A Tool for Describing and Evaluating Hierarchical Real-Time Bus Scheduling Policies

Trevor Meyerowitz
UC Berkeley
Berkeley, CA 94704
tcm@eecs.berkeley.edu

Claudio Pinello
UC Berkeley
Berkeley, CA 94704
pinello@eecs.berkeley.edu

Alberto Sangiovanni-Vincentelli
UC Berkeley
Berkeley, CA 94704
alberto@eecs.berkeley.edu

ABSTRACT

We present a tool suite for building, simulating, and analyzing the results of hierarchical descriptions of the scheduling policy for modules sharing a bus in real-time applications. These schedules can be based on a variety of factors including characteristics of messages and time slicing and are represented in a hierarchical tree-like structure that specifies multiple levels of arbitration. This structure can describe many popular arbitration schemes. Our simulator evaluates the specified scheduling structure on a set of message traces for a given bus. We illustrate our approach by applying it to two examples: the SAE Automotive Benchmark and Voice Over IP (VoIP). Although this paper deals with just bus scheduling policies, the approach can be easily extended to other real-time scheduling problems.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications;
J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Performance

Keywords

Scheduling, Hybrid Scheduling, Bus Scheduling, Metrics

1. INTRODUCTION

Today many embedded systems consist of multiple processing elements communicating via a potentially complicated communication structure. This distributed nature introduces more chances for error because of the increased complexity of interaction between blocks. This design becomes even more difficult when applications, e.g., automotive control, multimedia, and network QoS (quality of service) routing, have real time constraints. This paper focuses on the representation and evaluation of various scheduling policies for real-time messages among modules communicating via a shared bus. We assume that tasks have been allocated to processing elements, and that the topology of the bus to be optimized has been defined. From this point the arbitration[1] policy can be defined, evaluated through simulation, and optimized (either manually or automatically).

Our tool suite, called *STRANG*, provides a simple hierarchical tree-based language for describing the arbitration policy for multiple nodes contending for the usage of a common bus, and for simulating the policy using a trace-driven simulator. The simulator implements the arbitration policy specified by the scheduling tree, allowing the user to explore the design space without the need for the costly and error-prone process of writing (and possibly rewriting) a custom simulator.

2. BACKGROUND

2.1 Scheduling

Much work has been done on the scheduling problem for communication blocks and task scheduling on a processor. In [6] a good overview of the common real-time scheduling methods is given. In this paper, we examine only communication scheduling. The relevant scheduling approaches can be classified as follows.

2.1.1 Event Triggered Scheduling

In event-triggered scheduling messages are selected based on their priorities, notable examples are FIFO ordering, Fixed Priority, EDF (Earliest Deadline First) scheduling, and others. FIFO and Fixed priority are simple to implement. EDF is a dynamic method that gives priority to the message with the nearest deadline. While EDF does usually produce very good results it has a large implementation overhead due to its dynamic nature.

The CAN bus [17] is an event-triggered bus protocol that has been successfully used in real-time systems such as manufacturing and automotive control. It uses a fixed priority arbitration scheme based on message *id* numbers, where each node has knowledge of the bus, and they only can contend for the bus when there is no message being transmitted. It is a highly flexible scheme that ensures that the bus will always be used if there is a message present at one of the nodes.

2.1.2 Time-Triggered Scheduling

The simplest example is TDMA (Time Division Multiple Access), the basic period is divided into a sequence of time slices, that

[1]In this paper we use the terms scheduling and arbitration interchangeably.

cycle over and over. During a particular time slice only the node associated with that time slice is allowed to access the bus. This makes it easy to ensure fairness between the nodes.

A less rigid technique is called FTDMA (Flexible TDMA). Whe–reas TDMA dedicates an entire time slice to the selected node, FTDMA dedicates that time slice only if the node has a message to send on the bus at the start of the time slice, otherwise it moves on to the next node in the following cycle. This allows FTDMA to achieve higher bus utilization and response time than TDMA, at the added cost of a more complicated arbitration logic and jitter in the response time.

