) Note that when using 600 bytes-long UDP packets this value is near to the maximum achievable throughput.

\(^4\) Some preliminary results seem to indicate that CFS can easily provide temporal protection between tasks, but it cannot provide low latencies. We are still investigating the reason for this results.
On the Benefits of Relaxing the Periodicity Assumption for Control Tasks

Adolfo Anta and Paulo Tabuada
Dept. of Electrical Engineering
University of California, Los Angeles
E-mail: {adolfo, tabuada}@ee.ucla.edu

Abstract—Feedback control laws have been traditionally treated as periodic tasks when implemented on digital platforms. Although this approach facilitates the scheduling of control tasks, it also leads to inefficient implementations. In this paper we seek to demystify the periodicity assumption in favour of aperiodic self-triggered implementations of control tasks. We show that by adopting aperiodic models for control tasks we can considerably reduce processor utilization while ensuring stability and desired levels of control performance. Based on previous work by the authors, a modification of Cervin and Eker’s control server is proposed to fully exploit the benefits afforded by aperiodic self-triggered control tasks. We illustrate the proposed techniques on the control of two jet engine compressors.

I. INTRODUCTION

Historically, control applications have been developed by adopting a separation of concerns between control engineering and real-time scheduling: control engineers design feedback control laws under the assumption that implementation effects are negligible (zero delays and zero computation times) while software engineers schedule control tasks by minimizing jitter and input-output latency in the control loop. This approach leads to overly conservative designs since the same period is used for the control task independently of the processor load and the behavior of the system being controlled. Moreover, the period is designed in order to provide performance guarantees under worst case conditions even if these only rarely occur.

Recently many authors have proposed an integrated study of control design and real-time scheduling. Seto et al [SLSS96] approach the problem as an optimization problem, by defining a performance index as a function of both the sampling frequency and the dynamical response of the control system. In [AS90], an online modification of the controller parameters is used to compensate for the implementation effects. Another solution proposed in [CE00] uses feedback from the current state of the tasks to improve the scheduling. Most of this research has been done at the scheduling stage, assuming periodicity of control tasks and therefore unnecessarily overconstraining the design. That is, the starting point for many codeign problems is already based on far-from-optimal design choices. In contrast, a first attempt to study self-triggered models for control tasks was developed in [VFM03], by discretizing the plant; in [LCHZ07], where the computation of the transition matrix is required, making the approach inefficient; and in [AT08], where the scheduling problem was not addressed. We claim that the periodicity assumption is not needed, as it leads to overly conservative designs. This claim is substantiated by previous work by the authors on the real-time requirements of control tasks, reviewed in Section III, and by a modification of the control server, proposed in this paper, that exploits the benefits offered by aperiodic self-triggered models for control tasks.

In addition to advocate the use of aperiodic self-triggered models for control tasks, our contribution is twofold: a modification of the control server to utilise the advantages of the self-triggered model for control tasks; and a particular choice of interface between the control task and the real-time scheduler that facilitates the codesign. This interface allows an online modification of the relative deadlines under overload conditions while preserving stability and performance. We finally illustrate the proposed techniques on the control of two jet engine compressors.

II. PROBLEM STATEMENT

The starting point is a control system:

\[
\dot{x} = f(x, u), \quad x \in \mathbb{R}^n, \quad u \in \mathbb{R}^m \quad (II.1)
\]

for which a feedback controller:

\[
u = k(x) \quad (II.2)
\]

has been designed, rendering the closed loop system \(\dot{x} = f(x, k(x))\) stable. The feedback control law (II.2) is typically implemented in a digital platform by measuring the state \(x\) at time instant \(t_i\), computing \(u(t_i) = k(x(t_i))\), and updating the actuator values at time instant \(t_i + \Delta_i\), where \(\Delta_i \geq 0\) represents the time elapsed between the sensor measurement of the state to the update of the actuators.

The problems we are trying to solve can now be posed as follows.

- How can we adjust deadlines, for control tasks, online so as to guarantee performance and reduce processor usage?
- Once deadlines for the control tasks are set, how can we schedule these tasks in a real-time environment?
- How can we define a simple interface between control tasks and schedulers that facilitates system codesign?

To tackle these issues, we will explore the real-time requirements of control tasks discussed in [Tab07] and reviewed in the next section.

This research was partially supported by the National Science Foundation EHS award 0712502 and Mutua Madrileña Automovilista.
III. Event-triggered stabilization of linear systems

Although the results of this paper apply to nonlinear systems, we shall review the event-triggered stabilization in a linear context for simplicity of presentation. In the linear case, the control system defined in (II.1) becomes:

\[ \dot{x} = Ax + Bu \]  

(III.1)

and is asymptotically stabilized by a linear feedback:

\[ u = Kx \]  

(III.2)

The dynamics of the closed loop system under the controller \( u = Kx(t_i) \) is given by:

\[ \dot{x}(t) = Ax(t) + BKx(t_i) \]

\[ = (A + BK)x(t) + BK\sigma(t) \]  

(III.3)

where the measurement error \( e \) is defined by:

\[ t \in [t_i + \Delta t, t_{i+1} + \Delta t] \Rightarrow e(t) = x(t_i) - x(t) \]

Since (III.2) is a stabilizing controller, it is well known from control theory that there exists a Lyapunov function \( V \) satisfying:

\[ \dot{V} \leq -a|x|^2 + b|e|c \]  

\[ a, b > 0 \]  

(III.4)

where \(|·|\) denotes the Euclidean norm. If we restrict the error to satisfy:

\[ b|e| \leq \sigma a|x| \]  

(III.5)

the dynamics of \( V \) is bounded by:

\[ \dot{V} \leq (\sigma - 1)a|x|^2 \]

thus guaranteeing that \( V \) decreases provided that \( \sigma < 1 \). In the context of nonlinear systems, equation (III.4) becomes:

\[ \dot{V} \leq -\alpha|e| + \gamma(|e|) \]  

(III.6)

where \( \alpha \) and \( \gamma \) are strictly increasing continuous functions with \( \alpha(0) = \gamma(0) = 0 \); and (III.5) is replaced by:

\[ \gamma(|e|) \leq \sigma \alpha(|x|) \]

(III.7)

to preserve stability of the control loop. Inequality (III.5) can be enforced by executing the control task whenever:

\[ |e| = \sigma \frac{a}{b}|x| \]

(III.8)

Every time the control task is executed, the current state is measured, making \( x(t_i) = x(t) \) which implies \( e(t) = x(t_i) - x(t) = 0 \) and thus enforcing (III.5). Equality (III.8) generates a sequence of deadlines at which the control task has to be executed in order to guarantee stability. This strategy leads to a lower number of executions than the conservative periodic task model, since the controller is only updated when it is indeed required. The parameter \( \sigma \) represents the rate of convergence of the dynamical system and at the same time it determines how frequently the controller will be updated. Thus this parameter \( \sigma \) represents a simple abstraction of the control performance that will facilitate the codesign.

IV. Self-triggered stabilization of nonlinear systems

An event-triggered implementation based on equality (III.8) would require testing (III.8) frequently. Unless this testing process is implemented in hardware, one might run the risk of consuming the processor time freed-up by using an event-triggered implementation to test (III.8). A better solution that we propose here is to use the current measurement of the state to set the next deadline for the task, that is, a self-triggered control task.

To find the sequence of deadlines \( \{d_i\} \) described by equality (III.8) we need to analyze the dynamics of the control system, that determines the evolution of the ratio \(|e|/|x|\). The procedure is described in detail in our previous work [AT08]. Due to space limitations, we briefly summarize the idea here:

- To derive a self-triggered condition, the relative deadlines of a control task should be expressed in terms of the measured state. At a particular state \( x(t_j) \), the relative deadline \( d(x(t_j)) \) is related to the deadline for another state \( d(x(t_i)) \) according to the formula:

\[ d(x(t_j)) = \chi(x(t_j)) \cdot d(x(t_i)) \]  

(IV.1)

where \( \chi(·) \) is a function that is determined by the dynamics of the closed loop system. This equation allows us to obtain a sequence of relative deadlines once the initial deadline is known.

- In order to apply equation (IV.1) online, it is necessary to find a deadline preserving stability for the initial condition. It was shown in ([Tab07]) that this deadline can be obtained from the following equation:

\[ \tau^* = \alpha_1 + \alpha_2 \arctan(\alpha_3 + \alpha_4 \cdot \tau^*) \]

(IV.2)

where each \( \alpha_i \) is a function of the dynamics of the system (II.1) and the controller (II.2).

- In equation (IV.1), if we let \( d(x(t_j)) \) be the next relative deadline \( d_j \) and \( d(x(t_i)) \) be the initial deadline \( \tau^* \), we obtain the following self-triggered condition to be used online:

\[ d_j = \chi(x(t_j)) \cdot \tau^* \]  

(IV.3)

Hence the deadlines depend on the current state and on \( \tau^* \), which is in turn a function of \( \sigma \), the control performance. The scheduler could modify online the value of \( \tau^* \) to adjust for the processor load or to optimize global performance.

V. Scheduling self-triggered control tasks

Most of the current scheduling techniques for control tasks assume periodicity, and tend to reduce latency and jitter. When designing controllers, it is difficult to deal with unknown delays but feasible to account for a constant input-output latency. One way to achieve this fixed delay (and to keep it as low as possible) is through the control server, introduced in [CE03]. Although the control server was developed for periodic control tasks, here it will be extended for sporadic tasks: hence we will work with densities rather
than dealing with utilization factors. We assume at the outset preemptive EDF scheduling in a uniprocessor system.

A. Schedulability

Two categories of tasks are considered:

- Control tasks $C_i$ that appear herein as sporadic tasks. As it was mentioned before in equation (IV.3), the deadlines are functions of the control performance, and any value of $\sigma$ less than 1 guarantees stability. Hence we can talk about a range $[\hat{\sigma}_i, \tilde{\sigma}_i]$ of possible deadlines associated with a range $[\hat{\beta}_i, \tilde{\beta}_i]$ of allowed performance. Here $\hat{\sigma}_i$ represents the desired performance and $\tilde{\sigma}_i$ is the lowest performance allowed (i.e., maximum value of $\sigma$).

- Other hard tasks $O_i$, either periodic or aperiodic.

Each hard task is comprised of a string of jobs $\{J^k_i\}_{k \in K} = \{J^1_i, J^2_i, \ldots\}$ with execution times $\phi^k_i$, relative deadlines $d^k_i$, density $\hat{\beta}^k_i = \phi^k_i / d^k_i$ and instantaneous utilization $\max_k \beta^k_i$. For the control tasks, instead of a fixed deadline $d^k_i$ we have the range $[\hat{d}^k_i, \tilde{d}^k_i]$ and the corresponding density range $[\hat{\beta}^k_i, \tilde{\beta}^k_i]$. It is well known that this set of tasks is schedulable if the total sum of the instantaneous utilizations is less than 1:

$$\sum_{i} \max_k \beta^k_i + \sum_{O_i} \max_k \beta^k_i \leq 1 \quad (V.I)$$

It is straightforward to check schedulability under this setup since an upper bound for the density $\hat{\beta}^k_i$ is known. To achieve a fixed latency we resort to the control server, that is briefly reviewed in the next section.

B. The control server

To reduce the latency, the job of a control task $J^k_i$ may be split into several segments $\{S^k_{ij}\}_{j \in J} = \{S^k_{i1}, S^k_{i2}, \ldots\}$. Each segment is assigned a relative deadline $d^k_{ij}$ (or length) according to:

$$d^k_{ij} = \frac{\phi^k_{ij}}{\hat{\beta}^k_i}$$

where $\phi^k_{ij}$ is the computation time of segment $j$. This assignment of deadlines preserves the density of the job of the control task while achieving a shorter latency (that is in fact the length of the corresponding segment).

We extend this concept to reduce even further the latency of the control tasks: if there is some available time in the CPU, density $\hat{\beta}^k_i$ can be increased in order to decrease $d^k_{ij}$, as shown in Figure 1. This procedure creates an artificial segment $(S^k_{i3})$ in the diagram with an assigned density $\tilde{\beta}^k_i$ (since density has to be the same for all segments of a task), and this spare segment can be allotted to low priority tasks. More precisely, let the total density of the task set be $\Gamma^k = \sum_i \beta^k_i$. Hence the spare density becomes $\Delta \Gamma^k = 1 - \Gamma^k$ and it could be split between the $n$ control tasks to increase their density (and thus reducing the latency). For instance, if we consider different weights $\omega_i$ for each of the $n$ control tasks, the new densities will be given by:

$$\beta^k_{i_new} = \beta^k_i + \frac{\omega_i}{\sum_i \omega_i} \Delta \Gamma^k$$

Thus the new latencies become $d^k_{ij_{new}} = \beta^k_{i_{new}} / \beta^k_i$. This strategy preserves schedulability while reducing delays in the control loops. At the same time, the scheduler could modify the value of $\sigma$ in order to allocate more resources for high priority tasks or to accept new incoming tasks.

![Fig. 1. Reducing latency with the control server](image)

VI. Example

To illustrate the benefits of the previous approach, we consider a computational unit in charge of the control of two jet engine compressors. The processor is also executing a (hard) periodic task in addition to several (soft) aperiodic tasks. A simple FIFO queue handles the soft tasks. The first step in the analysis consists in the design of the controllers. We borrow the following model of a jet engine compressor from [KK95]:

$$\dot{\phi} = -\psi - \frac{3}{2} \phi^2 - \frac{1}{2} \phi^3$$

$$\dot{\psi} = \frac{1}{\beta^2} (\phi - \phi_T) \quad (VI.I)$$

where $\phi$ is the mass flow, $\beta$ a constant positive parameter, $\psi$ is the pressure rise and $\phi_T$ corresponds to the throttle mass flow. A control law $\phi = g(\phi, \psi)$ is designed to render the closed loop globally asymptotically stable. The closed loop equations are:

$$\dot{\phi} = -\frac{1}{2} (\phi^2 + 1)(\phi + y)$$

$$\dot{y} = -(\phi^2 + 1)y$$

where we have applied the nonlinear change of coordinates $y = 2 \phi^2 + \psi$. Applying equation (IV.3), we obtain the following formula describing the relative deadlines for the control task:

$$d_{i+1} = \frac{29\phi(t_i) + y^2}{5.36r\phi(t_i)^2 + r^2} \cdot r^* \quad (VI.2)$$

where $r$ is the norm of the previously measured state $(\phi(t_i), y(t_i))$ and $r^* \in [0.3ms, 9.2ms]$ (computed from (IV.2)) to preserve the stability of the system. The computation time for each control task is 2ms. The operation region will be a ball of radius 5 centered at the origin. In order to show the effectiveness of the approach, we consider 50 different initial conditions equally distributed along the boundary of the operation region. Let the desired performance be $\sigma = 0.33$ for both systems. This implies that the relative deadlines generated by equation (VI.2) are lower bounded by $d_{i+1} \geq 7.63ms$, and thus density are upper bounded by $\beta \leq 0.26$. The hard periodic task has period
\[
T_p = 5\text{ms} \text{ and computation time } C_p = 1\text{ms}. \text{ We check the schedulability of this set of tasks:}
\]
\[
\sum_{C_i} \beta_i + \beta_{pcr} \leq \frac{2}{7.63} + \frac{2}{7.63} + \frac{1}{5} = 0.72
\]

Since we still have some spare density \( \Delta \Gamma = 0.28 \), we can take advantage of the control server properties to reduce the latency in both control tasks. Density for each task can be increased in \( \Delta \Gamma = 0.14 \), leading to a reduction of 34\% in the latency.

In Figures 2 and 3 we compare the behaviour of both strategies, periodic and self-triggered. To choose a stabilizing period for our system, we select the worst case relative deadline obtained from (VI.2) (other procedures could be applied, leading to similar values). A disturbance is applied at \( t = 0.7s \) to both control systems to check the robustness of our strategy. Both systems exhibit a similar behaviour for the state variables for any initial condition (see Figure 2 for one particular initial condition). Figure 3 shows the evolution of the input for the control system. At the beginning, both the periodic and aperiodic use the same relative deadline, but as the system tends to the equilibrium point the aperiodic policy increases the time between executions, whereas the periodic policy keeps updating the controller at the same rate. The right side of Figure 3 zooms the last part of the simulation, where the inter-execution times for the aperiodic strategy is already 24 times larger than the periodic. Hence the self-triggered implementation leads to a much smaller number of executions, while achieving a similar performance. The number of executions required under the control server strategy for both implementations are shown in Table I, for different values of \( \sigma \) (and averaged over all initial conditions considered): the aperiodic policy executes the controller nearly 8 less times than the periodic for a simulation time of 3s. Finally, Figure 4 shows the schedule for the first second. At the beginning, both control tasks require more CPU time so the queue with the soft tasks is always full; then, inter-execution times tend to enlarge as the system tends to the equilibrium point, giving more CPU time to the soft tasks. At \( t = 0.7 \) the disturbance steers the system far from the origin, and therefore the CPU reduces the deadlines accordingly to guarantee the required performance at the expense of delaying other soft tasks.

**Table I**

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>periodic</th>
<th>self-triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.11</td>
<td>890</td>
<td>119</td>
</tr>
<tr>
<td>0.22</td>
<td>506</td>
<td>66</td>
</tr>
<tr>
<td>0.33</td>
<td>397</td>
<td>51</td>
</tr>
</tbody>
</table>

**References**


Mapping Overlay Networks for Real-Time Applications

Jawwad Shamsi and Monica Brockmeyer
Wayne State University, Detroit, MI, USA.
{jshamsi , mbrockmeyer}@wayne.edu

Abstract

QoSMap is an overlay mapping scheme which is highly feasible for real-time applications with stringent per-hop requirements. It is built upon two goals: (i) To construct overlays that bear high QoS, and (ii) to increase resilience against QoS failures. Both the aims are critical for real-time applications. In order to achieve the first goal, QoSMap considers only direct underlay links as an overlay path and promotes paths that provide high QoS – where QoS is computed according to the user specified criteria. For the second goal, QoSMap specifically constructs backup paths that meet application constraints. Each backup path consists of an intermediate node and is utilized upon the QoS failure of its primary path. We evaluated the performance of QoSMap through PlanetLab experiments and observed that it successfully achieves its goals.

