DESCRIBING, SIMULATING, AND OPTIMIZING HIERARCHICAL BUS SCHEDULING POLICIES

by

Trevor C. Meyerowitz and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M03/5

12 March 2003

DESCRIBING, SIMULATING, AND OPTIMIZING HIERARCHICAL BUS SCHEDULING POLICIES

by

Trevor C. Meyerowitz and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M03/5

12 March 2003

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Describing, Simulating, and Optimizing Hierarchical Bus Scheduling Policies

Trevor C. Meyerowitz, Alberto Sangiovanni-Vincentelli

March 12, 2003

Abstract

We present a tool suite for working with hierarchical bus schedule/arbitration descriptions for nodes sharing a bus in real-time applications. These schedules can be based on a variety of factors including time slicing and the characteristics of messages. These schedules are represented in a tree-like structure. Using this structure we can describe many popular arbitration schemes, and automatically evaluate them using our simulator that implements the scheduling policy of the tree. Additionally, we provide a genetic algorithm for automatically exploring the design space. As a proof of concept we apply these tools to several real examples.

Chapter 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. Such design becomes even more difficult when the applications have real time constraints. Examples of such applications include automotive control and network quality of service routing. Communication-based design, as advocated in the metropolis project [12], addresses these concerns by making the communication a full part of the design methodology as opposed to an afterthought. In this report we present a flexible language and tool that allow for the representation of a wide variety of arbitration[1] policies. Here we apply it to the problem of scheduling realtime messages onto a bus, but it is quite general, and could easily be applied to other scheduling problems with little or no modification.

This work focuses on optimizing the scheduling of real-time messages from nodes communicating with one another via a shared bus. It assumes that tasks have been allocated to processing elements, and the topology of the bus to be optimized has been defined. From this point the arbitration policy can be defined, evaluated through simulation, and optimized (either manually or automatically).

This work can be viewed by itself, but for our purposes we present it within the context of the Y-chart methodology. The Y-chart methodology is a popular methodology for designing heterogeneous embedded systems, but hasn't adequately addressed real-time systems. With our tool suite arbitration is made explicit, allowing optimization of time-constrained messages sharing a given resource (in this case a bus). Our approach is shown in this context in figure 1.1, where the steps in bold indicate those implemented in our tool. The dashed box with arbitration optimization indicates a novel step that, to our knowledge, hasn't before been integrated into such a flow.

Many HW/SW co-design tools, such as POLIS [26], follow the Y-chart method and provide environments to describe applications and map them to hardware or software. However, often there is no control over how the interfaces between blocks are synthesized, or the scheduling of communication. In the case of POLIS all of the communication is point to point, and only the RTOS's schedule can be explicitly specified. Metropolis [12] advocates separation of communication, computation, and coordination. Typical systems for co-design have only addressed one or two of these areas and

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

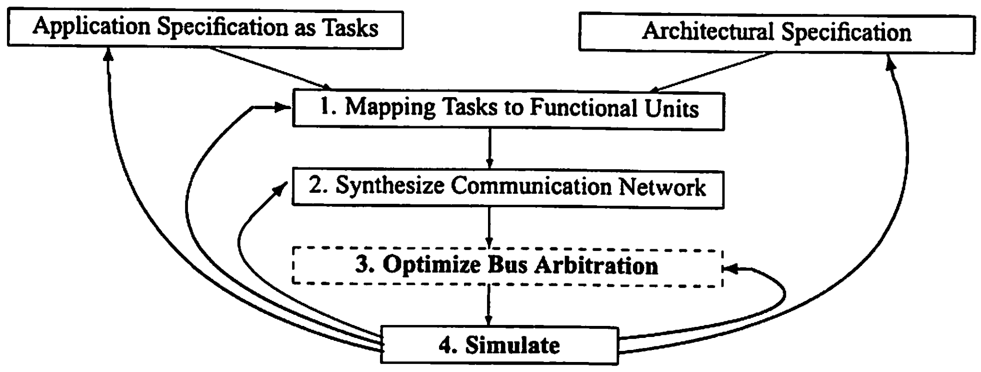
Application Specification as Tasks
Architectural Specification
1. Mapping Tasks to Functional Units
2. Synthesize Communication Network
3. Optimize Bus Arbitration
4. Simulate

Figure 1.1: **Our Expanded Design Flow**

had fixed solutions for the remaining, potentially leaving much room for optimization on the table.