TTP (Time-Triggered Protocol) utilizes the TDMA policy, resulting in a lower arbitration overhead than CAN, and has the potential for higher bandwidth utilization. While it is very easy to guarantee latencies, TTP is inflexible and potentially inefficient for transmitting non periodic messages. TTP has been used in hard real-time systems, such as automotive and avionics electronics subsystems.

2.1.3 Hybrid Approaches

Rather than selecting time-triggered or event-triggered scheduling policies, improved performance can be obtained by using a combination of the two. Recent work [1, 2] combines the predictability of TTP with the flexibility of CAN. These methods allow arbitration within some of the time slices, while keeping others exclusive, providing the flexibility of CAN with the determinism of TTP. In [8, 21, 19], hierarchical approaches such as hybrid static-dynamic and time slotting have been used for QOS (Quality of Service) applications achieving higher utilizations than traditional approaches [12] which are purely static or purely dynamic. Hybrid approaches have also been explored in multimedia domains. In [5], an MPEG decoder achieves a relatively constant rate by using hybrid methods. In [18], a Voice Over IP application is shown to benefit from customized hierarchical schedulers.

2.2 Arbitration and Communication Synthesis

Most of the previous work in this area has focused on selecting process mapping and communication topology, but doesn't focus on the arbitration policy of the bus. Usually a single simple policy is considered or a policy is selected from a library of protocols, missing out on the potential performance gains achieved by using a custom arbitration policy. Our work instead allows such optimizations, making it a nice complement to communication synthesis tools, such as the ones listed in this section.

In [13], a technique is presented for synthesizing and optimizing communication topologies connected via fixed protocols taken from a library.

In [22], a communication synthesis technique is presented for distributed embedded systems with periodic tasks that have real-time deadlines. It includes processing elements selection, task allocation, process priority assignment, and worst case timing analysis. An inverse deadline priority heuristic is used for bus arbitration.

In [15], a communication requirements graph is first constructed, and then an optimal implementation graph is selected using elements from a library of channels, multiplexors, and demultiplexors each with associated costs and capabilities. This work does not deal with arbitration between blocks.

In [16], time-triggered and event-triggered arbitration is combined in a single protocol. This is done with a top-level TDMA system that has slices open for dynamic tasks.

This paper does not specify how the arbitration occurs in the dynamic slots, and is not as flexible as STRANG, which can express schedules with arbitrary levels of hierarchy.

Dey's Communication Architecture Tuners in [10] has been a source of inspiration for this paper. Controllers are synthesized that base bus arbitration on certain properties of the messages. An example that is unschedulable using static priorities, is actually scheduled making the arbitration policy a function of multiple message characteristics (*i.e.- packet size, arrival time, and deadline*). In other examples, the number of deadlines missed is minimized. We go beyond this work by considering hierarchy, additional message variables, time-triggered scheduling and pre-emption policies. Furthermore, we provide a tool that simulates a schedule for a given bus and message trace, as well as a tool for generating the traces.

3. PROBLEM FORMULATION

We want to find a hierarchical (and potentially hybrid) arbitration policy to schedule messages between P entities communicating via a shared bus, as shown in Figure 1. We call these P entities primary nodes. The primary nodes communicate via deadlined messages over a common bus B. For a given configuration, the goal is to pick a scheduling policy that maximizes the given fitness metric (e.g. maximizing the number of messages meeting their deadlines).

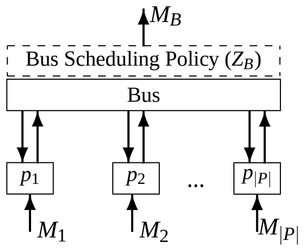

Figure 1: Physical System

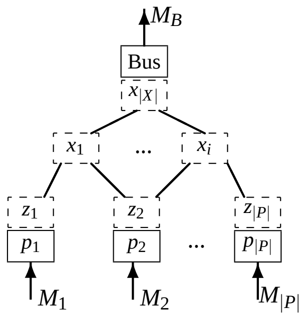

Figure 2: Bus Scheduling Policy Tree

Given message traces $M_1,..,M_{|P|}$ for the primary nodes, a message trace for the bus M_B is generated based on Z_B, the overall scheduling policy for the bus.