Keywords: Construction of Real-time Overlay Networks, Quality of Service.

1. Introduction

Overlay networks are increasingly used for Internet-based distributed systems. A wide variety of examples exist such as BitTorrent [3] for file sharing, M-bone [4] for multicast and PlanetLab [2] for evaluation platforms. However, the use of overlay networks have remained limited for real-time applications – largely due to Internet’s inability to satisfy timing constraints of real-time application. Applications such as collaboration environments, distributed gaming and simulation and high performance computing require timing guarantees which are dissuasive for Internet style best-effort communication.

A major reason for Internet’s inability to be a hot-spot for real-time applications is its variable communication characteristics. That is, the network characteristics of the Internet paths vary over time. As a consequence, the network constraints from the real-time application that are satisfied initially (during overlay formation) may be breached during the execution of an application. This may force the real-time application to halt the operation. In such a scenario, the overlay must be reconfigured [9] [5] to meet the stringent QoS demands of the real-time application. Since overlay reconfiguration is an expensive operation, which involves service interruption, overlay re-computation and application deployment, the allure of Internet based real-time overlay remains minimal.

Another challenge in meeting demands of real-time applications is that they often have hop-related network constraints such as latency and loss rate. For a hop-related constraint, the value of the network characteristic of an overlay path is aggregated with each underlay hop. Thus, each underlay hop in the overlay path decreases the quality of the path and affects the performance of the application. For such applications, it is preferred that a direct path should be considered, i.e. a direct link between the source and the destination nodes in the underlying network, such that the network characteristic related to the QoS constraint can be obtained directly from the monitoring service.

In this paper, we are inspired by the above mentioned challenges. To this end, we utilize QoSMap – a QoS aware overlay mapping algorithm which is highly feasible for real-time applications. QoSMap implements two approaches to meet the challenges: First, it satisfies the hop-related constraints of an application and strives to maximize the QoS by providing high quality paths. It only considers direct underlay paths as an end-to-end overlay path. Second, in order to extend the lifespan of the overlay (a feature desired by real-time applications) and reduce the cost and frequency of overlay reconfiguration, QoSMap computes supplemental backup routes that satisfy the QoS constraints of a real-time application.

1 QoSMap considers a path as direct if the network characteristics are available directly from the monitoring service. Similarly, a path is considered indirect if the network characteristics are aggregated.
Each supplemental path consists of an intermediate node and can be utilized upon the QoS failure of its primary path.

We previously described QoSMap [8] and evaluated its performance for an application that has constraints of latency and loss rate. This paper is an extension of our work in which we evaluate QoSMap for a real-time application. We conduct experiments on PlanetLab and utilize stringent per-hop QoS constraints of upper bound on latency and (the upper bound) violations as constraints from the application. Both the upper bound and the violations serve as a soft-guarantee of synchrony to the application. The goal of QoSMap was to provide overlays to the applications that can meet these stringent QoS constraints which are specific to real-time domain. We compared the performance of QoSMap with a simple QoS approach which do not specifically constructs supplemental paths or maximizes quality and observed that QoSMap yields more resilient and high quality overlays.

2. The QoSMap Approach

The joining application specifies its desired overlay topology and required real-time constraints along with their weights. QoSMap combines the constraints and their weights to form a metric \( M \) which specifies the quality of a path.

QoSMap is focused on meeting two specific goals. (i). To select overlay paths that bears high quality with respect to application specific criteria, and (ii) To increase resilience against QoS failures and reduce the cost and frequency of overlay re-configuration, thereby extending the lifetime\(^2\) of the overlay.

In order to achieve its first goal, i.e. paths with high quality, QoSMap evicts the links that do not meet the application constraints. It then prepares a list of underlay nodes that fulfill the degree requirements of the application, in that only the direct links between the underlay nodes are considered. From the filtered list, QoSMap maps the direct underlay links as overlay paths, while preferring the paths with high quality (\( M \)). Since a real-time application may have hop-specific constraints (such as latency and loss rate) in which the overall network characteristic of the path is the aggregate of its network characteristic at each hop, the consideration of only the direct links (single-hop) allows QoSMap to select paths with high quality.

For the second goal, i.e. increased resiliency against QoS failures, QoSMap builds supplemental paths that fulfills the application requirements and can be utilized upon the QoS failure of the direct path. For each supplemental path, QoSMap selects an intermediate node such that the supplemental path consists of two hops: from source node to the intermediate node and from intermediate node to the destination node. To reduce the number of extra nodes needed for supplemental routes, QoSMap prefers to select an intermediate node which is already included in the overlay as a previously mapped node or as an intermediary node for some other path. Supplemental paths must also satisfy application constraints and bear high quality. The detailed algorithm is explained in our previous paper [8].

The backup path method adopted by QoSMap in case of QoS failure is different then the backup path approach adopted by RON [1] in many ways. Foremost, RON is a fully connected overlay in which paths via intermediate nodes already exist. Contrary to that, QoSMap specifically constructs backup paths that meet QoS demands. In addition, QoSMap utilizes backup paths upon QoS failure - a scenario which is totally different than the RON’s goal of overcoming network outages.

3. Evaluation

In order to evaluate the performance of QoSMap under strict real-time requirements, we were motivated by PSON (Predictable Service Overlay Networks) [6]. The goal of PSON is to provide a communication infrastructure which provides bounded communication, as well as an estimate of upper bound of latency (maximum expected latency) to the application. QoSMap has been designed as a component of PSON. In PSON, the bound along each path (of the overlay) is constantly updated according to the measured latency and loss rate [7] and is reflective\(^3\) of the network characteristics of the path. The bound implies an assurance of synchrony (or predictability) in communication. Due to the volatile nature of the Internet, the bound can only serve as a soft guarantee in which it may be violated i.e., latency may exceed the bound (before the bound could be adjusted). The goal of PSON is to minimize the number of violations, while maintaining a low upper bound cost (difference between the upper bound and the latency).

\(^2\) Lifetime of the overlay is the duration from the overlay formation to the instant where a QoS failure occurs in the overlay such that it must be reconfigured to continue operation.

\(^3\) A too high or too low bound will affect the performance of the application.
Together, the bound and the violations indicate a level of synchrony or predictability an application receives from the network [6]. Several applications such as collaboration environments, distributed gaming and simulation and high performance computing can benefit from synchrony in order to properly admit a solution or exhibit improved performance.

In order to estimate upper bound on the Internet paths, we used SyncProbe [7] to continuously measure the bound and the violations across 20 PlanetLab nodes for 30 hours. Since each node estimated its upper bound to every other node this gave us a set of 380 paths. While our initial set of paths consist of 380 paths, we varied the QoS requirements from the application such that the resultant set is less-connected, after filtering the QoS-incompliant paths.

We considered five different types of QoS requests with varying upper bound acceptability of 100ms, 150ms, 200ms, 250ms and 300 ms, and fixed violation tolerance of 0.05% from the application. The weight of upper bound was set to 0.8, whereas the weight for violations was set to 0.2. For each level of QoS constraints, we considered five different overlay topologies: a completely connected overlay, randomly connected overlays with 50% and 25% connectivity, a tree topology and a ring topology, each having eight nodes (the tree topology has seven nodes). Overall, the five topology and the five QoS requests combined to form 25 different application requests. For each QoS request, a failure occurs if any of the mapped paths in the overlay exceeds the tolerance level of upper bound or violation. At that instant, a supplemental path that satisfies the QoS requirements must be used or the overlay should be reconfigured.

For performance comparison with QoSMap, we utilize a simple QoS approach which does not specifically constructs supplemental routes through intermediate nodes or maximizes QoS.

We used the collected data about the upper bound and violations for the 380 paths to fulfill the 25 QoS requests using both the QoSMap and the simple QoS schemes. During our experiments we observed that for the 100% connected overlay an overlay request cannot be fulfilled when the upper bound was 100 ms.

Following are the observations of our experiments related to the two goals of QoSMap, i.e. (i) achieving high QoS and (ii) increasing overlay lifetime.

\[\] However, in our analysis, we specifically check for the existence of supplemental backup paths for simple QoS.
Achieving High QoS: For both the QoSMap and simple QoS, we computed the average upper bound (over all overlay paths) for each of the 25 overlay requests. Since the upper bound represents a threshold or maximum tolerance level from the application, an overlay with low upper bound represents high quality. Similarly, an overlay with low rate of violations indicates high quality. During our experiments, we observed that the upper bound (on latency) for the overlays yielded by QoSMap is significantly lower as compared to the upper bound for the overlays from the simple QoS scheme. Further, the difference in the upper bound achieved by the overlays from the two schemes increases as the upper bound restrictions are relaxed.

We also noted that under most scenarios, the rate of violations achieved by the overlays of the two mapping schemes remains similar. That is SyncProbe (the upper bound estimation technique of PSON) [7] was able to keep a low rate of violations by rapidly adjusting the upper bound. Thus, most of the quality achieved was related in keeping low upper bound. Figures 1 and 2 illustrate the average upper bound achieved by the two mapping schemes.

Increasing Overlay Lifetime: To compute the lifetime of the overlay, we noted the difference in time between overlay formation and the instant where a QoS failure leads to overlay reconfiguration. We calculated the average lifetime of overlays from both the mapping schemes and observed that the existence of a large number of backup paths in the overlays from QoSMap averted the need for overlay configuration and increased the lifetime of the overlay. In comparison, overlays from the simple QoS experienced frequent QoS failures and a large number of reconfigurations. Thus, the average lifetime of the overlays from simple QoS was significantly lower than the average lifetime of the overlays from QoSMap. Figure 3 and figure 4 illustrate the results.

We also computed the cost of the resilience, which is computed as the number of extra nodes needed to achieve high resilience against QoS failures. While, the cost is zero for the simple QoS scheme as it does not constructs supplemental paths, the QoSMap approach requires intermediate nodes in order to construct backup paths. We observed that QoSMap was able to keep a low cost of resilience by utilizing nodes that are already included in the overlay. On average, the number of nodes needed by QoSMap varied from 0 to 1.5.

4. Conclusion and Future Work

We compared the performance of QoSMap with a simple QoS approach, under strict requirements of upper bound and violations. Our results indicate that the overlays yielded by QoSMap can be successfully used for real-time applications. The consideration of only the direct paths that promote high quality, allows QoSMap to meet the constraints of a real-time application and obtain high QoS, whereas the provision of backup paths increases the resilience against QoS failures and reduces the cost of service interruption and overlay reconfiguration.

Under some scenarios, a path with an intermediate node may provide higher quality as compared to the direct path. Paths with one intermediate node might also be useful, if the degree requirements of a node cannot be fulfilled through direct paths. At present, we are extending the algorithm to consider direct as well as indirect paths (with limited number of hops) to map the primary overlay paths. Such a consideration would permit a more cohesive approach in attaining high quality and meeting application requirements.

As a part of our future work, we will integrate QoSMap as an overlay construction mechanism for PSON [6]. We plan to deploy PSON as a service on a wide area platform (such as PlanetLab) and use it to construct overlays with more predictable and synchronous behavior.

5. References

Towards a Model-based Toolchain for the
High-Confidence Design of Embedded Systems

János Sztipanovits, Gábor Karsai, Sandeep Neema, Harmon Nine,
Joseph Porter, Ryan Thibodeaux, and Péter Völgyesi
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA
janos.sztipanovits@vanderbilt.edu

Abstract

While design automation for hardware systems is quite advanced, this is not the case for practical embedded systems. The current state-of-the-art is to use a software modeling environment and integrated development environment for code development and debugging, but these rarely include the sort of automatic synthesis and verification capabilities available in the VLSI domain. This paper introduces concepts, elements, and some early prototypes for an envisioned suite of tools for the development of embedded software that integrates verification steps into the overall process.

1. Introduction

Embedded software often operates in environments critical to human life and subject to our direct expectations. We assume that a handheld MP3 player will perform reliably, or that the unseen aircraft control system aboard our flight will function safely and correctly. Embedded environments require far more care than provided by the current best practices in software development. Often formal verification and system certification are required to insure correct behavior and conformance to legal standards. Embedded systems design challenges are well-documented [4], but industrial practice still falls short of these expectations.

Consider one style of modern development practice: graphical modeling and simulation tools (e.g. Mathworks’ Simulink/Stateflow or National Instruments’ Matrix-X) represent physical systems and engineering designs using block diagram notations for dataflows or state models. Design work revolves around simulation and test cases, with code generation following once the design is considered complete. Such methods frequently ignore software engineering constraints on the design and neglect issues that arise from embedded platform choices. At early stages of the design, often the platform is vaguely specified to the engineers as a set of possible tradeoffs, with incomplete details regarding actual platform function and performance.

Similarly, another development style uses UML (or similar) tools to capture software engineering concepts such as components, interactions, timing, fault handling, and deployment. These workflows focus on source code creation and management followed by testing and debugging on target hardware. In this case the physical and environmental constraints are not represented by the tools. At best such constraints may be provided informally as notes or documentation to developers and may remain poorly understood.

The interplay between these two prevalent development styles creates problems. Designers lack tools to model the interactions between the hardware, software, and the environment. For example, software generated from a carefully simulated functional dataflow model may fail to perform correctly when its functions are distributed over a shared network of processing nodes. Neither style of development supports comprehensive verification of certification requirements. To move towards a solution to these problems, we propose a suite of tools that address many of these challenges. Currently under development at Vanderbilt’s Institute for Software Integrated Systems (ISIS), these tools use domain-specific modeling languages (DSMLs) to integrate the disparate aspects of an embedded systems design.

The tool suite described here is built on the concept of platform-based design [8], and is shown conceptually in Figure 1. Componentization and higher-level services enable the designer to build correct systems from validated components. Additionally, if the DSMLs used in tool integration have formally defined behavioral semantics and well-defined models of computation (MoCs) for component interactions [7], system properties and models can be
expressed formally and verified with appropriate external tools. In the sequel we briefly describe the current state of the tool suite and conclude with a discussion of the direction of our future goals.

2. Elements of the Tool Suite

The domain of choice for this research is that of distributed and embedded control systems. Accordingly, the formal MoC chosen is that of the Time-Triggered Architecture (TTA) [6]. Time-triggered systems provide a number of essential guarantees for safety-critical control systems designs. In particular, the TTA provides precise timing for periodic tasks, distributed fault-tolerance, and replica determinism in redundant configurations. These basic guarantees and their implementations constitute some of the important high-level component services needed for our platform-based designs.

2.1. Software architecture specification

Simulink/Stateflow (SL/SF) models can be imported into a well-defined modeling format that allows for analysis, extension, and code generation. Graphical modeling tools can read these models and perform software engineering design tasks. The SL/SF models are embedded in software components with well-defined interfaces, and then mapped to well-defined distributed hardware models.

2.2. Code generation

Model transformations [3] can convert imported SL/SF models into a model representing an abstract syntax tree (AST) for C code fragments. Interpreters for the new AST model can create code or directly perform simple static analyses such as checking variable initializations. Generated C code is generic – the tools currently support execution on a hardware implementation of the TTA (hardware available from TTTech[2]) or on a time-triggered virtual machine (VM) running on Linux (described below).

2.3. Scheduling

Resource allocation in the TTA is controlled by a pre-generated cyclic schedule created from task specifications and their communication dependencies. We have created a simple schedule generation tool that uses the Gecode finite-domain constraint programming library to search for cyclic schedules that meet the specifications. Constraint models are an extension of earlier work in this area [9].

2.4. Modeling the execution platform

The chosen time-triggered model of computation has been formalized using the DEVS formalism (Figure 2) and simulated using the DEVS++ simulator [5]. Simulation results for a time-triggered triple modular redundancy experiment were consistent with observed performance of a time-triggered implementation [10].

2.5. Implementation of the execution platform

In addition to tests on available time-triggered hardware, we have developed a portable time-triggered VM running on a networked cluster of processors running standard Linux. The portability of the VM allows the direct exploration of the capabilities and limitations of the services provided by the underlying operating system, and the effects of those limitations on the guarantees provided by the chosen MoC [10].

3. Future work

As this research effort is a work in progress, we conclude with a brief summary of the next steps and future objectives for each of the tools presented. We must keep in mind the final goal of verifiable and certifiable software for embedded systems. This section contains forward-looking statements.

3.1. Software architecture specification

The chief limitation of our software architecture tools is the one-way design flow from the SL/SF design, through componentization, down to the final code. We aim to improve the ability to send design information back to the earlier stages of the design as needed. For example, platform-specific simulations may indicate that jitter or quantization
effects will impact the initial assumptions of a control design. Representing that data to control designers in a meaningful way will allow design changes without excessive workflow iterations. Schedulability is another area where downstream software design tools can provide meaningful feedback to the original design engineers.

3.2. Code generation

The abstract model in the code generator opens the door for a number of potential static analysis and verification opportunities. The current toolchain includes two code generators that produce C (and Java) source code from (single-rate) subsystems in Simulink and Stateflow models. The code generators have been implemented using graph transformation techniques, and they produce an AST from which the actual code is printed. To assist in system-level or functional code verification the AST could be extended to carry over information from the original model, thus providing guidance for the source code-level verification tool regarding the original model from which the code was generated and its properties. We believe this can significantly improve the performance of the verification step because the verifier does not have to reverse engineer the high-level abstractions from the source code, as the abstractions are readily available in the models.

3.3. Scheduling

We aim to expand the scheduling tools to include specific time-triggered models. One simple example is that of adding constraints to support the requirements of the TT-Tech TTP/C hardware. Another avenue for research is the exploration of interactions between the resource allocation model (via schedules) with other system objectives which can be modeled by constraint or optimization problems in other domains (such as continuous stability in the control design).