In it the application is a set of communicating tasks, and the architecture is a set of processing elements that can implement these tasks. The next step is to map the tasks to various processing elements. After this the communication system must be synthesized, and then the resultant system is simulated to see if it meets the performance requirements.

If performance constraints aren't met, then there traditionally are 3 options for improving the design. The first option is to go back to the start of the problem and change the application and/or architectural specifications. The second option, which is less drastic, is to redo the mapping. The third option is to resynthesize the communication network of the design. Even with all of these options, it is often difficult to achieve real-time performance, pointing to the need for further optimization of the communication structure.

As pictured in Figure 1.1, we add optimizing bus arbitration policies as another level of refinement in the Y-chart methodology. This ability is not typically available in previous communication synthesis work. In this way the performance of real-time traffic on the bus can be optimized without modifying the communication topology. This allows for a finer tuned optimization of a single bus within a given communication system.

Our tool suite is named STRANG, standing for "A Scheduling Tree Language". It provides a simple hierarchical tree-based language for describing the arbitration policy for multiple nodes contending for the usage of a common bus, and then simulating the policy using a trace-driven simulator. The simulator automatically implements the arbitration policy specified by the scheduling tree, drastically expanding the design space while eliminating the need for costly and error-prone rewrites of it. Also included are a trace generation tool, and a genetic algorithm for automating the design space exploration problem.

1.1 Background

In this section we present background information in several areas. First we talk about techniques used in scheduling, and some motivating applications. After this we review the work of communication synthesis systems, and explain how our tool can improve upon them.

1.1.1 Scheduling

Much work has been done on the scheduling problem for communication blocks and task scheduling on a processor. In this work we only examine the scheduling of communication. We begin by explaining event-triggered scheduling. We then discuss time-triggered scheduling. We finish by talking about work that has utilized hybrid approaches to achieve improved results.

Event Triggered Scheduling

Scheduling is a large area, and many methods are considered common knowledge. [27] provides a good overview of the common real-time scheduling methods, most of which are event triggered. As the name implies event triggered scheduling bases its arbitration on messages with different priorities These include FIFO ordering, Fixed Priority, EDF (Earliest Deadline First) scheduling, and others. FIFO and Fixed priority are simple to implement, but are less effective than EDF. EDF is a dynamic method that gives priority to the message with the nearest deadline. While EDF does produce very good results it has a very large implementation overhead due to its dynamic nature.

The CAN bus[1] is an event-triggered bus protocol that has found success in realtime 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.

Time Triggered Scheduling

Time triggered architectures base their arbitration on the time, which determines which node controls the bus. TDMA policy (Time Division Multiple-Access) that gives each node a specific time slice where it alone can use the bus. This makes it easy to ensure fairness between the nodes.

The second technique is called FTDMA (Flexible TDMA). Whereas TDMA dedicates an entire time slice to the selected node, FTDMA only dedicates that time slice if the node has a message to send on the bus at the start of a 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, with the tradeoff being that more complicated arbitration logic is needed to implement FTDMA.

TTP (the time-triggered protocol) utilizes the TDMA policy, giving it a lower arbitration overhead than CAN, and the potential for higher bandwidth implementations. While it is very easy to guarantee latencies with TTP, it's also inflexible and potentially inefficient. TTP has been used in hard-realtime systems, such as automotive applications.

Hybrid Approaches

Rather than select either time-triggered or event-triggered scheduling policies, many applications have found that they can achieve improved performance by using a combination of the two approaches. Here we reference hybrid approaches in domains ranging from automotive to multimedia.

Recent work [13, 14] has combined the best features TTP and CAN to form hybrid arbitration policies that offer the flexibility of CAN. They allows arbitration within some of the time slices, while keeping other ones exclusive, providing the flexibility of CAN with the determinism of TTP. Recent work [18, 19, 21] investigating hierarchical approaches such as hybrid static-dynamic and time slotting for Quality of service applications has achieved higher utilizations than purely static or dynamic approaches that are traditionally [16] used. Hybrid approaches have also been explored in multimedia domains. In [20] an MPEG decoder is optimized using such approaches so that it decodes at a relatively constant rate and in [23] Voice Over IP is shown to benefit from customized hierarchical schedulers.