3.1 Messages and Nodes

Each message has a set of attributes that can be used for determining priorities. These include: the *sender-id*, the *receiver-id*, the *size* of the message in bits, the *message-id*, the message *arrival-time*, the *deadline* of the message, the *time until the deadline*, and the *time elapsed since arrival*. All of these values may be used for selecting an appropriate arbitration policy, even though we suspect that in most cases a few will suffice.

Messages become available for transmission at the primary nodes. These choose among the available messages the one to present to the bus, based on a local scheduling policy. Then the bus scheduling policy determines which among these candidate messages go first.

3.2 Arbitration Characteristics

We are interested in mixing Event-Triggered priority-based scheduling with Time-Triggered scheduling, both in preemptive and non-preemptive settings. In order to restrict the search space and for the sake of implementation efficiency we limit the admissible scheduling policies to those not requiring a fixed point computation. This requirement will be translated in the absence of cycles in the hierarchy of schedulers.

3.3 Metrics

When a trace is run through a particular arbitration tree, statistics are collected, and the tree is assigned a value based on a specified "quality" metric, called *fitness function*. The statistics collected for each message include whether it was transmitted or not, and its transmission start and finish times.

Fitness functions provided in STRANG that can be used to rank arbitration policies are:

- The number of missed deadlines.
- The overall execution time
- The average throughput of the bus

Users can specify custom functions if they so desire.

4. SCHEDULER REPRESENTATION AND EVALUATION

In this section we explain how to specify a scheduling problem using STRANG, provide several example schedulers, and explain the semantics of the simulator. A full explanation is provided in [11].

4.1 Scheduler Configuration

Communication scheduling policies in STRANG are represented as an acyclic directed graph, like in Figure 2. It is useful to distinguish between primary nodes (leafs of the graph) and arbitration nodes. In particular removing the primary nodes we are left with the arbitration tree, that itself models the bus scheduling policy. Each arbitration node has a priority function that specifies how it selects between its children for event-triggered arbitration, and an allocation policy which specifies the time-triggered arbitration.

Message selection is made up by composing the selections of individual nodes in a bottom up fashion.

The bus is specified by the arbitration tree along with the bus characteristics specified in the configuration file. These characteristics are: *cycle-time*, *bandwidth* (bits per cycle), *message-overhead* (in bits), and the overheads for *arbitration* and *preemption* (in bus cycles).

4.2 Tree Representation

The arbitration tree is specified using the syntax shown in figure 3. First custom operation tree policies are specified (represented by *P*). Next, the top level arbitration node is specified. Finally children of the top node are specified. These children can be arbitration nodes (represented by *A*), or primary/sender nodes (specified by *S*). *PolicyID* indicates the policy number or predefined policy name used by the node. The predefined policies in STRANG are: FIFO, LIFO, EDF, and Fixed Priority. $Preemption$ is the preemption policy. $Alloc$ indicates the style of time allocation used by the node. *SndrID* is the *id* of the node used at the sender node. Finally, the number of children and a list of that length of durations (in cycles) are specified.

4.2.1 Operation Trees

As indicated above, custom scheduling policies can be specified by operation trees instead of using the predefined policies. The operation tree represents the function used to describe the different policies for sorting between various messages at a particular node. Each policy is a function of the 8 message variables and, in the case of arbitration nodes, the $child\_id$, which is based on the ordering of the arbitration-node children.

```
#PolicyFunctions
(P PolicyID_1 OpFunc1) ...
(P PolicyID_N OpFuncN))
(A PolicyID_a Alloc Preemption #Children (Durations)
        (child_1) ...
        (S PolicyID_ci Preemption SndrID) ...
        (A...)...
        (child_#children)
)
```

Figure 3: Arbitration Tree Syntax

The operation tree uses floating point constants in addition to addition, subtraction, and multiplication as operators. Division is not allowed because it would be difficult to check "divide by zero" errors. We use prefix ordering to ease parsing of these trees.