3.4. Extending the modeling of the execution platform

The formal DEVS model is a big step towards providing guaranteed safety and performance in time-triggered control system designs. DEVS also supports pure event-triggered behaviors in addition to timed models. Experimentation with this capability will hopefully lead to a better understanding of the limitations of heterogeneous component interactions in our system designs.

Platform simulation also opens up opportunities for exploration. The TrueTime tool suite from Lund University [1] extends Simulink models with concepts for modeling distributed platforms, scheduling policies, and communication protocols. TrueTime promises to help characterize behavioral changes due to the distribution of functionality over networked processors.

3.5. Extending the capabilities of the execution platform

As the capabilities of the formal models expand, we aim to extend our portable VM implementation to manage heterogeneous behaviors. The VM will also be ported to other operating platforms, including diverse hardware and RTOSes such as QNX and uC-OS. Different platforms provide different levels of assurance regarding timing, determinism, and resource management. These differences will need to be reflected in the models. New features may also be added to the VM as required to support interaction idioms such as remote procedure calls or rendezvous. We may also require additional component services such as health monitoring, fault management, robust clock synchronization, or failover.
4. Acknowledgements

This work was sponsored (in part) by the Air Force Office of Scientific Research, USAF, under grant/contract number FA9550-06-0312. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

Adding the Time Dimension to Majority Voting Strategies

Hüseyin Aysan, Sasikumar Punnekkat, and Radu Dobrin
Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
{huseyin.aysan, sasikumar.punnekkat, radu.dobrin}@mdh.se

Abstract

Real-time applications typically have to satisfy high dependability requirements and require fault tolerance in both value and time domains. A widely used approach to ensure fault tolerance in dependable systems is the N-modular redundancy (NMR) which typically uses a majority voting mechanism. However, NMR primarily focuses on producing the correct value, without taking into account the time dimension. In this paper, we propose a new approach, Voting on Time and Value (VTV), applicable to real-time systems, which extends the modular redundancy approach by explicitly considering both value and timing failures, such that correct value is produced at correct time, under specified assumptions. We illustrate the proposed approach by an algorithm applicable for triple modular redundancy (TMR).

1. Introduction

Most real-time applications typically have to satisfy high dependability requirements due to their interactions and possible impacts on the environment. Ensuring dependable performance of such systems typically involves both fault prevention and fault tolerance approaches in their design. Usage of redundancy is the key for achieving fault tolerance and it has been employed successfully in the physical, temporal, information and analytical domains of a large number of critical applications. Static techniques such as N-modular redundancy (NMR) have been used in safety and mission critical applications, most often in the well-known form of triple-modular redundancy (TMR), where three nodes are used for replication [9]. The key attraction of this approach lies in its low overhead and fault masking abilities, without the need for backward recovery. The disadvantages include the cost of redundancy and single point failure mode of the voter. Traditionally, voters are constructed as simple electronic circuits so that a very high reliability can be achieved. Usage of triplicated voters has been employed to take care of the single-point failure mode in case of highly critical systems [8]. Surveys and taxonomies on several voting strategies have been presented [7, 5].

Replicated nodes’ output delivery times can vary due to several factors, such as clock drifts, node failures, processing and scheduling variations at node level, as well as communication delays. Most of the existing voting strategies, however, focus solely on masking value failures by assuming that the system is tightly synchronized, as presented in [6]. On the other hand, loosely synchronized systems may be an attractive alternative due to, e.g., low overheads, requiring, however, specifically designed asynchronous voting algorithms to compensate for the timing variations.

A simple approach towards tolerating both value and timing failures in a replica using the NMR approach could be adding time stamps to the replica outputs. Then, majority voting on time stamp values could detect possible timing anomalies of the nodes, under the unrealistic assumptions that the communication is ideal and nodes never halt. Moreover, this approach is unable to mask late timing failures.

Shin and Dolter [11] proposed two voting techniques applicable to real-time systems, relaxing the tight synchronization requirements, viz., Quorum Majority Voting (QMV) and Compare Majority Voting (CMV). QMV performs majority voting among the received values as soon as \(2n+1\) out of \(3n+1\) replicas deliver their outputs to the voter, thus, guaranteeing detection of majority of non-faulty values even in the case \(n\) replicas fail. CMV masks failures of \(n\) out of \(2n+1\) replicas as in basic majority voting. The main difference is that in CMV the output is delivered as soon as a majority consisting of identical values has been received, i.e., without waiting for the rest of the replicas. Both QMV and CMV provide outputs within a bounded time interval, as long as the assumptions regarding the maximum number of failures hold. However, QMV and CMV are unable to detect assumption violations in the time domain.

In this paper, we propose a novel approach, Voting on Time and Value (VTV), which performs majority voting in both time and value domains. Our approach enhances the fault tolerance abilities of NMR by restricting the replica outputs to be both correct in value, and delivered within a
specified admissible time interval, under specified assumptions. Furthermore, our approach is able to detect assumption violations in time domain.

The rest of the paper is organized as follows: In Section 2 we present the system model and the assumptions used in this paper. Section 3 describes our approach, illustrates it by an instantiation to a system using triplicated nodes. We conclude the paper in Section 4 outlining the on-going and future work.

2. System Model

In this paper, we assume a distributed real-time system, where each critical node is replicated for fault tolerance, and replica outputs are voted to ensure correctness in both value and time. For the sake of readability, in the rest of the paper, we denote the $i^{th}$ replica of a node $N$ by $N_i$. The output delivered by $N_i$, is specified by two domain parameters, viz., value and time $[1, 10, 3]$:

Specified output for $N_i = < v_i^*, t_i^*, \Delta_v, \Delta_t >$

where $v_i^*$ is the correct value, $t_i^*$ is the correct time point when the output should be delivered, $[v_i^* - \Delta_v, v_i^* + \Delta_v]$ is the admissible value range and $[t_i^* - \Delta_t, t_i^* + \Delta_t]$ is the admissible time interval for output delivery as per the real-time system specifications.

An output delivered by $N_i$ is denoted as:

Delivered output from $N_i = < v_i, t_i >$

where $v_i$ is the value and $t_i$ is the time point at which the value was delivered.

We define the output generated by replica $N_i$ as incorrect in value domain if $v_i < v_i^* - \Delta_v$, or $v_i > v_i^* + \Delta_v$, and incorrect in time domain if $t_i < t_i^* - \Delta_t$ (early timing failure), or if $t_i > t_i^* + \Delta_t$ (late timing failure).

Assumptions: Our approach relies on the following set of assumptions (to a large extent based on [4]):

1. non-faulty nodes produce values within a specified admissible range and within a specified time interval after each computation block
2. replica outputs with incorrect values do not form majority
3. incorrectly timed replica outputs do not form majority
4. a maximum permissible drift $\delta$ from the global time is specified and ensured by infrequent synchronization (which is significantly less costly than tight synchronization)
5. the voter does not fail.

3. Voting on Time and Value (VTV)

In this section we present our novel voting strategy that explicitly considers failures in both time and value domains. As a consequence of assumption 5, in the worst case, the maximum deviation between any two replica outputs is $2\delta$. Hence, in VTV approach, agreement in the time domain is reached when a majority of replicas deliver their outputs within this derived time interval of $2\delta$ (referred to as feasible window henceforth). If a node has $n$ replicas, then at least $m = \lceil \frac{n+1}{2} \rceil$ outputs from these replicas need to match for establishing majority. The number of groups with $m$ sequential replica outputs within $n$ replica outputs is $n - m + 1$. Since the majority in time domain can be formed by any of these groups, a separate feasible window needs to be initiated upon receiving each of first $n - m + 1$ replica outputs. We keep track of the feasible windows by using simple countdown timers. Once an agreement in time domain is obtained, then values are voted. If an agreement in value domain is not obtained for a particular feasible window, the process continues with subsequent feasible windows, until a majority in time and an agreement in value can be formed, or an assumption violation is detected.

![Figure 1. Replica output flow through voter](image)

Depending on the real-time application characteristics, a value produced by a node may be considered valid or invalid for the purpose of voting, in case it is produced early. An illustration of replica output flow through the voter is given in Figure 1. An issue is the choice of the set of valid values to be used in the voting mechanism, i.e., all received values vs. all timely received values. We illustrate this voting dilemma by using the scenario described in Figure 2. Let us assume, e.g., an airbag control system where a sensor is replicated in five different nodes and produces one out of two values periodically, e.g., value $a$ in case of a collision detection and value $b$ otherwise. If a collision is detected at a time $t \leq t_1$ let us assume that the airbag has to inflate within a time interval $[t_{start}, t_{end}]$, where $t_2 < t_{start} \leq t_3$.
and \( t_5 \leq t_{\text{end}} \). In our example, the first two values are detected as early and the last three are identified as timely. However, in this case, an early value has to be taken into consideration in the voting since an early collision detection is still a valid output with respect to the value domain. Thus, the output has to be voted upon receiving the last value at time \( t_5 \), among all values, i.e., \( a, a, a, b, \) and \( b \), resulting in an output \( a \) at time \((t_5 + \epsilon)\) (where \( \epsilon \) is the time required for the voting and is assumed to be negligible in this paper for simplifying the presentation).

On the other hand, let us assume that the same Figure 2 illustrating an altitude measurement sensor in an airplane, replicated by five nodes to read and output the altitude periodically to the voter, where data freshness may be a more desirable aspect. As the correct window of time for the output is the same as described in the previous example, the only relevant values to be taken into consideration by the voter are \( a, b, \) and \( b \) corresponding to the time points \( t_3, t_4, \) and \( t_5 \) respectively. Hence, the output produced at time \((t_5 + \epsilon)\) is \( b \).

![Figure 2. Voting dilemma](image)

Upon finding a feasible window, if a majority in value domain is obtained with all the values received so far, the voter delivers the majority value without waiting for the rest of the replicas. Otherwise, either a majority in value domain, receipt of all replica outputs, or the end of the feasible window is waited for, whichever comes first. If a majority in value domain is obtained while waiting, it is delivered as the correct output. The decision on whether the early generated replica outputs are involved in value voting or not results in two cases at this point:

**Case 1 Early and timely outputs are considered valid.** If the end of the feasible window is reached with a majority among the received values, it is delivered as correct output.

**Case 2 Only timely outputs are considered valid.** If the end of the feasible window is reached with a majority among the timely received values, it is delivered as correct output.

If the end of the feasible window is reached without an agreement in value domain, the process continues with a subsequent feasible window. If the last feasible window is reached, or all replica outputs are received without reaching an agreement on the values, disagreement is signalled to the rest of the system.

### 3.1 VTV in TMR

In this section, we present an instantiation of our approach to triple modular redundancy which can tolerate single node failures in value domain, time domain or both (Algorithm 1). In this example, we assume early timing failures as invalid for the purpose of voting. However, the validity of such values can be easily tuned in the algorithm.

Majority in time domain is achieved if at least two values are delivered to the voter within a time interval less than or equal to \( 2\delta \), since this is the maximum deviation in time among all the values as long as there is no failure. Majority in value domain is formed if at least two of the timely outputs have the same value.

The algorithm signals disagreement in case majority condition is not satisfied in any of the domains, thus enabling a fail-safe or fail-stop behavior of the system.

The replicated nodes’ output values are stored in local variables \( V_1, V_2 \) and \( V_3 \). Values are assigned to these variables in the order of receiving inputs from the nodes (i.e., the first received value is stored in \( V_1 \), the second one in \( V_2 \) and the last one in \( V_3 \)). Two countdown timers, \( C_1 \) and \( C_2 \), initially set to \( 2\delta \), are used to keep track of feasible windows in order to identify majority in time domain.

The algorithm waits for the first node output to be delivered and then starts \( C_1 \). It continues by waiting for the second node output and starts \( C_2 \) upon its arrival. If both values have arrived before \( C_1 \) expires, and have matching values, the voter will output the correct value. Otherwise we have two cases:

**Case 1** \( C_1 \) has not reached zero, but the values \( V_1 \) and \( V_2 \) do not match. In this case, the algorithm waits for \( V_3 \) until \( C_1 \) reaches zero. If the third value arrives before \( C_1 \) reaches zero and matches either \( V_1 \) or \( V_2 \), the algorithm outputs the matching value since all values are timely and there is an agreed value. In case of assumption violation, i.e., there exists no replica output pair matching in value domain, the algorithm signals disagreement. If the third value does not arrive before \( C_1 \) reaches zero, the algorithm waits for \( V_3 \) until \( C_2 \) reaches zero. If \( V_3 \) is received and matches \( V_2 \) before
C2 reaches zero, the algorithm outputs the matching value. Otherwise the algorithm signals disagreement.

Case 2 C1 has reached zero. In this case, V1 is considered invalid, and the algorithm waits for V3 until C2 reaches zero, as only a match between V2 and V3 may result in an agreement. If the values do not match or V3 has not been received at all, the algorithm signals disagreement.

4. Conclusions

In this paper we have presented a new voting strategy called Voting on Value and Time (VTV) for redundant systems, to explicitly consider both value and timing failures for achieving fault tolerance in real-time applications. Under specified failure assumptions, our method is capable of producing the correct output as well as identifying the correct window of time in which the output has to be delivered.

We have presented an algorithm for the particular case where one output is replicated in three different nodes, and illustrated the basic idea on how we perform the voting in both value and time domain.

Our ongoing research indicates that VTV, when used in the general case to mask arbitrary number of value and timing failures, is cost-effective in comparison with the number of nodes required by majority voting in NMR. The main reason is that, in our approach, a non-faulty node can be successfully used to mask both a value and a timing failure in the voting procedure.

References

An Experimental Model for the Verification of Dynamic Voltage-Scaling Scheduling Techniques on Embedded Systems

William Wiles  Gang Quan
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208
{wilesw, gquan}@cse.sc.edu

Abstract

Tremendous theoretical research efforts have been made in the past decade to address the stringent real-time constraints and soaring power consumption challenges in embedded systems. However, the experimental work that can validate and evaluate the applicability and effectiveness of these theoretical results is very limited, largely due to the lengthy and challenging process in the design and development of the experiment infrastructure. In this paper, we present a general experimental model using Linux and a commercial off-the-shelf embedded platform as a proving-ground for real-time scheduling algorithms with a specific focus on techniques that take advantage of the dynamic voltage scaling (DVS) capabilities of modern processors. With this model, system designers have the capability to plug in arbitrary real-time schedulers easily into the kernel and run real-time tasks at the user level applying the desired scheduling techniques. Three well-known priority-driven real-time scheduling algorithms are implemented to study the capability and potential of this model.

1 Introduction

Currently there is a strong correlation between real-time devices and embedded devices in today’s marketplace. Embedded technologies have enjoyed increased computational performance from more advanced microprocessors, but in contrast, less impressive improvements in energy capacity from mobile power supplies. This increasing gap has led to an increased desire in energy efficient computing to prolong battery life. Also, the advantages from energy-efficient computing can transcend mobile devices, with improved energy efficiency also comes improved thermal performance, a benefit for all high-performance devices. In this manner, it is easy to see that as the industry continues the natural progression of increasing computational power, power-aware scheduling algorithms will only become more relevant in more areas of computing.

The theoretical foundation for power-aware scheduling is very extensive (e.g. see [9]). While simulation is strong for showing trends in an algorithm’s performance relative to a baseline or some other benchmark, a simulation is only as good as the system model it is designed to emulate. Due to the complexity of hardware devices today, behavior may arise on a hardware system that is unaccounted for in a simulation model that is important enough to give insight into modifications that improve upon ideal algorithms. In contrast to simulations, experimental evaluation allows us to strip away the idealized environment and see real-world performance that takes into account any important details that may not have been considered in theoretic models. The cornerstone of this type of analysis is that it allows us to place our algorithms and research in an active environment so that we can study the performance and identify possible physical factors that cause our results to deviate from expectations. Through this we gain a better understanding of the environment, and can increase performance and improve our models.

It is therefore our goal to develop a general purpose, power-aware, real-time testing environment based on commercial off-the-shelf hardware to be used for evaluation of a large range of scheduling algorithms. Work done in this field involve varied platforms ranging from non-embedded generalized hardware to custom designed devices. Due to the popularity of the i386 instruction set, the general personal computing platform has been very popular for evaluating algorithms, such as Pillai and Shin’s validation of novel DVS algorithms [20]. While at the other end of the spectrum, analysis has been done on custom embedded designs such as the IBM PPC 405LP by Anantaraman et.
al. [4], or the fully custom Low-Power StrongARM (LART) used by Pouwelse [21]. These custom solutions offer advantages over more commercially available embedded systems with respect to real-time operation; however, since the availability of these devices is limited, a general platform is better suited to be based on hardware more easily obtainable. The middle ground offers what can seem to be the worst of both worlds, lacking the optimized performance of custom designs with less popular instruction sets than the general i386, that do not readily support a large library of applications. However this middle area of off-the-shelf hardware is the most popular and readily available of embedded devices, hence the name. Rajkumar et al. has developed a LinuxRK operating system on these commercial devices, specifically the Compaq iPaq and ADS BitsyX [22], and we seek to take another step further in generality within this commercial environment.

In this paper, we propose a general framework to develop a test environment where various power-aware real-time scheduling algorithms can be easily compared and validated on equal footing, while taking into consideration that integrating a new scheduler into an existing operating system is non-trivial. Essentially we offer a Linux kernel for users to hot-swap any scheduling algorithm desired. Alongside this in user space real-time tasks are issued and ‘elevated’ to real-time through new system calls into the kernel. Based on our framework, we develop a test environment based on a widely available, commercial off-the-shelf embedded platform, i.e. the ARM based BitsyXb running Linux 2.6.17. We further implement three popular real-time scheduling algorithms: the rate monotonic scheduler (RMS), earliest deadline first (EDF) scheduler, and Yao’s optimal offline EDF derivative [29] (henceforth referred to as lowest-power earliest deadline first LPEDF), to study the capability, limitations, and potential of this platform in more general experimental study.