1.1.2 Communication Synthesis

Communication synthesis for distributed embedded systems is a very well studied problem. Most of the previous work in this area has focused on selecting the process mapping and the communication topology, but don't focus on the policy arbitration on the bus. Usually they just pick a single simple policy or select from a library of protocols, potentially missing out on performance optimizations gained from picking a custom arbitration policy. Our work allows such optimizations, making it a nice complement to communication synthesis tools, such as the ones listed in this section.

In [6] Boriello describes a technique for synthesizing and optimizing communication topologies connected via fixed protocols taken from a library. This is somewhat similar to [7] where Gasteier and Glesner synthesize communication topologies for statically schedulable systems. These approaches create a very restrictive design space, that can only be expanded by manually writing additional library elements. We are focus on optimizing the communication in a single bus for a given topology, which can expand and finely tune the above techniques.

In [4], Wolf and Yen, describe a communication synthesis technique for distributed embedded systems with periodic tasks that have realtime deadlines. It includes selecting the number of PE's (processing elements), task allocation, process priority assignment, and worst case timing analysis. For process priority assignment they use rate monotonic scheduling, and use an inverse deadline priority heuristic for bus arbitration. We are only focusing on a single bus, but allow a much wider variety of bus policies to be simulated, potentially improving on the performance of buses in this approach.

Constraint-Driven Communication Synthesis [25] starts with a communication requirements graph, and then selects an optimal implementation based on selecting from a channels, multiplexors, and demultiplexors each with associated costs and capabilities. This work doesn't specify how the arbitration between the blocks should be specified, making this tool a potentially nice complement to it.

1.2 The Bus Scheduling Problem

Scheduling occurs when multiple entities contend for a smaller number of shared resources. Figure 1.2 shows such a case with nodes communicating with a shared bus. We call these N entities primary nodes. These primary nodes communicate via deadlined messages over a common bus B. For a given configuration, the goal is to pick the best scheduling policy for the senders so as to maximize the number of messages that make their deadlines. This scheduling policy is made up of a scheduling policy at each node, a scheduling policy at the bus, and sometimes intermediate arbitration nodes. We call these policies the scheduling trees. They are actually directed-acyclic graphs because the leafs can be shared (and there can only be one scheduling policy per primary node). We represent these policies as trees. Scheduling begins at the primary nodes (or leafs) and ends at the top node. Each node has a specific scheduling policy that it implements at that point.

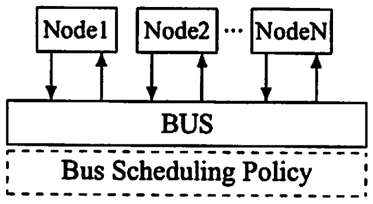

Figure 1.2: **Physical System**

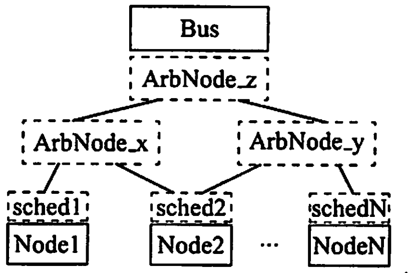

Figure 1.3: **Bus Scheduling Policy Tree**

The tree representing the bus scheduling policy of 1.2 is shown in Figure 1.3. At the bottom are the primary nodes (also referred to as the sender nodes) that send and receive the messages. Each of these have a scheduling policy for the messages queued for sending itself, these are represented by the dashed boxes called sched1, sched2 and schedN. These select the fittest message based on the arbitration policy and then present it to the parent node(s) (in this case ArbNode_x and ArbNode_y). The parent nodes then select the highest priority message from those presented by their children. These messages are again passed upwards until the top arbitration node is reached (in this case ArbNode_z). There can only be one top arbitration node and this is the node that decides which message will be sent on the bus at any particular time. We lay this out in greater depth in the next chapter.

1.3 Related Work

Distributed scheduling using genetic algorithms is explored in [11]. The genetic algorithm only applies to non-preemptive offline scheduling, whereas we allow for dynamic scheduling and preemption. Additionally the algorithm is only applied to contrived examples, making it difficult to evaluate its quality.

The Network Simulator (or ns) [31] is a very complete network simulation framework. It provides a large number of protocols, and scripting capabilities using TCL and C++. It models connections as direct links (or wired channels), and has no notion of shared links. Additionally, there is no built-in notion for arbitration policies, making it necessary to hard-code such examples and making exploration time consuming. ns isn't intended for realtime systems, and this shows, it is, however, useful for QOS-type applications.