4.2.2 Sample Trees

Figure 4 shows a CAN tree. It has a custom priority based upon the *id* of the messages used by all nodes and a single arbitration node with the primary nodes as children.

Figure 5 shows a TTP tree. It has the same topology as the CAN tree. The top level node is based on time allocation, the children having slices of 10, 20, 30 cycles respectively.

Figure 6 mixes time-triggered and event-triggered scheduling domains. It has a top level node that has a fixed priority so it always favors its first child over its second child. The first child presents messages from the first sender in the order specified by the custom priority function $P1 = messageID + deadline$, the structure of which is shown in **Figure 7**. The second child is a TDMA node with 2 senders as children, namely, 2nd and 3rd senders with FIFO ordering of their messages.

```
1
(P 1 messageID)
(A 1 NONE NO 3 (0 0 0)
      (S 1 NONE 1)
      (S 1 NONE 2)
      (S 1 NONE 3)
)
```

Figure 4: Simple CAN Tree

```
0
(A FIFO TDMA NO 3 (10 20 30)
      (S FIFO NONE 1)
      (S FIFO NONE 2)
      (S FIFO NONE 3)
)
```

Figure 5: Simple TTP Tree

```
1
(P 1 + messageID deadline)
(A FIXED NONE NO 2 (10 10)
    (S 1 NONE 1)
    (A FIFO TDMA NO 2 (10 10)
        (S FIFO NONE 2)
        (S FIFO NONE 3) )
)
```

Figure 6: Simple Hybrid Tree

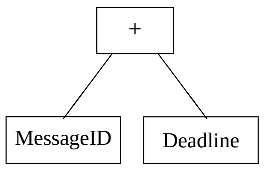

Figure 7: Sample Operation Tree

4.3 Scheduler Evaluation

There are two parts to this section, how a given policy, configuration, and trace are simulated to get results, and how to evaluate such results.

4.3.1 Simulator Behavior

The simulator is a trace-driven discrete-event simulator. The simulator obtains its timing and configuration information from the configuration file. Message traces are loaded from the trace file, where each message is loaded into the event queue based on its arrival time. Next, the scheduling tree is loaded. If the scheduling tree has time dependent modes, then the mode update events are added to the event queue. From here the events are popped out of the event queue and executed in order.

When a message arrives it is placed in a message queue at the appropriate sender node. The scheduling tree is used to select which message will be transmitted on the bus. When a message begins transmission on the bus, the bus state changes to running, and an event is scheduled for when the transmission ends. Unless there is preemption there can be no other messages submitted to the bus while one is transmitting.

4.3.2 *Evaluating a Design*

Once a policy is simulated, one may wish to improve the quality of his/her design. The simulator outputs the complete message trace with *begin* and *end* of transmission times making it easy to calculate the quality of the results according to an appropriate fitness function. In section 5.1.4, we will show how to use the analysis to improve the scheduling policy.

We are exploring the use of automatic techniques such as the tracer tokens from [10] and genetic algorithms to aid in the exploration of the design space.

5. EXPERIMENTAL RESULTS

In this section, we provide a set of examples that have been run using the tool. We begin by evaluating CAN and TTP solutions to the SAE automotive control benchmark. From here, we optimize both solutions; CAN by adding EDF arbitration and TTP by sharing some time slots. We then evaluate different scheduling policies for a Voice over IP (VoIP) example.