The structure of the paper is as follows, we introduce our framework in Section 2), followed by our experiments and results (Section 3) and a short summary (Section 4).

2 The General Framework

The general framework is built upon the existing 2.6 Linux kernel. We choose Linux for its generality and flexibility. Extremely flexible, open source, operating systems like RTAI [11], Xenomai [12], and eCOS [1] allow for a very comfortable manipulation of pre-existing source, however have a much stronger compatibility with i386 desktop/laptop platforms than with low-power embedded devices. Conversely, commercial operating systems like MontaVista Linux [3] along with the proprietary Microsoft Windows CE have varying degrees of flexibility concerning source code manipulation, none of which supported to the degree of the open source community projects.

Figure 1 is an overview of our model, and highlights the generality goal mentioned earlier. Within our design, all of the interfaces to hardware, noted in the figure as the Hardware Abstraction Layer (HAL), are maintained identically to the standard kernel implementation.

At the kernel space, the task_struct structure used to identify processes is amended to hold pertinent real-time information that separates between real-time processes and ordinary processes. A system call, i.e. promote_to_rt(), is also defined that allows a user to pass in a process to be associated as a real-time process with sufficient parameters. Users can define all of their required scheduling algorithms in the kernel scheduler, sched.c, and referring to a unique algorithm by means of defining a real-time policy in sched.h. In this manner, users will be able to use multiple scheduling algorithms concurrently, in a similar manner to the preestablished method in Linux. Modifications are also made to the scheduler to natively include frequency scaling functions from cpufreq.c to be utilized in DVS algorithms.

Within the user space, we implemented utilities to generate periodic/aperiodic process and allow process elevation to real-time execution, as well as verification for task deadlines. The periodic real-time task model is supported by a governing process that accepts a function or process to be elevated to a real-time task at a specified arrival, with an additionally specified period and deadline, we refer to this as the Task Issuer in Figure 1. Furthermore, this governing process also can be used for profiling purposes to determine a worst-case execution bound, should a particular scheduling algorithm need this information. With an established kernel and user space design as well as interfacing, the overall development cycle using the testbed can be described as shown in Figure 2.

Figure 1. Our Linux Model Overview.
3 Implementation and Experiments

The hardware platform chosen for this study is a commercial platform based on the Intel XScale PXA270 microprocessor, the ADS BitsyXb [25]. This device was chosen primarily due to the popularity of the ARM architecture in embedded devices today [5] in conjunction with the aggressive DVS capability of the PXA270. The PXA270 frequency is based off of a 13MHz system clock, and in terms of DVS, has five voltage/frequency steppings between a lower bound of 104MHz to an upper bound of 520MHz at 104MHz intervals. In addition, through correspondence with the manufacturer we were able to determine a reference point for the processor’s variable voltage, and a series resistor for current measurement through our DAQ [10] and therefore isolate the power consumption and energy usage for the processor specifically.

We take three scheduling algorithms inside of our model for use in testing the power consumption on the hardware platform. The selected algorithms are the classical rate monotonic scheduler [16] (RMS), earliest deadline first [16] (EDF) formalized by Liu and Layland, and the offline power-optimal scheduler LPEDF [29]. The LPEDF algorithm was chosen to highlight a best effort energy savings by the platform, which would provide a bound for online algorithms, should the designer decide to investigate further. Experiments were done using periodic task sets with a workload consisting of a high complexity, but very deterministic, operation, in this case: the matrix multiplication.

Using our task set, the LPEDF speed/voltage schedule is sketched in Figure 3, with the grey area indicating the speed schedule adjusted to the BitsyXb’s valid frequencies. Figure 4 shows the processor’s core voltage throughout the execution of the task set for each algorithm, with samples taken at 10 Hz. Our experimental results in Figure 4 concur with the model’s expected voltage schedule (Figure 3). Ta-

<table>
<thead>
<tr>
<th>$Task_i$</th>
<th>Period</th>
<th>Deadline</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1800</td>
<td>1300</td>
<td>720</td>
</tr>
<tr>
<td>$T_2$</td>
<td>2400</td>
<td>1200</td>
<td>180</td>
</tr>
<tr>
<td>$T_3$</td>
<td>3600</td>
<td>3000</td>
<td>180</td>
</tr>
</tbody>
</table>

Table 1. Task set for power evaluations (all units in ms).

![Figure 2. The Experimental Flow Overview.](image)

![Figure 3. LPEDF speed/voltage schedule (horizontal axis in ms. The vertical axis denotes the relative processor speeds with ‘1’ the maximal working frequency, i.e. 520Mhz).](image)
Figure 4. Processor voltage during operation.

<table>
<thead>
<tr>
<th></th>
<th>LPEDF</th>
<th>EDF</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1.9998</td>
<td>0.9796</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2. Energy savings, normalized to RMS.

ble 2 surmises the total energy savings for the interval, with LPEDF gaining nearly a two-fold relative energy savings over RMS. As expected, RMS and EDF perform similarly, since their voltage remains unchanged for the duration.

4 Conclusions and the future work

Power-aware scheduling will continue to play a critical role in more areas of computing. While the theoretical foundation is imperative to progression in the field, it is also important to build upon this foundation with solid experimental verification. In this paper, we present a general experimental model using the Linux framework and a commercial off-the-shelf embedded platform. With functional scheduling algorithms implemented in a hot-swappable fashion, the model provides a robust and consistent method to investigate the algorithm in practical scenarios. For the future work, we would further improve at the kernel to reduce the task context-switching latency. Further, more theoretical results will be tested using this platform. Finally, to extend this platform and model to a SMP architecture would be an interesting problem and worth further study.

References

[23] S. Souhlal. Make the tsc safe to be used by gettimeofday(), 2005.
Toward an Effective Execution Policy for Distributed Real-Time Embedded Systems

Thomas Huining Feng, Edward A. Lee, Hiren D. Patel, and Jia Zou
Center for Hybrid and Embedded Software Systems, EECS
University of California, Berkeley
Berkeley, CA 94720, USA
{tfeng,eal,hiren,jiazou}@eecs.berkeley.edu

Abstract—Zhao, Liu, and Lee have proposed using a discrete-event (DE) model of computation as a programming model for distributed real-time embedded systems. The advantage of using DE is that it provides a semantic foundation that is simple, time-aware, deterministic and natural as a specification language for many applications. This programming model is based on a carefully chosen relationship between DE’s model time and real time (physical time). We define here a criterion that preserves conservative execution (thus not requiring back-tracking) while allowing for concurrent and distributed execution. The classic Chandy and Misra technique is one execution policy that satisfies the criterion, but the criterion explicitly allows many other alternatives. We discuss alternatives that offer more concurrency than Chandy and Misra and that exploit time synchronization to eliminate the need for null messages.

I. INTRODUCTION

Current programming practices for distributed real-time embedded systems often employ commercial-off-the-shelf real-time operating systems (RTOS) and real-time object request brokers as utilities for implementing the system. Programmers also use languages such as C with concurrency expressed by threads. RTOSs and threads however, provide only weak guarantees that the system will meet real-time constraints. They also do not guarantee that the behavior of the system is deterministic. A consequence is that the only way to achieve confidence in the implementation is through extensive testing. This validates that the functionality and real-time requirements of the system are met for the tested scenarios. However this technique is inherently flawed, because no assurance can be given about the behavior of the entire system. We identify the source of the problem for such techniques as the lack of a timed semantic foundation combined with the inherent nondeterminism in threads [1].

These problems can be addressed by using a distributed discrete-event (DE) model of computation (MoC) [2]. Though normally used for simulation (of hardware, networks, and systems of systems, for example), by carefully binding real time with model time at sensors, actuators, and network interfaces, DE can be used for distributed embedded systems [3]. The advantage of using DE as a semantic foundation is that it is simple, time-aware, deterministic, and natural as a specification language for many applications.

Distributed DE simulation is an old topic [2]. The focus has been on accelerating simulation by exploiting parallel computing resources. A brute-force technique for distributed DE execution uses a single global event queue that sorts events by time stamp. This technique, however, is only suitable for extremely coarse grained computations, and it provides a vulnerable single point of failure. For these reasons, the community has developed distributed schedulers that can react to time-stamped events concurrently. So-called “conservative” techniques process time-stamped events only when it is known to be safe to do so [4], [5]. It is safe to process a time-stamped event if we can be sure that at no time later in the execution will an event with an earlier time stamp appear that should have been processed first. So-called “optimistic” techniques [6] speculatively process events even when there is no such assurance, and roll back if necessary. For distributed embedded systems, the potential for roll back is limited by actuators (which cannot be rolled back once they have had an effect on the physical world) [7].

Established conservative techniques however, also prove inadequate. In the classic Chandy and Misra technique [4], [5], each compute platform in a distributed simulator sends messages even when there are no events to convey in order to provide lower bounds on the time stamps of future messages. This technique carries an unacceptably high price in our context. In particular, messages need to be frequent enough to prevent violating real-time constraints due to waiting for such messages. Messages that only carry time stamp information and no data are called “null messages.” These messages increase networking overhead and also reduce the available precision of real-time constraints. Moreover, the technique is not robust; failure of single component results in no more such messages, thus blocking progress in other components. Our work is related to several efforts to reduce the number of null messages, such as [8], but makes much heavier use of static analysis.

The key idea of Zhao, Liu and Lee in [3] is to leverage static analysis of DE models to achieve distributed DE scheduling.
that is conservative but does not require null messages. The static analysis enables independent events to be processed out of time stamp order. For events where there are dependencies, the technique goes a step further by requiring clocks on the distributed computational platforms to be synchronized with bounded error. In this case, the mere passage of time obviates the need for null messages.

By extending the work of [3] we are moving toward defining a programming model that 1) builds on top of a strong timed semantic foundation, 2) maximizes concurrency of the implementation, 3) provides deterministic schedulability analysis, and 4) eases specification of real-time constraints. We call the programming model PTIDES (pronounced “tides,” where the “P” is silent, as in “Ptolemy”), an acronym for programming temporally integrated distributed embedded systems. In this work-in-progress paper however, we only elaborate on the carefully chosen relationship between model time and real time, and then present our formulation of a general execution strategy for a PTIDES specification.

II. MODEL TIME AND PHYSICAL TIME

In our DE MoC, actors are concurrent components with input and output ports. The input ports receive time-stamped messages from other actors, and the output ports send time-stamped messages to other actors. Actors react to input messages by “firing,” by which we mean performing a finite computation and possibly sending output messages. An actor may also send a time-stamped message to itself, effectively requesting a future firing.

The “time” in time stamps is model time, not physical time. DE semantics is agnostic about when in physical time time-stamped events are processed. All that matters is that each actor process input events in time-stamp order. That is, if it fires in response to an input event with time stamp \( t \), it should not later fire in response to an input event with time stamp less than \( t \).

The semantics of DE models is studied in [9], [10], [11], [12]. In particular, the structure of model time is important for dealing correctly with simultaneous events and feedback systems. For the purposes of this paper, we only care that there are policies for dealing predictably with multiple events with identical time stamps. To be concrete, we will assume that time stamps are elements of the set \( \mathbb{R}^+ \cup \{\infty\} \). In full generality, however, our techniques work for any set of time stamps that is totally ordered, has a top and a bottom, and has a closed addition operator.

Since we are focused on distributed embedded systems rather than distributed simulation, some of the actors are wrappers for sensors and actuators. Sensors and actuators interact with the physical world, and we can assume that in the physical world, there is also a notion of time. To distinguish it from model time, we refer to it as physical time or real time. Here, we assume a classical Newtonian notion of physical time, and assume that each compute platform in a distributed system maintains a clock that measures the passage of physical time. These clocks are not perfect, so each platform has a distinct local notion of physical time. We assume further that we can find a bound on the discrepancies between clocks on different platforms. That is, at any global instant, any two clocks in the system agree on the notion of physical time up to some bounded error.

Synchronized clocks turn out to be quite practical [13]. We have had available for some time generic clock synchronization protocols like NTP [14]. Recently, however, techniques have been developed that deliver astonishing precision, such as IEEE 1588 [15]. Hardware interfaces for Ethernet have recently become available that advertise a precision of 8ns over a local area network. Such precise clock synchronization offers truly game-changing opportunities for distributed embedded software.

We assume that model time and physical time are disjoint, but that they can be compared. That is, we assume that model time is in fact a representation of physical time, even though time-stamped events can occur at arbitrary physical times. In our DE models, an actor that wraps a sensor, however, cannot produce time-stamped events at arbitrary times. In particular, it will produce a time-stamped output only after physical time (the local notion of physical time) equals or exceeds the value of the time stamp. That is, the time stamp represents the physical time at which the sensor reading is taken, and hence it cannot appear at a physical time earlier than the value of the time stamp.

An actor that wraps an actuator has a complementary constraint. A time-stamped input to such an actor will be interpreted as a command to produce a physical effect at (local) physical time equal to the time stamp. Consequently, the model-time time stamp is a physical-time deadline for delivery of an event to an actuator.

At actors that are neither sensors or actuators, there is no relationship between physical and model time. At these actors, input events must be processed in model-time order, but such processing can occur at any physical time (earlier or later than the time stamp).

III. THE PTIDES EXECUTION STRATEGY

Following [3], we capture the information of minimum model-time delay with relevant dependency [3]. In our formal representation of actor-oriented models, a model consists of a set of actors. Any actor \( \alpha \in A \) has a set of input ports \( I_\alpha \) and a set of output ports \( O_\alpha \). Without loss of generality, we assume \( I_\alpha \) and \( O_\alpha \) to be disjoint. We also assume that any local state maintained by the actor appears at an output port, so we do not need to address it explicitly. We further assume that ports are interconnected by a fixed, static network, where each input port is connected to at most one output port. This will ensure that all data dependencies are relations between ports. The set of all input ports is \( I = \bigcup_{\alpha \in A} I_\alpha \), the set of all output ports is \( O = \bigcup_{\alpha \in A} O_\alpha \), and the set of all ports is \( P = I \cup O \).

The minimum delay (in model time) is defined as function \( \delta : P \times P \to \mathbb{R}^+ \cup \{\infty\} \), where \( \mathbb{R}^+ \) is the set of non-negative real numbers. For \( p_1, p_2 \in P \), \( \delta(p_1, p_2) \) is the minimum
The difference between the model time stamp of any event $e_1$ at $p_1$ and that of any event $e_2$ at $p_2$ that totally or partially depends on $e_1$. Intuitively, this number represents the delay (in model time, not physical time) that it takes for $e_1$ at $p_1$ to influence any event at $p_2$. If no event at $p_2$ depends on the events at $p_1$, then we define $\delta(p_1, p_2) = \infty$.

We assume that for every port $\alpha$, $\delta(p_1, p_2)$ is known for all $p_1 \in I_\alpha$ and $p_2 \in O_\alpha$. This information constitutes an interface definition for the actor [16]. To compose these interfaces, we use a min-plus algebra [17] to compute $\delta$ for any pair of ports based on [3]. The min-plus algebra aggregates these dependencies over multiple paths between ports.

We define a path from port $p_1$ to $p_n$ to be a sequence of ports $[p_1, p_2, \ldots, p_n]$, where for any $j (1 \leq j < n)$, either $p_j$ is directly connected to $p_{j+1}$, or $p_j \in I_\alpha$ and $p_{j+1} \in O_\alpha$ for some actor $\alpha$ and $\delta(p_j, p_{j+1}) < \infty$. A subpath is a sequence of consecutive ports in a path. For any pair of ports $p_1, p_n$, the minimum delay $\delta(p_1, p_n)$ is the minimum of the total delays on all the paths from $p_1$ to $p_n$.

An example of calculating the minimum delay is provided in Figure 1. The input ports are labeled $i_1$ through $i_8$ and the output ports are labeled $o_1$ through $o_6$. The actors are represented by rectangles. A triangle pointing into an actor denotes an input port. ($i_8$ is a multi-port denoted by a hollow triangle, which accepts multiple input connections. It can be represented as multiple separate ports in our formulation.) The minimum delay between $i_1$ and $i_8$, $\delta(i_1, i_8)$, can be computed by $\min\{\delta(i_1, o_1) + \delta(i_5, o_5), \delta(i_1, o_1) + \delta(i_6, o_6)\}$. (Direct connections, such as the one between $o_1$ and $i_5$, do not incur any delay in model time.)

An actor may have an output port at which events never depend on events at some of its input ports. This leads us to partition the set of input ports $I$ into equivalence classes $E = \{E_1, E_2, \ldots, E_k\} \subseteq 2^I$. We first define relation $\sim$ such that for any two ports $i_1$ and $i_2$, $i_1 \sim i_2$ if and only if they are both in $I_\alpha$ for some actor $\alpha$ and there exists an output port $o \in O_\alpha$ such that $\delta(i_1, o) < \infty$ and $\delta(i_2, o) < \infty$. An equivalence class is then a transitive closure of the $\sim$ relation. Intuitively, if $i_1$ and $i_2$ are in an equivalence class, then 1) they belong to the same actor, and 2) the events received at them directly or indirectly influence the output signal of an output port of that actor. This means that these events must be processed in time stamp order. If $i_1$ and $i_2$ are not in the same equivalence class, then the input events at $i_1$ can be processed independently of those at $i_2$, and vice versa.

We now define relevant dependency [3] to function $d : E \times E \rightarrow \mathbb{R}^+ \cup \{\infty\}$. For $E_j, E_k \in E$,

$$d(E_j, E_k) = \min_{i_m \in E_j, i_n \in E_k} \{\delta(i_m, i_n)\}$$

As an example, in Figure 1, $E_1$ through $E_6$ are equivalence classes. The relevant dependency between $E_4$ and $E_6$ is $\min\{\delta(i_4, o_5), \delta(i_5, o_5)\}$. The relevant dependency function is pre-computed in a static analysis before execution. Based on this information, we can execute a DE model according to the PTDIES processing strategy, which we discuss in this section.

A set $C_E \subseteq E$ is called a dependency cut for equivalence class $E$ [7] if it is a minimal set of equivalence classes that satisfies the following condition:

For any input port $i \in E$ and any path $\rho$ to $i$, there exist $E' \in C_E$, input port $i' \in E'$ and a path $\rho'$ from $i'$ to $i$, such that either $\rho$ is a subpath of $\rho'$ or $\rho$ is a subpath of $\rho$.

Intuitively, a dependency cut for $E$ is a “complete” set of equivalence classes on which $E$ depends. Completeness in this case means that for each port in $E$, all ports it depends on will be accounted for in $C_E$, either directly by being included or indirectly by having either upstream or downstream ports included. Again using Figure 1 as an example, the dashed curve depicts one possible dependency cut for $E_6$, namely $C_{E_6} = \{E_1, E_2, E_3, E_4\}$. Note that an equivalence class $E$ can have many distinct dependency cuts. The dependency cut is not unique. Note further that $\{E\}$ is always a (trivial) dependency cut for $E$.

A dependency cut can be used to determine when an actor can fire. Specifically, given a dependency cut $C_E$, the actor $\alpha$ to which the ports in $E$ belong determines whether it can process input events received at the ports in $E$ with model time stamps less than or equal to $t$ using the following strategy [7]:

If for any $E' \in C_E$, $\alpha$ has received all events at the ports in $E'$ that depend on events at the ports in $E'$ with model times smaller than $t - d(E', E)$, then it can fire and process the input event received at a port in $E$ with smallest model time (among all the available events at the ports in $E$) that is less than or equal to $t$.

This principle, of course, can be satisfied by a classical DE scheduler, which uses a global event queue to sort events by time stamp. In this case, the oldest event (with the least time stamp) can always be processed. However, this principle relaxes the policy considerably, clarifying that we only need to know whether an event is “oldest” among the events that

1This assumes, of course, that all actors are causal, so events that are produced in reaction to processing an event always have a time stamp at least as great as that of the processed event.
can appear in a dependency cut. We do not need to know that it is globally oldest.

The classic distributed DE execution strategy of Chandy and Misra [4], [5] uses multiple event queues, one on each execution platform. The technique is equivalent to defining the dependency cut to include the ports at the boundaries between platforms. It then simply assumes that all events with time stamps up to that of the most recently received event have been seen. This technique requires messages to be received in order to make progress, hence the requirement for null messages.

The technique of Zhao, Liu, and Lee [3] augments the Chandy and Misra model with an assumption that real-time clocks on the distributed platforms are synchronized up to some bounded error. It further imposes relationships between real time and model time at sensors and actuators. It then uses relevant dependency analysis to determine at any given real time that all events at the boundary ports have been seen with time stamps up real time minus a statically calculated offset.

An obvious extension would combine these two techniques. Non-real-time portions of a DE model may use a technique like Chandy and Misra while real-time portions use a technique like Zhao, Liu, and Lee. The above principle allows for freely intermixing these. If the non-real-time portions can be shown to be sufficiently “ahead of time,” then the use of Chandy and Misra would not compromise the ability to meet real-time constraints.

More interestingly, the above principle allows for other choices of dependency cuts. Putting a dependency cut on the boundary between platforms imposes a constraint that either events traversing that boundary have real-time constraints or that null messages are used. The above principle, however, allows choices other than the boundaries.

Another possibility is to offer system designers explicit control over the relationship between model time and real time at the platform boundaries. For example, a NetworkInterface actor might be defined to have input ports like those of an actuator, which impose a real-time constraint on events delivered to those ports. Specifically, we require that events delivered to the network interface with time stamp \( t \) be delivered at physical time less than or equal to \( t \). If we further assume a bounded network delay \( N_{\text{delay}} \) for a message to be sent across the network, then the receiving platform is guaranteed to receive those events at real time no later than \( t + N_{\text{delay}} \). This real time is in terms of the sending platform’s local clock, but using a time synchronization protocol with bounded error, such as IEEE 1588 [15], the receiving platform can decide a lower bound of the time stamps of future input events by merely checking its own local clock. This allows it to independently determine whether it can process events that it has already received. If all network communication links use network interfaces, then scheduling and schedulability analysis becomes separable by platform.

Another possible objective could be to choose dependency cuts to facilitate schedulability analysis. In particular, whether we have worst-case execution time information or not for particular actors could affect the choice of dependency cut, and hence affect how the distributed model is executed.

IV. Conclusion

We have defined a correctness principle for conservative execution of a distributed discrete-event model that is suitable for both classical distributed simulation and for distributed real-time execution. Our correctness principle relies on a choice of dependency cut. The principle can be applied in a variety of ways, obtaining previously given techniques as special cases, but also clarifying that there are many more alternatives. A remaining challenge is to formulate appropriate optimization problems that guide the application of the principle, to solve these optimization problems, and to provide a distributed execution engine that implements them.

REFERENCES

Cooperative Network and Energy Management for Reservation-based Wireless Real-Time Environments

Jun Yi, Christian Poellabauer, Xiaobo Sharon Hu, Dinesh Rajan, Department of Computer Science and Engineering, University of Notre Dame 
{ jyi, cpoellab, shu, dpandiar } @nd.edu

Liqiang Zhang
Department of Computer and Information Sciences, Indiana University, South Bend
{ liqzhang @iusb. edu }

Abstract

Reservation-based bandwidth allocation mechanisms in wireless and mobile environments, such as supported by the IEEE 802.11e standard, promise to offer enhanced support for real-time services and applications (e.g., mobile multimedia). This work is concerned with the scheduling of real-time traffic during the reserved medium access periods such that the applications' real-time communication needs are met. This is particularly challenging in systems where the bandwidth reservations are insufficient to meet all packets' deadlines. Further, this work observes that the increasingly popular energy management technique DVS (Dynamic Voltage Scaling) can further exacerbate this problem by delaying job executions and thereby packet generation (bringing them closer to their deadlines). Finally, wireless bandwidth is often affected by environmental inferences, which will further affect network performance. This paper studies these effects and presents an adaptive and cooperative mechanism to coordinate DVS, real-time packet scheduling, and link-layer adaptation, thereby increasing the number of packets meeting their deadlines, while ensuring that system-wide energy consumption is reduced.

1 Introduction

As the number of hand-held and mobile devices rapidly increases and wireless network hotspots are increasingly deployed, real-time media streaming applications on those devices will become more popular. It is challenging to support this and other real-time applications on wireless devices due to the unpredictability of the wireless medium. However, recent efforts have introduced resource (i.e., bandwidth) reservation mechanisms that can facilitate real-time streaming. For example, the proposed IEEE 802.11e standard [2] provides enhanced real-time and QoS support for real-time applications. This standard specifies a central control authority named the Hybrid Coordination Function (HCF) and offers contention-free medium access in the HCF Controlled Channel Access (HCCA) mechanism. In HCCA, the HCF (which typically exists at the access point) takes control of the channel and allocates transmission opportunities to each of the nodes in the network [2]. This is achieved by polling each node in a pre-determined order (e.g., round-robin) where each polling frame specifies the start and maximum duration of the channel access period, termed Service Period (SP), allocated to a node. On reception of a polling frame, a node transmits its packets to HCF within the provided SP. At the end of a node’s SP, the HCF polls the next node in its schedule and this process is continued for the remainder of the HCCA phase. The period of recurrence of the service periods at each node is referred to as the Service Interval (SI).

While there have been numerous efforts on packet scheduling, including for real-time traffic, there is a dearth of research on packet scheduling in reservation-based systems. The challenge here is to allocate real-time packets to the available SP intervals such that all packets (in over-provisioned systems) or as many as possible (in under-provisioned systems) meet their deadlines.

This challenge is further exacerbated by the increasing use of energy management technique. Most notably, Dynamic Voltage Scaling (DVS) [4] has received wide attention and can be found in numerous wireless and mobile devices. However, as we will discuss below, the delay in job execution, and consequently in packet generation, can further complicate the real-time packet scheduling problem.

Finally, link adaptation [5] to dynamically vary the data transmission rate has been recognized as an effective way to improve the throughput performance of IEEE 802.11 and other wireless local-area networks (WLANs). There are a number of mechanisms to ensure proper adaptation of the transmission rate (e.g., adaptive rate selection among 1/5/5/2/1Mbps for 801.11b) in response to environmental interferences. The actual transmission time of packets may therefore vary and thus the effective allocated bandwidth of the device may vary as well.

In this paper, we consider a mobile device executing a set of periodic real-time tasks that generate real-time traffic in this 802.11e network. We assume that the device has already been allocated a pair of SP and SI values through a resource reservation mechanism by the access point. The goal
is to transmit real-time packets in those SP intervals before their deadlines expire. Our proposed solution closely integrates existing DVS mechanisms on a wireless device with a novel packet scheduler and the wireless link layer. For example, decreasing the operating frequency of the CPU by the DVS algorithm will affect the timeliness of real-time packets. Increasing the operating frequency, on the other hand, leads to increased energy consumptions. Finally, the transmission rate and packet sizes (even of the same task) may vary and affect the effective bandwidth allocated to this device. This requires a packet management mechanism that coordinates task executions and packet transmissions to improve the timeliness of the packets and to maintain large system-wide energy savings.

2 Observations

In this section, we discuss our observations on the effects of real-time packet scheduling and the use of DVS. We use the following notations: $J_{i,j}$ represents the $j^{th}$ job of the $i^{th}$ task; $P_{i,j}$ represents the packet generated by job $J_{i,j}$; $AS_{i,j}$ and $WS_{i,j}$ represent the actual and worst-case size of packet $P_{i,j}$; $G_{i,j}$ represents the packet generation time of $P_{i,j}$ (i.e., the time a job submits a packet to the packet queue); $D_{i,j}$ represents the transmission deadline of packet $P_{i,j}$; and $d_{i,j}$ represents the deadline of job $J_{i,j}$. We further assume that a job can generate a packet at any time during job execution. DVS mechanisms have been used in

![Figure 1. Effects of a DVS mechanism on real-time packet scheduling: (a) job schedule and clock frequency, (b) packet sizes (expressed as transmission times), (c) EDF scheduling for real-time packets.](image)

the past to conserve energy at the processor level, including DVS approaches that ensure that the deadline requirements of real-time tasks are met [4]. Our observation, however, is that a DVS mechanism not only delays job execution, but also packet generation, thereby potentially causing some real-time packets to miss their transmission deadlines. This problem is exacerbated in reservation-based systems, where packet schedulers have only limited transmission opportunities during the SP intervals. That is, even slight delays in packet generation may push a packet out of its intended SP interval and prevent it from being transmitted before its deadline (if the next SP interval does not begin until the packet’s deadline). As a consequence, packets will either be transmitted late or dropped altogether, e.g., Figure 1 illustrates a case where packets $P_{3,2}$ and $P_{3,3}$ miss their deadlines.

On the other hand, if we modify job deadlines such that packets can easily fit into their intended SP intervals (as illustrated in the example in Figure 2), the clock frequencies for job execution are increased (thereby increasing the energy consumption). Further, since the actual packet sizes may be less than their worst-case sizes, idle intervals within an SP interval may arise, i.e., durations where no transmission takes place, while energy at the wireless device is still consumed.

![Figure 2. Effects of network scheduling mechanism: (a) job deadlines are modified to speed up job execution and packet generation, (b) actual and worst-case packet sizes (expressed as transmission times), (c) EDF scheduling of packets in the actual case.](image)

Our final observation is that the network transmission rate also affects energy efficiency and real-time performance. Network transmission rate is in turn affected by environmental interferences [5]. With low transmission rates, it is better for the task management service to prolong the deadlines of jobs whose deadlines cannot be met (as illustrated in (c) of Figure 2), or to suspend some less critical real-time jobs. With high transmission rates, more real-time workload is allowed leading to less idle time in SP intervals.

3 System Model

Based on the our observations, we present an adaptive and cooperative model that integrates a processor-level DVS mechanism with a novel packet scheduler for reservation-based networks and the wireless link layer (WLL). When a new real-time job is entered into a run-queue (and it is expected that this job will generate a packet with real-time requirements), the DVS mechanism provides the corresponding packet parameters (Table 1) to the packet scheduler. Note that besides this new functionality, we do not make any further assumptions about the DVS algorithm. The earliest ready time is the worst-case packet generation time at the earliest possible job completion time (i.e., the packet is generated at the end of job execution at the earliest possible job completion). Note that the packet may be generated even earlier than its earliest ready time. One possible method to compute the earliest possible ready times is to run all jobs at the highest frequency using any desired task scheduling algorithm and then compute their completion
times as earliest completion times. We further distinguish

between two kinds of packets: packets which have not been
generated yet are called Type-2 packets and already gener-
ated packets are called Type-1 packets. When a job is re-
leased (i.e., entered into a run-queue), it informs the packet
scheduler of the Type-2 packet it will generate. When a
packet is generated, it becomes a Type-1 packet and its ac-
tual size is recorded. The goal of the packet scheduler is to
allocate all Type-1 and Type-2 packets into the available SP
intervals and to provide this resulting transmission schedule
to the WLL. Further, to ensure that all Type-2 packets will
fit into their assigned SP intervals (remember that Type-2
packets have not been generated yet), the packet scheduler
can inform DVS of a modified (i.e., earlier) job deadline,
ensuring that DVS will run the job sufficiently fast such that
the job’s packet will be generated in time. DVS adjusts its
frequency schedule according to this feedback information
(where existing schedulability tests can ensure that the new
frequency schedule will not violate any job deadlines).

<table>
<thead>
<tr>
<th>Name</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual size</td>
<td>AS_{i,j}</td>
</tr>
<tr>
<td>Worst-case size</td>
<td>WS_{i,j}</td>
</tr>
<tr>
<td>Type</td>
<td>TP_{i,j}</td>
</tr>
<tr>
<td>Deadline</td>
<td>D_{i,j}</td>
</tr>
<tr>
<td>Weight</td>
<td>W_{i,j}</td>
</tr>
<tr>
<td>Earliest ready time</td>
<td>E_{i,j}</td>
</tr>
</tbody>
</table>

Finally, WLL uses a link adaptation mechanism to adapt
the transmission rate to environmental interferences and
feeds rate changes back to the packet scheduler. The packer
scheduler, in response, can re-order packets in the transmis-
sion schedule. As a consequence, if the packet scheduler
decides to push a Type-2 packet to a later SP interval, it
may relax the corresponding job deadline again. Note that
schedule changes triggered by WLL feedback cannot result
in earlier job deadlines, i.e., only when a job is entered into
a run-queue, the algorithm can set an earlier job deadline.
This ensures that a job’s deadline is not pushed earlier when
a job is currently executing. Also, DVS can only move ear-
liest ready times earlier to ensure all packets will be gen-
erated before their transmission times. Figure 3 outlines

the proposed integrative DVS and packet scheduling mech-
anism.

4 Cooperative Energy and Real-Time Man-
agement

The primary goal of the integrative approach is to ensure
that as many packets as possible can be transmitted during
the limited transmission intervals. Each packet can have a
weight associated, e.g., the weight could be the size of the
packet or some user-specified urgency parameter. In this
case, the goal is to increase the weighted sum of packets that
meet their deadlines (note that finding an optimal solution
to this problem is NP-hard, we therefore focus on heuristic
solutions). The secondary goal is to increase the system-
wide energy conservation, i.e., the combined energy saved
at both the processor level and the network level. In over-
provisioned systems (i.e., the SP periods offer more trans-
mission opportunities than necessary to meet all deadlines),
the main objective is to increase the energy savings as long
as all deadlines are met. In under-provisioned systems, it
may be impossible to meet all packets’ deadlines and the
objective is to increase the weighted sum of packets that
meet their deadlines (while energy conservation is a sec-
ondary concern).

The packet scheduler’s goal is to allocate packets to the
available SP intervals and to adjust job deadlines. Packet
allocation and deadline adjustment is triggered by these
events:

- A new real-time job enters its run-queue, which results
  in a notification to the scheduler that a new Type-2
  packet is available;
- a Type-2 packet becomes a Type-1 packet, where the
  actual packet size is less than its worst-case size,
  thereby opening up space in the SP interval for other
  packets;
- and the transmission rate is increased or decreased,
  which means that more or less packets can be trans-
  mitted and that the DVS mechanism may be allowed
  to adjust the frequency schedule.

For dynamic workloads of packets, there is no global de-
terministic and optimal packet scheduling algorithm when
we assume that we have no knowledge about these packets
(e.g., actual sizes and generation times). For a static work-
load, this problem can be reduced from the bin-packing
problem and thus is NP-hard [1] (e.g., SPs are treated as
bins and packets are treated as items of various sizes and
weights).

Due to the large overheads [3] composed of medium
access control (MAC) header, PHY preamble/header, ac-
knowledgement (ACK) transmission, and some inter-frame
spaces (IFSs), especially preamble and header are always
transmitted at a much lower rate relative to the payload
transmission rate. The size of packets will make a small
difference on transmission times. As a result, we can rea-
sonably assume that the transmission times of all packets

Figure 3. Adaptive and Cooperative Model

Table 1. Packet Parameters
vary in a narrow range, although their sizes vary in a wider range. If a packet of higher weight competes for the same time slot with a packet of smaller weight, allocating the slot to the packet of higher weight will almost surely result in higher total weight. To increase the weighted sum of packets being transmitted on time, packets of higher weight or with closer deadlines should have higher transmission priority. From this reasoning, we construct a heuristic algorithm for packet allocation and deadline adjustment, consisting of the following major steps:

- **Step 1: Initialization.** The transmission rate is updated. The release time \( R_{i,j} \) of packet \( P_{i,j} \) is its earliest ready time (i.e., \( R_{i,j} = E_{i,j} \)) if \( TP_{i,j} = 2 \) and otherwise the current time (i.e., \( R_{i,j} = currTime \)). The transmission duration \( Trans_{i,j} \) of Type-1 packet \( P_{i,j} \) is calculated based on actual size of its physical encapsulation, current transmission rate, and protocol overhead time. The transmission duration \( Trans_{i,j} \) of Type-2 packet \( P_{i,j} \) is calculated based on worst-case size instead.