5.1 The SAE Automotive Control Benchmark

Here we compare the results of several different protocols at the bus speeds of 100Kbps, 125Kbps, and 250Kbps for the SAE (Society of Automotive Engineers) benchmark as laid out in [17]. The SAE benchmark has 53 message types that travel between 7 nodes in a system, as shown below in figure 8. The messages are either sporadic or periodic, each with required deadlines, jitter, and average period. For all of the results we use 5 second long message traces, these traces vary based on the clustering of the messages.

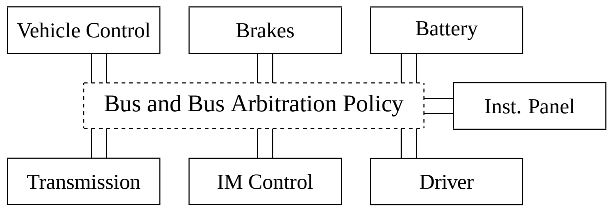

Figure 8: Physical System

5.1.1 *CAN Bus*

Here we use the simplified solution from [17], where different messages are grouped together to reduce the total number of message types to 17. Also, we follow their solution and give sporadic messages a regular period of 20ms to represent the worst case.

In the CAN bus each message has an 11-bit priority identifier, and the bus is constructed in such a way that it will only accept writes from the highest priority message. To represent a CAN-like bus we merely have to specify the priority based on the message-ID number, and specify a estimated overhead of 55 bits[2].

Below the results of running the CAN solution to the SAE control benchmark is shown at 3 different bandwidths, 100Kbps, 125Kbps, and 250Kbps. These are evaluated on the same 7365 message trace.

Bandwidth	**100Kbps**	**125Kbps**	**250Kbps**
Deadlines Missed	746	0	0
Bus Utilization	>100.0%	84.4%	41.7%
Message Utilization	22.2%	18.6%	9.3%
Median Deadline Slack (ms)	3.91	4.43	4.72
Min. Deadline Slack (ms)	-5072.56	1.28	3.81

Figure 9: Regular CAN SAE Results

Bus Utilization refers to the percentage of the time that the bus is in use. *Message Utilization* refers to the percentage of time that the bus is being used to transmit the actual data. A message's *Deadline Slack* is its deadline minus its transmission completion time.

5.1.2 *TTP Bus*

In [9] Kopetz presents a TTP solution to the SAE benchmark, but doesn't fully explain how messages are grouped to achieve the general schedule. Even without this information we can model their solution[3]. Following Kopetz's solution, we leave out the instrument panel messages in this benchmark.

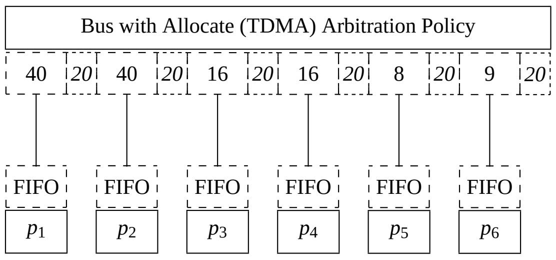

Figure 10: TTP-style Arbitration Tree

Figure 10 shows a graphical representation of the arbitration tree used for the SAE solution. In it the arbitration node for the bus is a time allocate node, which is segmented into different slots. Each slot contains the number of cycles that it is active for, and may have a line connecting it to the node that it sends messages from. Every other slot is a dummy node used to model the TTP overhead.

We use the full 53 message traces, with the sporadic messages having a strict period of 50ms. We use the same 12481 message trace for all TTP and modified TTP results.

Figure 11 gives the results of the SAE automotive benchmark running on a TTP bus at 3 different speeds. At each bus speed all of the deadlines are met. Every instance automatically has an overhead of at least 48% because each time slice has 20 cycles devoted to the CRC and TTP overhead.

[2]CAN has a fixed overhead of 47 bits per message, but employs bit stuffing when there are 5 identical bits in a row. We estimate this by adding an overhead of 8 (out of a possible 19) bits per message.
[3]We simply define the configuration as having an additional sender node, with no message overhead or arbitration overhead. Every other time slot is the dummy sender with duration of 20 cycles to represent the overhead of TTP. This allows us to implicitly model the clustering of Kopetz's solution.