- **Step 2: Compute a weighted EDF-based schedule \( Tbl \).** The algorithm starts at the current time (\( currPoint = currTime \)) and scans all packets in increasing order of their deadlines. If the packet \( P_{i,j} \) with the earliest deadline cannot meet its deadline (i.e., \( currPoint + Trans_{i,j} > D_{i,j} \)), the algorithm scans all packets \( \{P_{n,m} | [R_{n,m}, D_{n,m}] \cap Tbl[R_{i,j}, D_{i,j}] \neq \emptyset \} \) released from the start time to the deadline of the packet. Discards the packet \( (ligtestPacket = \text{argmin}[W_{i,j}|P_{i,j} \in \text{PrePackets}]) \) with the lowest weight, and then re-pack the schedule \( Tbl \) from the starting point (i.e., \( startPoint(ligtestPacket, Tbl) \)) of \( ligtestPacket \) packet until \( P_{i,j} \) can fit in or is discarded, whichever comes first. If \( P_{i,j} \) fits in \( Tbl \), the algorithm moves the current scheduling point further (i.e., \( startPoint = currPoint + Trans_{i,j} \)). This process repeats until the packet with the latest deadline is processed. This step's goal is to increase the weighted sum of packets that the SP intervals can accommodate.

- **Step 3: Slack exploitation.** This step scans all Type-2 packets \( P_{i,j} \) in \( Tbl \) in decreasing order of their deadlines, and schedules each Type-2 packet as late as possible. If the successor \( \text{succ} \) of a Type-2 packet in \( Tbl \) is a Type-1 packet, the algorithm first attempts to exchange the scheduling order of \( \text{succ} \) and \( P_{i,j} \) (i.e., \( \text{succ} \leftarrow P_{i,j} \)), as long as the deadline of \( P_{i,j} \) is still met. If this fails, the algorithm attempts to move \( P_{i,j} \) and its subsequent packets as late as possible \( (P_{i,j} \rightarrow, \text{succ} \rightarrow, \cdots) \), as long as the deadlines of those packets are all able to be met. This continues until the \( P_{i,j} \) is unable to be moved further. If \( \text{succ} \) is a Type-2 packet, the algorithm moves \( P_{i,j} \) later, to at most the starting point of \( \text{succ} \) or its own deadline (i.e., \( \min(D_{i,j}, startPoint(\text{succ}, Tbl)) \)). The above process repeats until the Type-2 packet with the earliest deadline is processed. The goal of this step is to delay required job deadlines as late as possible under the condition that the weighted sum of packets being transmitted on time are maintained.

- **Step 4: Modify job deadlines and update the packet schedule.** Here, the packet scheduler computes for each Type-2 packet a new job deadline, which is the beginning time of the SP interval a Type-2 packet occupies. The packet scheduler informs the DVS mechanism of these new deadlines. Further, the packet scheduler replaces the previous packet schedule with this resulting schedule \( Tbl \) and passes the schedule to WLL.

As a natural consequence of EDF, the resulting schedule is optimal if SPs are not overloaded. The wireless link layer always selects the earliest packet from the current schedule as the next packet. When the next packet is of Type-2 and its earliest ready time is at least \( n \) milliseconds away (where \( n \) is a platform specific parameter), the network card can switch to a power saving modes if available.

## 5 Conclusions and Future Work

This paper investigates the conflicts between energy savings and real-time requirements of mobile devices in reservation-based wireless environments. We present our initial work on a collaborative approach to integrate processor-level energy management with network scheduling to ensure that as many packets as possible meet their deadlines, while energy consumption is kept low. The work in this paper can also be applied to generic bandwidth allocation situations in both wired or wireless environments. Our future work will extend the described approach by evaluating energy and real-time performance of our models and algorithms using experiments and simulations and investigating the effects of dynamically changing job deadlines on task scheduling and DVS.

## References

Maximizing Job Benefits on Multiprocessor Systems Using a Greedy Algorithm

Behnaz Sanati and Albert Mo Kim Cheng
Real-Time Systems Laboratory, Department of Computer Science
University of Houston, Texas, USA

Abstract
This project considers a benefit model for on-line preemptive multiprocessor scheduling. In this model, each job arrives with its own benefit function and execution time. The flow time of a job is the time between its arrival and its completion. The benefit function determines the benefit gained for any given flow time. The goal is to maximize the total benefit gained only by the jobs that meet their deadlines. In order to achieve this goal, a variety of approximation algorithms and their applications in multiprocessor scheduling were studied. A greedy algorithm with 2-approximation ratio is proposed to be added to an existing benefit based scheduling algorithm, in order to reduce the delay of each job, by assigning it to the processor with least utilization so far. This method will decrease the flow time of the jobs, resulting in higher benefits gained by each job. Also, evaluation of this approach shows that it uses the CPU cycles more efficiently by providing more balanced distribution of the jobs between the processors. Therefore, more jobs can meet their deadlines and add their gained benefits to the total benefit. In addition, the proposed method is computationally less expensive than the existing benefit based method.*

1. Introduction
Multiprocessor platforms are widely adopted for many different applications in embedded systems and server systems. They are becoming even more popular since many chip makers including Intel and AMD are releasing multi-core chips. Adopting multiprocessor platforms can enhance the system performance, but scheduling jobs optimally on a multiprocessor system is an NP-hard problem.

There are two major models for this scheduling problem. The first is the cost model and its goal is to minimize the total flow time. The second model is the benefit model which aims to maximize the benefit of jobs that meet their deadlines. This research focuses mostly on the benefit model, but also uses greedy approximation algorithm to reduce the flow time.

In the following two subsections, approximation algorithms in general and greedy algorithms in more detail are discussed as an approximate solution to the multiprocessor job scheduling. Subsection 1.3 provides an overview of the previous work on maximizing benefit on-line for multiprocessors. Section 2 will introduce a new approach using a greedy algorithm with 2-approximation ratio, in addition to the previous benefit based algorithm. It also includes the complexity analysis of the new method and an example to illustrate its differences from the previous method. The last section concludes the results of this project.

1.1 Approximation Algorithms
Approximation algorithms are often used to attack difficult optimization problems, such as job scheduling on multiprocessor systems which is an NP-hard problem. An approximation algorithm settles for non-optimal solutions found in polynomial time, when it is very unlikely to find an efficient, polynomial time, exact algorithm to solve NP-hard problems, or the sizes of the data sets are so large that make the polynomial exact algorithms too expensive.

The performance of the approximation algorithms are measured by comparing them with the optimum solution. A \( \rho \)-approximation algorithm defines that approximation ‘a’ won’t be more (or less, depending on situation) than a factor \( \rho \) times the optimum solution \( S \). \( \rho \) is the relative performance guarantee.

\[
S \leq a \leq \rho S, \quad \text{if} \quad \rho > 1 \\
\rho S \leq a \leq S, \quad \text{if} \quad \rho < 1
\]

The next subsection will explain the greedy algorithm which is used in this project and shown to be a 2-approximation ratio algorithm in [1]. A greedy algorithm is also used by Chen et al [4] to maximize the entire profit of uniprocessor systems under energy and timing constraints.

* This work is supported in part by the National Science Foundation under Award No. 0720856 and GEAR Grant No. I092831-38963.
1.2 Greedy Algorithms

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, in the hope that the outcome will lead to a desired outcome for the global problem. In some cases such a strategy is guaranteed to offer optimal solutions, and in some other cases it may provide a compromise that produces acceptable approximations.

Typically, greedy algorithms employ strategies that are simple to implement and require a minimal amount of resources. Greedy approaches can be applied to a wide variety of applications such as map coloring, vertex covering, voting districts, Egyptian Fractions, Dijkstra’s Single-Source Shortest Paths Algorithm, Kruskal’s Minimal Spanning Tree Algorithm and also 0/1 Knapsack problem. The next section explains the definition of the 0/1 knapsack problem which has a guaranteed approximate solution using a greedy algorithm. The multiprocessor scheduling problem can be considered a knapsack problem and a greedy algorithm therefore could be adopted to solve it.

Knapsack

The knapsack problem is defined as follows: Given a set of N items \((v_i, w_i)\), and a container of capacity \(C\), find a subset of the items that maximizes the value \(v_i\) while satisfying the weight constraints \(w_i \leq C\). This problem is an NP-hard problem, requiring an exhaustive search over the \(2^N\) possible combinations of items, for determining an exact solution. A greedy algorithm may consider the items in order of decreasing value-per-unit weight \(v_i/w_i\). Such an approach guarantees a solution with a value no worse than 1/2 the optimal solution.

1.3 Maximizing Job Benefits On-Line

Previous Work

Awerbuch et al presented a constant competitive ratio algorithm for a benefit model of on-line preemptive scheduling [3]. This method can be used on both uniprocessor and multiprocessor systems. In a multiprocessor system, each processor has a stack and a garbage collection, and there is a pool shared by all the processors.

Each job \(j\) arrives with its own execution time \((w_j)\) and benefit density function \(B_j(t)\) for \((t \geq w_j)\). The benefit gained for any given flow time \(f_j\) is \(w_j B_j(f_j)\).

The flow time of a job is the time that passes from its release time \((r_j)\), to its completion time \((c_j)\) and is defined as \(f_j = c_j - r_j\) and is at least equal to \(w_j\) (execution time).

A desired property of the system is the possibility to delay jobs without drastically reducing overall system performance. Also, this algorithm does not use migration on the multiprocessor system.

The job on the top of the stack is the job that is running and all other jobs in the stack are preempted. The time that job \(j\) is pushed onto the stack is denoted by \(s_j\) and the breakpoint is defined as \(s_j + 2w_j\). The priority of each job in the pool at time \(t\) is denoted by \(d_j(t)\) and for \(t \leq s_j\) is \(B_j(t + w_j - r_j)\). For \(t > s_j\) it is \(d'_k = B_j(s_j + w_j - r_j)\). The notation \(d'_k\) is used for the priority of the running job \(k\) on the top of the stack.

Once a new job \(j\) is released, if there is a machine such that \(d_j(t) > 4d'_k\) or stack is empty, then the newly released job is pushed onto the stack and starts running, otherwise it will be added to the pool.

When a currently running job on a machine completes or reaches its breakpoint, it is popped from the stack. If the job has reached its breakpoint before completion, it will not add any benefit to the system.

![Figure 1: Three job storage locations for each machine (pool, stack, garbage collection)](image-url)
and is inserted to the garbage collection. Then, the processor runs the next job on its stack if \( d_j(t) \leq 4d_k \) for all \( j \) in pool, otherwise, it gets the job with max \( d_j(t) \) from pool, puts it into the stack and runs it.

### 2. A New Approach

The above algorithm only focuses on maximizing the total benefit without being concerned about minimizing the flow time of each job. In the meanwhile, the benefit gained by each job that completes before its break point is \( w_j B_j(t_j) \). Since the benefit density function is a non-increasing, non-negative function of time, by definition [3], the more the flow time, the less the benefit gained. Therefore, this paper proposes a new method in order to reduce the flow times by distributing jobs between processors in a more balanced way.

This approach is possible if each processor has its own pool instead of sharing a pool with other processors (see Figure 1). Also, a greedy 2-approximation algorithm similar to the one used in [2] will be deployed as explained in the next section.

![Figure 2: Software Architecture of the System](image)

**2.1 The Algorithm**

A greedy algorithm will add a newly released job to the pool of a machine with the least work load (where sum of \( w_j \)'s of the jobs in its pool and on its stack is the minimum).

The greedy algorithm is as follows:

- When a new job \( j \) is released, if it can not be executed immediately and has to wait in a pool, it will be assigned to the processor that has the least work load so far.
- If \( P_m \) is a set of jobs in the pool of processor \( m \), and \( U_m \) is the utilization of processor \( m \) (total execution time of the jobs in its pool and on its stack), then:
  
  1. Find the smallest \( U_m \) among \( m \) processors
  2. \( P_m := P_m \cup \{ j \} \) and \( U_m := U_m + w_j \)

If the priority of the new job is so high that it can start its execution immediately and also it has more than one option, e.g. processors, it will be pushed to the stack whose processor has less work load (including the new one). This rule will also cover the case that more than one job arrives at the same time and with high priority enough to be executed immediately.

Figure 2 shows the software architecture of the system.

### 2.2 The Computational Complexity Analysis

In the original method, at each time step, the priority of all jobs in the shared pool must be compared with the priority of the running jobs on the top of all processor stacks. If there are \( m \) processors in the system and \( X \) waiting jobs in the pool, \( X \) times \( m \) comparisons are done at each time step to determine if any of the waiting jobs can be pushed onto any stack and start running.

On the other hand, the greedy method will perform \((m - 1)\) comparisons at each job arrival to find the least utilized processor and adds the execution time of new job \( j \) to its utilization for future comparisons, resulting in \( m \) operations at each job arrival.

Then, at each time step, if \( x_1 \) is the number of waiting jobs in first pool, \( x_2 \) in the second pool, and so on so forth, then \( X \) is the total number of waiting jobs \((X = x_1 + x_2 + \ldots + x_m)\).

Since the greedy method only compares the priorities of waiting jobs in each pool with the
priority of the running job on the corresponding stack, only X comparisons are done at each time step. It is now clear that the greedy method is computationally less expensive than the original one. In only one condition it can have the same number of comparisons and that is when there are m new job arrivals at each time step.

2.3 An Example

The following examples are provided to illustrate the differences between the two methods:

Consider a system with three processors, when five jobs are arriving with \( r_j = (1,1,1,1,3) \) and \( w_j = (3,10,4,5,2) \), and are scheduled using both the original and the greedy methods. The total benefit gained by the original method was 2.11. However, the total benefit was improved by about 6.6% resulting in 2.25 by the greedy method.

If the number of jobs is much higher than the number of processors, the original method is more likely to miss some deadlines than the greedy method. In the above example no job was missed. However, a job that misses its deadline will not provide any benefit. In that case the greedy method will show better improvement in the total results.

The algorithms were tested for a 2-processor system and five jobs with \( r_j = (0,0,1,1,1) \) and \( w_j = (10,15,4,3,1) \). The benefit gained by the previous algorithm was even slightly better, but after adding two more jobs to the task set with \( r_j = (1,2) \) and \( W_j = (2,5) \), the results were almost the same (2.25 vs. 2.23). Then the test was repeated with nine jobs, first seven jobs exactly the same as the former case and jobs 8 and 9 with arrival time 15 and 16, and execution time (\( W_j \)) of 3 and 5, respectively. This time, the results were 2.9 vs. 3.11. Our algorithm could improve the benefit by 7.2% approximately. As expected a task set with heavier load could be handled better with the greedy algorithm.

3. Conclusion

The previous work [3] was only a benefit model to maximize the benefit gained. This research project uses a greedy 2-approximation algorithm to assign a newly released job to the machine with the minimum work load (total \( w_j \)).

The greedy method is computationally less expensive than the original one. In only one condition in our experiments, we have the same number of comparisons and that is when there are m new job arrivals at each time step (when there are m processors in the system).

Also, it is shown that the greedy method has improved the performance of the original benefit based method specially in the cases with heavier work load, by assigning each newly arrived job to the machine with less utilization resulting in fewer missed deadlines and shorter flow times which will increase the total benefits. The greedy method distributes the work load between the processors in a more balanced way, so that there will be less waste of CPU cycles and even in those cases that the previous method could gain more benefit, it took longer to finish the whole task set.

This means that the whole task set can be executed faster using the greedy method. Therefore, the method can be considered as a combination of the cost model and the benefit model, which are explained in the first section of this paper. In other words, the greedy algorithm can be applied to more variant types of applications, either those which need a more cost effective scheduling method or a benefit based method.

In the ongoing work, the performance analysis is being done. More research and a thorough analysis of these algorithms using more test cases can result in better understanding of how much this new greedy algorithm can improve the existing benefit based algorithm.

References

Timing Analysis of the Priority based FRP System

Chaitanya Belwal  
cbelwal@cs.uh.edu  
Dept. of Computer Science, University of Houston, TX

Albert M. K. Cheng  
cheng@cs.uh.edu  
Dept. of Computer Science, University of Houston, TX

Walid Taha  
taha@rice.edu  
Dept. of Computer Science, Rice University, Houston, TX

Angela Zhu  
angela.zhu@cs.rice.edu  
Dept. of Computer Science, Rice University, Houston, TX

Abstract

Kaiabachev, Taha, Zhu [1] have presented a declarative programming paradigm called Functional Reactive Programming, which is based on behaviors and events. An improved system called P-FRP uses fixed priority scheduling for tasks. The system allows for the currently executing lower priority tasks to be rolled back to restoring the original state and allowing a higher priority task to run. These aborted tasks will restart again when no tasks of higher priority are in the queue. Since P-FRP has many applications in the real time domain it is critical to understand the time bound in which the tasks which have been aborted are guaranteed to run, and if the task set is schedulable. In this paper we provide an analysis of the unique execution paradigm of the P-FRP system and study the timing bounds using different constraint variables.

1. Introduction

Reactive Programming has been found to be ideal in the area of real time systems. Most real time systems are reactive where the host raises events which are acted upon in a certain time frame. Functional programming is a paradigm based on lambda – calculus and offers various advantages over non-Neumann style of programming that is prevalent in standard languages. In [4] and [5] Functional Reactive Programming has been implemented for Real Time applications. Wan, Taha, Hudak [2] have given a statically-typed language called RT-FRP for real time systems which considers and space and time cost of execution. In [3] a compilation strategy to convert RT-FRP semantics into efficient code is given. The code of this new system called E-FRP has been tested on a small microcontroller driven robot. All events in E-FRP are assumed to have the same priority. Events go into the queue and are executed in order, and the next event can execute only when the one before has completed execution. System

Interrupts with critical deadlines will have to wait for the execution queue to complete before it can start. This will cause the interrupts to miss its deadline leading to potentially catastrophic results. To overcome come this, a priority based FRP (P-FRP) system has been developed. This system used fixed priority scheduling to assign a priority number to every task before execution. If a task is executing and a higher priority enters the queue then the currently executing task is stopped and using a rollback mechanism the task is aborted and system state is restored. This prevents any side effect from the execution of the lower priority task. The higher priority task then starts execution. Though it may seem that the lower priority task has been ‘preempted’, when it starts execution it will have to restart. Hence from an execution standpoint the task can be considered non preempt-able, even though significant CPU resources might have gone into executing and then rolling it back. The system also needs to account for asynchronous and aperiodic tasks. These combined with the semantics of rollbacks offer significant challenges in the study of bounds of various task execution parameters. By constraining other variables we can assume that the entire task set is non – pre emptive. However this will give an inaccurate picture of the actual resources used by the system since even though the task has rolled back and has not executed it has still consumed CPU resources. The actual resource bound will not be the same as when the tasks are considered simply non preempt-able. For example if the FRP system runs on a power aware real time host the actual power consumed will be much more than if the tasks are considered to be simply non-preempt-able not have executed. Rollbacks take significant CPU (and disk) resources, and hence should be considered in the timing analysis.

2. E-FRP

The original semantics of E-FRP follow no priority or deadline scheduling. This scheme can be compared to First in First out (FIFO) scheme where tasks that come in first are executed. New tasks are put in queue and wait while other tasks ahead in are completed. As shown in [14] FIFO gives an infeasible schedule when deadlines and priorities are given. It is easy to put a general upper bound on the wait time of the task. Once a task is put in the queue it has to wait for the all the previous tasks to
finish. If there are \( n \) tasks and \( t_i \) is execution time for task \( i \), then the maximum possible wait for task \( k \) is when it is placed last in the queue. In this case the wait time will be sum of execution times of all tasks before \( k \). Therefore maximum wait time = \( \sum_{i=1}^{n} t_i - t_k \).

### 3. Priority based FRP

In P-FRP a fixed priority is assigned to every task before compile time. Each event in the system is mapped to its fixed priority, numbers for which are selected from a fixed range of integer values. All events are executed atomically since task preemption is a rollback action. This way P-FRP retains the execution semantics of P-FRP. A bound on the waiting time for low priority tasks has been analyzed as follows.

There are \( n \) events, event \( i \) is represented by \( I_i \) each having an arrival rate of \( r_i \), which is the number of occurrences of the event per second. Task \( I_i \) has a priority of \( i \). The maximum wait for an event \( k \) has been deduced to be \( (n - k) G_k \), where

\[
G_k = \frac{1}{\max(r_{k+1}, r_{k+2}, \ldots, r_n)} \cdot (n - k) \cdot \min(r_{k+1}, \ldots, r_n)
\]

Tasks \( k+1 \), \( k+2 \) ... \( n \) are of higher priority than \( k \).

However this time bound is restricted if certain conditions are true. These are:

1. \( t_k \gg t_{k+1} \)
2. \( G_k \gg t_k \)
3. Same event will not occur if prior occurrence has not handled.

Where \( t_k \) is the execution time of task \( k \). \( G_k \) is the maximum gap guaranteed to exist. Gap is the time period that exists between occurrences of task \( I_j \) and task \( I_m \) where \( j = m \) and \( m, j > k \). Any task whose priority is greater than \( k \), cannot execute in the gap. The gap is available exclusively to run task \( k \).

The first condition says that tasks with lower priority have an extremely low execution time relative to higher priority tasks. This is valid in some execution scenarios, for example a normal operating system where higher priority tasks can be system interrupts and low priority tasks are normal applications. Most interrupt handlers have small and fast executing code whereas application tasks are large in both time and space. Though no deadline is specified this can be compared to a soft real time system, since interrupts have to be handled fast as other application behavior might depend on them. As an example some application which is waiting for a mouse click event will have to idle till the mouse / keyboard interrupt is handled. Clearly the mouse interrupt has a higher priority and also a soft deadline.

However in general for real time systems both hard and soft this assumption can lead to incorrect results. In such systems execution time of tasks is not indirectly (or directly) proportional to their priorities, and no relation can be formed between the two. It is possible for tasks with a large execution time to have a higher priority than tasks with relatively less execution time. Execution time of tasks is an important consideration in analyzing any real time system. Worse Case Execution Time (WCET) of tasks is used to get the upper bound on wait times and is used when any useful scheduling policy for the system has to be defined.

The second assumption says that the maximum gap available to task \( k \) should be larger than the execution time of the task. This is important since if the gap is less than the execution time then the task will never be able to complete within the observed time period. In such a case the task will start execution in an available gap, then a higher priority event will enter the queue forcing the executing task to stop and rollback. The aborted task will restart in the second available gap only to be aborted again. This will be repeated many times though the task will still not complete since it has to start re start execution in any available gap. This means the task set is not schedulable and is therefore not suited for study of time bounds. Schedulability of the task set is an inherent assumption with the second condition.

The third condition again implies schedulability of the task set, In E-FRP the length of the task queue is bounded by \( \sum_{i=1}^{n} t_i \). If an event comes and the queue is full then the event will not be run at all. When the first instance runs the length of the queue becomes \( \sum_{i=1}^{n} t_i - 1 \).

Hence this condition deals with the resource bound ness of the system. Some real time systems can have an event generated before the first one is handled. Hence those systems will not have this time / gap bound, though the queue size can be increased by adding empty task sets. It is clear from the wait time equation that a task of highest priority \( I_0 \) will require no wait since \( (n - k) G_k = 0 \). Further study is required to find out the tightness of this bound. A new method also needs to be derived by relaxing some of the conditions which should be a more practical representation of existing real time systems. Our work aims to derive an upper bound which accounts
for task execution time and where the WCET is related to priorities of a task.

The timing analysis in [1] also does not consider the start time of tasks. Higher priority tasks are sporadic though a minimum period of separation is not specified. They also do not have any explicit deadline. It is assumed that a high priority the task starts execution immediately on entering the queue. When deadline and task execution time is considered the time taken for rollback will also have to be accounted for. If roll back time is too much a higher priority task may miss its deadline. We have to find a relation between size of the task and the time taken to abort it, do get a real picture on the schedulability of the task. We will also try to find out the cost in term of CPU time incurred during rollbacks. The total time can be accounted as context switches time, though in this case it is more prominent and cannot be ignored. An upper bound on context switch will have to be derived while finding the maximum wait time. It will also impact the bound ness of CPU resources, and can be used to find out the power consumed by the system in a more accurate way.

4. Example

Consider the following set of 3 tasks $T_1$, $T_2$, and $T_3$. $t_i$ is the execution speed of task $i$ in seconds, and $r_i$ is the arrival rate (number of occurrences / second)

- $T_1$: $r_1 = 1$, $t_1 = .7$
- $T_2$: $r_2 = 2$, $t_2 = .1$
- $T_3$: $r_3 = 3$, $t_3 = .05$

In E-FRP the maximum wait time for task $T_3$ will be:

$$\sum_{i=1}^{3} t_i - t_2 = 0.75$$

Now we assign a static priority order to this task set. $p_i$ is the priority of task $i$, and $p_3 > p_2 > p_1$. The execution times for this task set satisfy the necessary condition for the gap bound given in [1] to be used. Hence the maximum wait time for $T_2 = (3 - 2) G_2$.

$$G_2 = 1 / \max(r_3 , (3 - 2) \cdot \min(r_3))$$

$$= 1 / \max(3, (3 - 2) \cdot \min(3))$$

$$= 1 / \max(3, 3)$$

$$= 1 / 3$$

$$\therefore$$ Maximum wait time = $1 * 1 / 3 = 0.33$

Hence we can see that with P-FRP, higher priority tasks will have a lesser wait time.

5. Real – time Databases

The P-FRP system has asynchronous release of tasks, the intervals between them are aperiodic and executed tasks can be rolled back without completion. This makes the task set non-preempt-able though it implements preemption semantics. Studying the time bound of such a system is challenging. Research has been done where the task set running on the CPU is non-Preemptive with variable execution time [6], is asynchronous where the start time is unknown[7] and where task set is non preemptive and sporadic [8]. In [9] algorithms have been given to find multiple feasible intervals (gaps) for a non-preempt-able task run. However no study has been done where these variables exist alongside with the consideration of an executing task set aborting and restarting again. We have looked at systems which have real time behavior but support task aborts. The rollback and abort mechanisms are implemented by databases and if we add time constraints the subset is real time databases.

To allow for data consistency every database transaction is atomic with respect to each other. Hence all databases implement a system for concurrency control to guarantee atomicity of the transactions. Concurrency control strategies in databases are generally of two types pessimistic and optimistic. Pessimistic strategies block the execution of a transaction that will lead to data conflicts. An optimistic strategy continues with the operation till the end and then rollback the transaction that will lead to conflicts. In our study we will look at optimistic strategies that have been implemented with timing constraints. This models the priority based FRP closely.

According to Shu [10] abort – oriented protocols were mainly developed to cope up with situations where the blocking property provided by pure locking protocols such a priority ceiling were not capable of scheduling tasks due to excessive blocking. A transaction is aborted if it prevents the completion of other high priority tasks. Though this allows the transaction set to be scheduled, it incurs additional costs in terms of aborting and re-execution. This cost has been studied in the Shu’s work. Aborting a task also leads to priority inversion where a low priority task can run before a higher priority one. Method like the Priority Ceiling Protocol [12] prevents this from occurring. Byun, Burns, Wellings [9] do a response time analysis of hard real time transactions. For concurrency control they use priority abort where a lower priority transaction is aborted to allow transaction of a higher priority to run. However transactions that are waiting for a commit are not aborted to save time. Liang, Kuo and Shu [11] provide a class of abort oriented protocols for real time databases. The motivation for
developing these protocols is to avoid excessive blocking. This paper analyzes which standard scheduling algorithms like Earliest Deadline First (EDF) or Least Laxity First (LLF) can be used with transactions without affecting the validity of the data. Compatibility between the two is important, and this study will be important for P-FRP when new scheduling algorithms like Rate Monotonic, or dynamic Algorithms like EDF / LLF will replace the current priority assignment of tasks. A Basic Aborting Protocol (BAP) and its various derivations have been given. Tasks in BAP are classified as abortable or non-abortable which is determined by an offline schedulability analysis. In our study we have to consider all tasks as abortable because P-FRP does not distinguish tasks which can be aborted or not. Cheng [15] and Cheng, Chang [16] have developed schedulability tests for transactions in real-time systems.

6. Conclusion

We are looking to determine the timing bounds of the priority FRP system which allows for time bound tasks to run in the system and allows task pre-emption by aborting the tasks. The task abortion finds an analogy in databases. Real time databases allow for both task aborting and timing constraints to be present in the system. Hence a study of system in real time database is important to understand the timing requirements of the P-FRP system. We also have to account for asynchronous release of tasks which are aperiodic in nature and study the Worse Case Response Time of the system. The original paper has studied this response time which is subject to lot of constraints. Our task is to come out with an improved timing analysis which closely models real time systems in practice today.

References

Abstract

The Supervisory Control and Data Acquisition System (SCADA) monitor and control real-time systems. SCADA systems are at the backbone of the critical infrastructure, and any compromise in their security can have grave consequences. Therefore, there is a need to have a SCADA testbed for checking vulnerabilities and validating security solutions. In this paper we develop such a SCADA testbed.

1 Introduction

SCADA refers to a large-scale, distributed measurement (and control) system. The supervisory control system is placed on top of a real time control system to control an external process. SCADA systems are used to monitor or to control chemical or transport processes, in municipal water supply systems, to control electric power generation, transmission and distribution, gas and oil pipelines, and other distributed processes.

SCADA systems are comprised of three components:

1) Remote Terminal Units (RTU): connects to the physical equipment and collects the bulk of the data. The RTUs must provide data reliability and data security.

2) Master station and Human Machine Interface (HMI): consists of the servers and software that connect to the field equipment. HMI is responsible for compiling and formatting the collected data so that the human operator can make appropriate supervisory control decisions.

3) Communication infrastructure: used to connect various components of the SCADA system together. This infrastructure consists of, for example, multiplexed fiber-optic, satellite network, and Internet.

More details of these components will be given in Section 2. Given the critical nature of the SCADA systems, ensuring their security is of great importance. Attacks on the SCADA system can have serious consequences, such as endangerment of public health and safety, environmental damage, and significant financial impacts. There is a growing interest that the current SCADA systems are vulnerable to many cyber attacks [14]. Protection of SCADA systems has traditionally been based on the security by the obscurity concept. Proprietary protocols prevent an attacker from breaking into the system due to insufficient knowledge. Today such protection relies mainly on standards, recommendations, policies, and suggestions for possible countermeasures [1]. In order to better understand how to protect SCADA systems, it is imperative to perform vulnerability assessment on these systems and develop appropriate security mechanisms to protect the SCADA systems against attacks. To do so, developing a SCADA system testbed is essential. Recently, a SCADA testbed for the power system has been developed in [18]. Sandia National Laboratories SCADA testbed [4] is an example of a government sponsored testbed. The European community has also started working on creating a SCADA security testbed [5].

In this paper, we describe our SCADA security testbed. The rest of the paper is organized as follows: Section 2 discusses the reference architecture for the SCADA testbed. Section 3 explains the testbed implementation of our system in detail. Section 4 discusses the attack scenarios we plan to perform on the SCADA testbed. Sections 5 and 6 describe the status of the SCADA testbed and the next steps in the process. Section 7 concludes the paper.

2 Reference Architecture

In this section we detail the functional layers of our SCADA testbed architecture and discuss the interactions between them. Figure 1 shows the reference architecture for this testbed.

The corporate network represents the business end of an utility. This network is typical of an enterprise with a LAN/WAN connected to the Internet. However, in the case of utilities and industrial plants, the corporate network is often connected to the SCADA network in order to simplify business processes by allowing network...
access to critical data on SCADA servers. This is one of the biggest information assurance concerns related to SCADA systems as an attacker can now connect to the SCADA network via the Internet by compromising nodes on the corporate network.

The SCADA master station consists of the SCADA master servers and the HMI. The master station is located in a central control center from where operators can monitor the performance of the entire system. SCADA master servers run the server side applications that communicate with the RTUs. The SCADA master servers poll the RTUs for data and send control messages to supervise and control the utility’s physical infrastructure. Backup servers are used to increase fault-tolerance of the system. In order to add resilience, a backup master station may also reside in a physically separate location with independent communications channels to the RTUs. Various backup configurations may be used including hot, warm and cold backups.

Figure 1 also shows the various communication media commonly seen in a SCADA network. Dial-up modem, private leased line, wireless/radio and LAN/WAN links are widely used. From a SCADA system perspective, the primary difference between these links is generally the speed of communication and the noise on the channel. The communication protocols used over these channels vary based on the RTUs. There exist hundreds of different SCADA protocols, many of which are proprietary. However, Modbus (RTU, ASCII or TCP) [16] and DNP3 [7] are by far the most prevalent. Almost all SCADA protocols lack any authentication or confidentiality mechanisms, making these communications channels vulnerable to attacks.

A utility may have anywhere from hundreds to thousands of RTUs controlling its infrastructure. RTUs are generally physically distant from the SCADA control center and can be miles away. In many cases, the RTUs are not physically secured. Most RTUs (especially legacy units) do not have proper information security mechanisms. Passwords are often sent in the clear and there is no way to authenticate the SCADA master server. RTUs have analog and digital I/O that interface with sensors and actuators connected to the infrastructure. This interface can be wired or wireless. Wireless HART [11] is an example of a wireless communications protocol used by RTUs to communicate with the sensors and actuators. The RTUs may be configured in a variety of different network topologies. The link between the SCADA master server and RTUs may be point-to-point or point-to-multipoint. The RTUs may themselves be configured in a cascading topology as well.

The physical infrastructure can represent the power grid, natural gas distribution/transmission system, water distribution system etc. It is the infrastructure being controlled and monitored by the SCADA system. SCADA systems may regulate the pressure of the...
gas/water pipeline or the voltage in the electric power grid. Sensors and actuators connected to the RTUs are placed along various points of the infrastructure in order to effectively perform this task. In many cases, the physical infrastructure has significant redundancy built in to provide increased availability and fault-tolerance for the physical system.

3 Testbed Implementation

We envision (at least) three different realizations of the reference architecture: single simulation-based, federated simulation-based, and emulation- and implementation-based.

The single simulation-based instantiation has all elements implemented using a simulation framework and language, like Simulink/Stateflow from Mathworks [15]. We envision that the individual components of the architecture are implemented as Simulink subsystems that include the plant simulation, sensor simulations, simulations for the data acquisition and control activities on the RTUs, simulation of the computations performed on the SCADA servers, etc. For high-fidelity simulations we will model and simulate the implementation platforms as well: the OS schedulers and the networking mechanisms. The TrueTime toolsuite [23] provides a good example for doing this in the Simulink framework. For some, e.g. network attack scenarios these models will be extended to faithfully simulate the dynamic behavior of the network under attack.

The federated simulation-based instantiation uses several, dedicated, coordinated simulation engines that simulate the various architectural elements. Here, the key is that the individual simulation engines work with high-fidelity, industrial-grade models, possibly using off-the-shelf, commercial products. The same architectural elements are instantiated with a different technology, for example Speedup [2] for plant simulations, Omnet++ [19] for network simulation, and DEVS [24] for simulating software modules, etc. In this case the problem is the timed coordination across these simulation engines, but DoD’s High-Level Architecture (HLA) [13] offers a platform to solve this problem. HLA provides services for simulation time coordination and data interchange during the simulation process, and several simulation engines have HLA interfaces implemented.