Bandwidth	100Kbps	125Kbps	250Kbps
Bus Utilization	64.4%	61.2%	54.7%
Message Utilization	16%	12.8%	6.4%
Median Deadline Slack (ms)	4.39	4.53	4.78
Min. Deadline Slack (ms)	0.26	1.23	3.07

Figure 11: TTP SAE Results

5.1.3 *CAN with EDF*

We model the work done in [7], where a CAN bus protocol is modified to use EDF arbitration, instead of being based on the message-id. We keep the same CAN trace, and add 3 to the message overhead to account for the added complexity of EDF.

Bandwidth	100Kbps	125Kbps	250Kbps
Deadlines Missed	7235	0	0
Bus Utilization	$>$100%	86.9%	43.5%
Message Utilization	21.4%	18.6%	9.3%
Median Deadline Slack (ms)	-207.4	4.4	4.7
Min. Deadline Slack (ms)	-425.1	2.74	4.0

Figure 12: CAN with EDF SAE Results

The bus utilization of the EDF extension to CAN is higher than regular CAN because of our added overhead. At 125Kbps the median deadline slack does decrease slightly, but the more critical minimum deadline slack improves from 1.28ms to 2.74ms.

5.1.4 *Shared TTP*

To try to increase the flexibility of the TTP solution, we examined the message results. We observed that messages from the Sender 1 (the first primary node) have significantly higher deadline slacks than those from Sender 6. Sender 2 also has significant slack when compared to Sender 5. We modified the scheduling tree so that Sender 6 can use Sender 1's slot when Sender 1 isn't using it, and that Sender 5 can use Sender 2's slot when Sender 2 isn't using it. A graphical representation of this tree is shown in figure 13, and the input file for it is shown in figure 14.

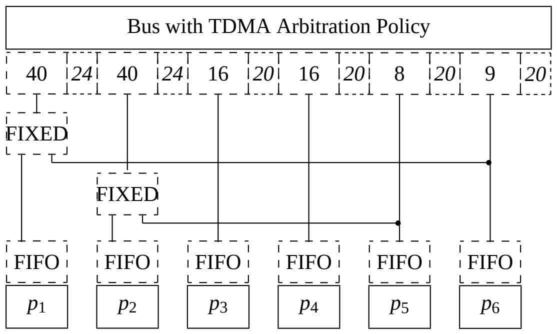

Figure 13: Shared TTP Arbitration Tree

To model the increased complexity of shared slots we added an overhead of 4 cycles to each of the shared slots. By doing this we increased the minimum deadline slack, making the system more jitter tolerant.

As can be seen the minimum slack is greatly improved over that of TTP at the lowest bit rate, and slightly improved at the highest bit rate.

```
0
(A FIFO NONE TDMA 12
        (40 24 40 24 16 20 16 20 8 20 8 20)
        (A FIXED NONE NO 2 (0 0)
                (S FIFO NONE 1)
                (S FIFO NONE 6)
        )
        (S FIFO NONE 8)
        (A FIXED NONE NO 2 (0 0)
                (S FIFO NONE 2)
                (S FIFO NONE 5)
        )
        (S FIFO NONE 8)
        (S FIFO NONE 3) (S FIFO NONE 8)
        (S FIFO NONE 4) (S FIFO NONE 8)
        (S FIFO NONE 5) (S FIFO NONE 8)
        (S FIFO NONE 6) (S FIFO NONE 8)
)
```

Figure 14: Shared TTP Arbitration Text

Bandwidth	100Kbps	125Kbps	250Kbps
Bus Utilization	65.2%	62.8%	56.4%
Message Utilization	16%	12.8%	6.4%
Median Deadline Slack (ms)	4.56	4.65	4.84
Min. Deadline Slack (ms)	0.53	1.28	3.33

Figure 15: Shared TTP SAE Results

5.1.5 *SAE Results Discussion*

These results are summarized in Figure 16. TTP is more effective than CAN for lower bandwidths. If the bandwidth is higher, then CAN exhibits faster response times. Adding EDF to CAN improves matters even further. Through careful analysis of the TTP message trace we were able to substantially improve the minimum deadline slack by sharing some time slots.