The emulation- and implementation-based instantiation uses actual commercial SCADA devices along with implementations of the software modules performing the data processing (running on realistic hardware), simulations of the network (running on a network emulator like EmuLab [9]), and real-time simulations for the plant (running on dedicated, high-performance hardware). We believe such an emulation/implementation-based realization is feasible and could be made highly realistic and scalable. Attacks on the network and computing nodes could be analyzed in a contained laboratory environment, which is safely decoupled from the ‘real network’, yet provides a highly realistic environment (e.g. like DETER [6] testbed).

4 Planned Experiments

SCADA networks are increasingly interconnected with other networks, and ensuring sufficient level of security for these networks is a challenge. An attack on any software component has an inevitable impact on the physical system with potential dire consequences. Therefore, securing both software and the physical system is essential. The security objectives that are of great importance in SCADA systems are integrity and availability. Integrity, in this framework, means that each component of the system functions and interacts with other components in the manner intended. This also includes the integrity of the collected data. The integrity directly maps into the reliability of the system.

In this work, we will implement specific experimental attack scenarios that compromise the integrity and availability of the entire system. Our goal is to develop methods to detect, predict and quantify the impact of these security attacks on the SCADA system.

An exhaustive analysis of all possible attacks is not feasible, but attacks trees are generally used in the literature to categorize different types of attacks [17]. In this work, we focus on specific scenarios and corresponding countermeasures, prioritizing threats that have a stronger impact on the integrity and availability of the entire system. The priority will be determined by the classification of vulnerabilities based on the consequences of the corresponding attack. The specific experiment scenarios that we analyze are:

- Denial of service attacks on sensors: We consider two types of denial of service attacks: jamming, and exploit of communication protocol design flaws. Jamming results in the loss of functionality by the network. TCP vulnerabilities or design flaws may also be leveraged. For example, a sensor node can be flooded with TCP requests which results in power exhaustion.

- Integrity attacks: Sensor outputs are essential to the situation awareness of a system. Consequently, sensors that transmit misleading outputs are a security threat. Our goal is to establish means to detect a sensor that emits corrupted data. In addition, we look at the software integrity of the RTU firmware to combat attacks that modify the behavior of the RTU. We consider software based attestation [20], secure code execution [21] and secure code update schemes for the RTUs [22].
Phishing attacks: These are attacks against a web server that allows the attacker to access to protected information. This attack often is the first stage of a more complex attack [8].

In order to investigate these attacks, we need to provide the necessary modeling foundations on which threats and mitigation methodologies are based. We plan to develop mathematical and computational models for the interaction between the software infrastructure and the physical processes. The data-traffic generated by a SCADA system is complex and heterogeneous; the resources are dynamically distributed so that any analysis scheme has to adapt to continuous changes to the data-traffic patterns. In order to differentiate between normal changes and results of attacks or hardware failure, we plan to use accurate process modeling which is an abstraction of the time-evolution of the SCADA system.

5 Status

Work on the single simulation-based instantiation has started and we have a simulation of the physical infrastructure and its interaction with sensors and actuators. We are also working on a simple version of the emulation- and implementation-based instantiation of the testbed. We will use commercial RTUs and simulate the SCADA master server using commercial and custom applications. Our initial goal is to test and develop mechanisms to ensure the integrity of the RTUs.

6 Next Steps

In the following months we plan to improve upon our single simulation-based instantiation and simulate the SCADA servers, RTUs and sensors as well. We will then test high-level attack scenarios and solutions on this testbed. The results of these tests will be used to generate an attack tree to categorize attack scenarios and countermeasures. We eventually plan to shift our single simulation-based instantiation to a federated simulation-based instantiation of the testbed. This testbed will allow us to test various attack scenarios and solutions in a realistic but simulated environment. We will also continue improving our emulation- and implementation-based instantiation along the way to allow for tests on a more realistic and scalable environment.

7 Conclusion

It is imperative that SCADA systems be secured, given their critical nature. The SCADA testbed will help us design and test solutions to various attacks against SCADA systems. We hope to design retrofit solutions that will help secure existing and legacy SCADA systems as well as cutting-edge solutions that will help protect future SCADA systems for many years to come.

8 Acknowledgements

This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422) and the following organizations: AFOSR (#FA9550-06-1-0244), BT, Cisco, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia, and United Technologies.

References

Partial Program Admission by Path Enumeration

Michael Wilson
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: mlw2@arl.wustl.edu

Ron Cytron
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: cytron@cse.wustl.edu

Jonathan Turner
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, Missouri 63130
Email: jon.turner@arl.wustl.edu

Abstract—Real-time systems on non-preemptive platforms require a means of bounding the execution time of programs for admission purposes. Worst-Case Execution Time (WCET) is most commonly used to bound program execution times. While bounding a program’s WCET statically is possible, computing its true WCET is difficult without significant semantic knowledge. We present an algorithm for partial program admission, suited for non-preemptive platforms, using dynamic programming to perform explicit enumeration of program paths. Paths – possible or not – are bounded by the available execution time and admitted on a path-by-path basis without requiring semantic knowledge of the program beyond its Control Flow Graph (CFG).

I. INTRODUCTION

Admission control in real-time systems running on non-preemptive platforms requires the ability to bound the execution time of applications. In a trusted environment, a single administrator can make an out-of-band determination of execution boundedness. Untrusted, shared environments are more difficult. As an example of such an environment, consider network virtualization, which has been advanced as a way to foster innovation in the Internet [1].

In network virtualization, core router platforms host 3rd-party application code, running at Internet core speeds, allowing the creation of high-speed overlay services [2]. These platforms, of which the IXP 28XX is a representative example, usually have no preemption mechanism suitable for use at high speeds. Internet core speeds necessitate extremely tight cycle budgets for packet processing. To share this type of system among untrusted parties requires stringent admission control.

In other domains, instrumentation with runtime checks to enforce proper behavior is a practical solutions. Unfortunately, Internet core speeds render runtime checks impractical. At 5Gbps, an IXP 2800-based system with 1.4 GHz microengines and 8 hardware thread contexts has a compute budget of 170 cycles. With such tight budgets, even a few runtime checks can quickly push otherwise admissible program paths over budget. A practical solution must therefore impose as little runtime overhead as possible.

Worst-Case Execution Time (WCET) analysis is the currently accepted approach. A WCET bound can be established statically, assuming that all program paths are viable. However, some well behaved programs might be rejected. For example, a program may have mutually exclusive code paths that, taken together, exceed the cycle budget. Demonstrating that these paths are mutually exclusive takes semantic knowledge, either provided by the developer or deduced by analysis at admission time. In most domains, this information is provided by the developer as branch constraints. For our virtualization application, we cannot trust the developer; any semantic knowledge must come from the analysis.

We propose partial program admission as a practical solution to this problem. By explicitly examining all paths, we can perform static analysis to re-write 3rd-party applications to achieve the following goals:

1) all “safe” paths (paths that complete under budget) are admitted,
2) no “unsafe” paths (paths that complete over budget, or that do not complete) are admitted,
3) no runtime penalty is imposed on any safe path, and
4) no semantic knowledge is required.

To re-write the program, we actually duplicate some code paths. While this causes some code expansion, or “bloat”, in practical cases the bloat proves to be within acceptable limits.

II. ALGORITHM OVERVIEW

Our algorithm should be considered in the context of a simplified processor model. Our idealized processor has instructions taking exactly one cycle to complete. All memory accesses complete in one cycle. There is no pipeline.

Our computational model is event-driven, where code is executed only in response to these events. For the network virtualization application, the event is packet arrival.

Finally, we require the developer to add a “time-exceeded” exception handler to her code. The exception handler is required to adhere to strict coding guidelines which make static analysis simple and easy.

A. Path Enumeration

Our input to the algorithm consists of an assembly level representation of the program. From this, we can develop a Control Flow Graph (CFG) of the program, in which edges are labeled by the execution time required for the corresponding program segments. Our objective is to derive a new CFG that executes the same sequence of instructions for program executions that complete within a specified time bound B, while
terminating in an exception handler for program executions that exceed the budget \( B \).

The conceptual starting point for this construction is the creation of a Control Flow Tree (CFT) from the CFG. The CFT duplicates nodes in the CFG as necessary, in order to convert the graph into a tree.

See Figure 1 for an example. Nodes \( S \) and \( T \) are dummy nodes used to delineate entry and exit points, and contain no actual code. Similarly, in the CFT, \( T1 - T4 \) are copies of the dummy node \( T \) and contain no code.

Code generated from the CFT is functionally identical to the original CFG. If the length of the path from the root node to a node \( u \) in the tree exceeds \( B \), then we can replace the subtree rooted at \( u \) with an exception node, representing a jump to the exception handling routine. As an additional step, if after applying this step, the CFT contains a subtree whose leaves are all exception nodes, we can replace the entire subtree with an exception node.

This pruning procedure is illustrated on Figure 1. Let us consider a budget of 10 cycles. While it would be valid to execute the path \( A \to C \to D2 \to F2 \to G4 \) before aborting to the exception handler, it is clear that any execution path reaching \( F2 \) will go over budget. Our earliest chance to raise the exception is by intercepting the branch instruction at \( D2 \), with the result shown in Figure 2.

We refer to the tree constructed in this way as the \( B \)-bounded execution tree of the original control flow graph. We note that such a tree can be defined relative to any node \( u \) in the CFG and we let \( bxt_B(u) \) (or generally, BXT) denote this execution tree.

While one could generate a version of the original program directly from the BXT, this typically results in an excessive amount of code duplication. We can dramatically reduce the amount of code duplication by merging equivalent subtrees of the BXT in a systematic way.

B. Code Duplication Reduction

The BXT typically contains multiple subtrees that are identical to one another and can be merged. To make this precise, we define two nodes \( u_1 \) and \( u_2 \) in the BXT to be equivalent if they were derived from the same node \( u \) in the original CFG (that is, they represent copies of the same original program segment). Two subtrees of the BXT are equivalent if they are structurally identical and all of the corresponding node pairs are equivalent. We can merge any pair of equivalent subtrees without changing the set of executions, yielding a bounded execution graph (BXG) equivalent to the BXT. Conceptually, the merging is performed in a top down fashion. That is, if \( u_1 \) and \( u_2 \) are roots of equivalent subtrees, we merge them so long as there are no ancestors \( v_1 \) of \( u_1 \) and \( v_2 \) of \( u_2 \) that are also roots of equivalent subtree. The merging process continues, as long as there are equivalent subtrees that can be merged.

Returning to our example, nodes \( D1 \) and \( D2 \) cannot be coalesced because their child execution trees are different. \( D1 \) has children \( E1 \) and \( F2 \); \( D2 \) has children \( E2 \) and \( X \). However, the subtrees rooted at \( E1 \) and \( E2 \) are identical. There is no need to retain both trees. Instead, we can coalesce them into a single subtree. Even further, the tree rooted at \( G2 \) is identical to the subtrees rooted at \( G1 \) and \( G3 \). We can also coalesce the \( G2 \) node with the \( G1/G3 \) node from the \( E1/E2 \) execution tree. See Figure 3.

In contrast to the massive code duplication in the BXT, in the BXG only one node (\( D \)) needed to be duplicated.

While one can derive the BXG by explicitly constructing the BXT and then merging nodes, there is a more efficient dynamic programming procedure that can be used to construct the BXG directly. This procedure is based on the observation that the structure of a BXT subtree with root node \( u_1 \) is a function of just two things – the node \( u \) in the original CFG that \( u_1 \) was derived from and the amount of available execution time that remains after execution has reached \( u_1 \). If the length of the path from the root to \( u_1 \) is \( p \), then the remaining execution time is \( B - p \) where \( B \) is the overall bound. We
Fig. 3. Coalescence of equivalent execution subtrees

Note that the BXT subtree with root \( u_1 \) is \( bxt_{B-p}(u) \). So two nodes \( u_1 \) and \( u_2 \) derived from the same CFG node \( u \) will have identical subtrees if the lengths of their paths from the root are identical. More generally, if their path lengths are \( p \) and \( q \), they will have identical subtrees if \( bxt_{B-p}(u) = bxt_{B-q}(u) \). This will be true for values of \( B - p \) and \( B - q \) that are “close enough” in a certain sense. For each node \( u \) in the original CFG, the dynamic programming procedure produces a partition on the integers 0 to \( B \). Two values \( i \) and \( j \) fall in the same block of the partition if and only if \( bxt_i(u) = bxt_j(u) \). Using these partitions, we can construct the BXG directly from the CFG, without having to explicitly construct the BXT. See [3] for a complete description of the algorithm, a correctness proof and execution time analysis.

III. PERFORMANCE

We have implemented this algorithm and tested it on a variety of CFGs and budgets.

A. Synthetic CFGs

Our synthetic CFGs were generated by a series of vertex substitutions that parallel grammar production rules in a C-like language. For our acyclic CFGs, we include simple statements, if, if-then-else, and switch/case statements. For our cyclic CFGs, we added while, do/while, and for loops. In both cases, the typical size of the synthetic input CFG was roughly double the size of the largest packet processing code block we have seen in our router virtualization efforts, and quadruple the target size for a typical code block.

Examine Figure 4. This represents the results of running the algorithm on 1000 different acyclic synthetic CFGs. We show the resulting distribution of the maximum code duplication factor required for each synthetic CFG over all possible budgets. The vast majority (82%) require a maximum duplication factor from 1–2, with an average maximum of 1.6. Large duplication factors are actually very rare; one pathological case required a duplication factor of 23.5. Subsequent analysis of this example showed that it was composed almost exclusively of a series of nested switch/case statements.

The results on cyclic CFGs are uninteresting and omitted. While the algorithm works on cyclic CFGs, it works by implicitly unrolling the loop to the limit of the budget. Thus, the code duplication factor is bounded only by the budget. As expected, in actual simulation the code duplication factor for cyclic graphs is linear in the budget.

B. Real CFG: IPv4 Header Rewriting

For a real CFG, we used the code that rewrites the IPv4 header for next-hop forwarding. This consists of 180 instructions, designed to run at over 5 Gbps on our virtualized router. See Figure 5. The real CFG necessitated some minor modification to the algorithm to deal with pipeline stalls due to unfilled deferral slots.

At very small budgets, the algorithm actually generates less code than the original CFG. This is due to pruning when the budget is too low for this code block. That is, so many
paths are pruned that many vertices are never emitted at all. For most application code, this represents a serious developer error and would be reported as such. It is simple for our algorithm to report when certain paths are never admitted, and we implemented this in our experimental version.

Above 108 cycles, we reach the maximum length path of the CFG. At this point, all paths are admissible and no duplication is necessary. The original CFG is accepted with no modification.

A suitable budget for 5 Gbps would be 170 cycles. Clearly, we are under 170. For 10 Gbps we need 85 cycles. The IPv4 header format code is not currently able to achieve 10 Gbps, as the chart makes obvious. Even worse, 85 cycles is the peak of our code duplication, at 296 instructions. This still yields a duplication factor of only 1.64, well in line with our synthetic cases.

IV. RELATED WORK

The major competing technology is WCET analysis using mixed integer programming [4]. This differs from our work in that it makes no effort to solve the code emission problem, and requires that we trust the developer to provide semantic information on branch constraints.

Our problem is different. We need to accept and handle untrusted code in a shared environment. Thus, we must derive any semantic information from the program, not the developer. In the absence of programmer specific semantic information, we can re-write programs to create provably safe CFGs via code duplication.

We also note that the decision to use integer programming to solve the WCET problem was because the developers considered explicit path enumeration infeasible. This fails to consider the possibilities of dynamic programming.

\[
\text{for } (i=0; i<100; i++) \{ \\
\text{ } \text{ } \text{if } (\text{rand()} > 0.5) j++; \\
\text{ } \text{ } \text{else } k++; \\
\} \\
\]

Fig. 6. “Difficult” WCET analysis for explicit path enumeration

Consider the code snippet in Figure 6. The argument is that this snippet contains \(2^{100}\) possible paths, and that to enumerate them all is simply impractical. However, using a dynamic programming approach with loop bounds, we can determine WCET for this snippet in linear time.

V. CONTINUING WORK

Our current implementation of the algorithm does not yet perform emission, nor does it incorporate a parser to accept real-world code. This is our current developmental priority.

We have also identified additional ways to reduce duplication. One immediate gain can be made by noting duplicated paths that contain no safe paths “close” to the budget. We can coalesce these paths by adding runtime checks that lengthen safe paths but do not actually push them over the budget. One possible way to reduce the expense of the runtime check is inspired by Ball and Larus [5], who developed single-counter methods for tracking execution paths through a CFG and applied those to optimize the “hot” paths. In our work, we are interested in using the same techniques to differentiate safe vs. unsafe paths.

Much greater gains can be made by extracting semantic information from the code itself. If we have complete semantic information, we can avoid path enumeration for impossible paths in the CFG. The problem becomes a limited, finite form of the Halting Problem: does this code, when started with any of the possible inputs, halt within \(B\) cycles? Any finite form of the Halting Problem is decidable.

We believe that a data flow framework solution is appropriate. With explicit path enumeration, we can solve the constant propagation problem to completion over branch conditions. This would allow us to deduce loop iteration bounds, mutually exclusive paths, and even unreachable code.

We consider this the most important area for additional study. The current state of the algorithm allows duplication to stand in lieu of semantic knowledge. Code that is semantically safe but unsafe in the CFG can be admitted by rewriting the code to guarantee that the unsafe but semantically impossible paths are never taken. With a complete semantic analysis, we would never need to strip those paths, and our code duplication would be reserved for those cases where a genuinely unsafe path is included.

In our application of event-driven, tight budget real-time guarantees, this line of research is very promising. The number of input values to examine is limited by the paucity of available cycles for reading data from memory. We know that our constant propagation will never need to deal with more than a few dozen values, because any code that examines more than this will be over budget due to memory latencies.

VI. CONCLUSION

In this paper, we have introduced a new technique for partial program admission. We have demonstrated that dynamic programming can be used to render explicit path enumeration eminently feasible. The same construction can be used to emit a modified CFG that meets event-drive real-time guarantees.

This method shows great promise in the realm of network virtualization. Other applications in similar fields may be equally promising.

REFERENCES