5.2 Voice over IP Benchmark

We have taken 5 seconds from 6 generated VoIP traces and simulated them on a shared 128kbps link with different arbitration policies. We obtained the information about the G.729A voice codec, the delay overheads, and the protocol overheads from [20] of 40 bytes per packet. The G.729A produces 10 byte samples, which occur every 10ms. Between 1 and 10 samples from G.729A can be clustered into a packet. The 6 streams have 1, 4, 4, 5, 5, and 6 samples per packet respectively. The delay of each packet can be calculated with the following formula, $D = 5 + 10 * N$, where N is the number of samples and D is the queueing delay in milliseconds. A stream's performance is acceptable if it has a one-way total latency of less than 150ms. To account for transmission and decode time we give each sample a deadline of 100ms. A packet in a stream has a deadline of 100ms minus the queueing delay. For each of these we assume a jitter of 0.1 ms, and no arbitration overhead.

We evaluate this using 4 types of arbitration policies: EDF, FIFO, Fixed Priority with RMS (Rate Monotonic Scheduling), Fixed Priority with DMS (Deadline Monotonic Scheduling). The *RMS* solution is a fixed non-preemptive ordering where the messages with the shortest periods have the highest priorities. The *DMS* solution orders the messages, where the ones with the shortest deadlines get the highest priorities, which in this case is the opposite ordering of *RMS*. Because the messages have deadlines greater than their periods, none of the theoretical guarantees about *RMS* or *DMS* apply.

As expected, EDF provided the best result with a minimum deadline slack of 5.07ms. The FIFO policy achieves a worse minimum

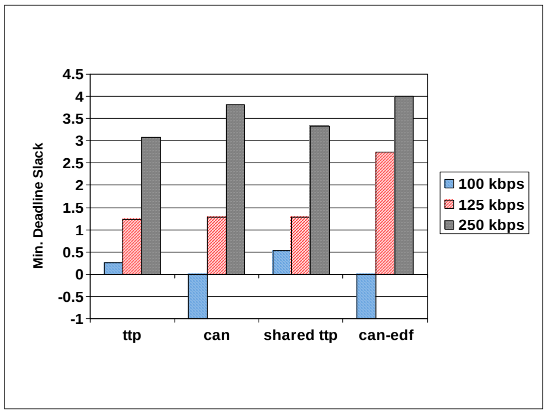

Figure 16: SAE Min. DS Results Summary (Negative results truncated at -1ms for graphical clarity)

Policy	EDF	FIFO	DMS	RMS
Deadlines Missed	0	0	0	25
Avg. Deadline Slack (ms)	57.29	50.71	57.29	58.01
Med. Deadline Slack (ms)	50	50	50	50
Min. Deadline Slack (ms)	5.07	1.79	5.07	-23.05

Figure 17: Voice over IP Benchmark Results

deadline slack of 1.79ms. The surprising result was that the DMS fixed priorities achieved the same results as the EDF, at a lower implementation cost. On the other hand, the RMS solution misses some deadlines.

6. CONCLUSIONS

We have motivated and formulated the problem of scheduling real-time messages on a shared bus, as well as shown the benefit of using hierarchical arbitration policies for optimizing the schedule. We presented a tool that can represent a wide variety of trees, and simulate them using message traces. We exercise the tool on several non-trivial examples and has shown results consistent with the literature, and have improved upon them using hierarchical schedulers.

STRANG specifies scheduling in a general sense and can easily be applied to more than just buses. It could easily be generalized to evaluate scheduling policies in a variety of other domains. Possible domains include: deadlined resource contention problems, RTOS scheduling, and QOS Network Routing.

The scheduling tree structure is extensible, and natural to work with. Two simple extensions are: the addition of round robin scheduling and token-bucket models. Constructs should be added for specifying non-trivial overheads, and custom cost functions. Additionally, the ability to estimate the complexity of a custom cost function needs to be added to make the results from optimization tools (such as the genetic algorithm we are currently developing) more meaningful. Future work includes: expanding STRANG to handle multiple resources, synthesizing protocols for the given arbitration schemes, and adding more flexible mode switching.

The simulator should be modified so that it could be easily interfaced with other simulators such as [4], and system-level design environments such as [3], as well as point tools such as [15, 14] to create a complete real-time system design flow.

7. ACKNOWLEDGEMENTS

This research was funded in part by the Semiconductor Research Corporation and the MARCO GSRC program.

8. REFERENCES

[1] Byte flight consortium web site. http://www.byteflight.com.
[2] FlexRay consortium web site. http://www.flexray-group.com.
[3] Metropolis group web site. http://www.gigascale.org/metro.
[4] SystemC web site. http://www.systemc.org/.
[5] L. Abeni and G. Buttazzo. Hierarchical qos management for time sensitive applications. In *Seventh IEEE Real-Time Technology and Applications Symposium*, pages 63–72. IEEE Comput. Soc., 2001.
[6] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, 15(1):71–82, Jan.-March 1998.
[7] M. Di Natale. Scheduling the can bus with earliest deadline techniques. In *21st IEEE Real-Time Systems Symposium*, pages 259–68. IEEE, 2000.
[8] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *21st IEEE Real-Time Systems Symposium*, pages 207–16. IEEE, 2000.
[9] H. Kopetz. A solution to an automotive control system benchmark. In *Proceedings of 1994 Real-Time Systems Symposium*, pages 154–8.
[10] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey. Communication architecture tuners: a methodology for the design of high-performance communication architectures for system-on-chips. In *Design Automation Conference*, pages 513–18. IEEE, June 2000.
[11] T. Meyerowitz and A. Sangiovanni-Vincentelli. Describing, simulating, and optimizing hierarchical bus scheduling policies. UCB/ERL Technical Report M03/5 available at: http://www-cad.eecs.berkeley.edu/˜tcm.
[12] K. Nicols, V. Jacobson, and L. Zang. A two-bit differential services architecture for the internet. In *Internet Draft*, 1997.
[13] R. Ortega and G. Borriello. Communication synthesis for distributed embedded systems. In *Proc. Int. Conf. Computer Aided Design*, pages 437–444, June 1998.
[14] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *35th Design Automation Conference*, pages 8–13. ACM, June 1998.
[15] A. Pinto, L. Carloni, and A. Sangiovanni-Vincentelli. Constraint-driven communication synthesis. In *Proceedings 39th Design Automation Conference*, pages 783–8, 2002.
[16] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 187–92. ACM, 2002.
[17] K. Tindell and A. Burns. Guaranteeing message latencies on controller area network (can). In *Proceedings of 1st International CAN Conference*, September 1994.
[18] C.-Y. Wang, C.-C. Hsu, and Y. Huang. Voral: a system for voice over ip routing in application layer. In *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, pages 165–70. IEEE Comput. Soc., 2001.
[19] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *21st IEEE Real-Time Systems Symposium.*, pages 239–48. IEEE, 2000.
[20] D. Wright. *Voice Over Packet Networks*. John Wiley & Sons Ltd., Chichester, 2001.
[21] C.-H. Yeh. Scalable qos supports for multimedia applications in the next-generation internet. In *Proc. IEEE Real Time Technologies and Applications Symp.*, pages 39–50. IEEE, May/Jun 2001.
[22] T. Yen and W. Wolf. Communication synthesis for distributed embedded systems. In *Proc. Int. Conf. Computer Aided Design*, pages 288–294, November 1995.