[29] examines combining time-triggered and event-triggered arbitration in a single protocol. It does it by creating a single TDMA system that has slices open for event-triggered (or dynamic) tasks. This is essentially a two-level scheduling hierarchy, with time-triggered arbitration being at the top level. It doesn't specify how the arbitration occurs in the dynamic slots, and isn't as flexible as our suite, which can express arbitrarily hierarchical schedules.

[30] looks at supporting both real-time and multimedia tasks on a shared memory multiprocessor machine. It uses a planning-based scheduler, which determines if the set of tasks is schedulable based on their worst-case execution times, and if not it doesn't allow the admission of the new tasks. Furthermore they provide 4 possibilities for scheduling policies, with the choices being flexible vs static and individual vs proportional. The first choice involves statically scheduling multimedia servers, or dynamically scheduling. The second choice is whether to have one server per multimedia task, or share tasks among servers. These scheduling techniques are more for servers, and not applicable to most real-time embedded systems.

Dey's Communication Architecture Tuners in [3] inspired much of this work. In it they describe synthesizing controllers that base bus arbitration on certain properties of messages in a distributed system. For example they present a QOS (quality of service) modification of TCP/IP with the 3 steps of the checksum, ip_check and ethernet driver scheduled on a shared bus. In this example there is no way to schedule the system using static priorities, but by making the arbitration policy of multiple message characteristics (in this case: packet size, arrival time, and deadline) they are able to reduce the number of deadlines missed. We go beyond this work by explicitly adding the additional message variables, time division multiplexing and preemption policies in the hierarchical tree-based schedule description language. Furthermore we provide a simulate that implements the policies specified by the trees, and a genetic algorithm to explore the design space.

1.4 Overview of Paper

In this chapter we've introduced the tool and talked about previous related work and how it compares to our work. In the next chapter we lay out the problem space in mathematical terms. In the third chapter we explain how to represent scheduling and simulate scheduling problems using

STRANG. In chapter 4 the genetic algorithm used for automating the exploration of the design space is discussed. In chapter 5 results are listed for simple examples, an automotive example, a QOS example, and of the genetic algorithm. In the final chapter conclusions are drawn, and future work for extending STRANG is discussed.

Chapter 2

Problem Formulation

This chapter mathematically expresses the problem of scheduling deadlined messages from nodes connected together with a shared bus. Each message has a message id number, a source node, a destination node, a size, an arrival time, and a deadline. The goal is to select an arbitration policy in the form of a scheduling tree (such as the one shown below in Figure 2.1), so that the majority of messages meet their deadlines (or some other user-defined cost function is optimized). The notation and ideas presented here are similar to that of the tagged-signal model [15].

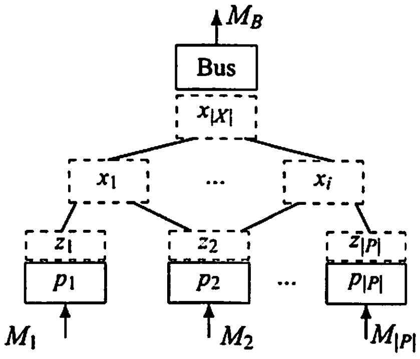

Figure 2.1: **Bus Scheduling Policy Tree**

2.1 Nodes and Messages

P is the set of primary nodes communicating via the shared bus. X is the set of extra (or arbitration) nodes that build on top of other nodes (both primary and extra) to construct the scheduling tree. $N = P \cup X$, and represents the set of nodes. The i-th node is referred to as $n_i \in N$. The first $|P|$ nodes in N are the primary nodes, and the remaining $|X|$ nodes are the extra nodes. The arbitration policies will be described in more depth in section 2.2.

2.1.1 Messages

Each message has a set of values. $S(m_{i_j}) \in P$ represents the *sender node* of the message. $R(m_{i_j}) \in P$ represents the *receiver node* of the message. $Size(m_{i_j})$ is the *size* of the message in bits. The *arrival time* and *deadline time* of the message are represented as $A(m_{i_j})$ and $D(m_{i_j})$ respectively, with their co-domains being time. Each message also has a *message id number*, $ID(m_{i_j}) \in \mathbb{N}$, which can be used as a priority. We assume that all of these characteristics can be used for defining arbitration policy functions.

2.1.2 Primary Nodes and Message Traces

The message trace for primary node p_i is called M_i. It is assumed fully ordered, if two messages arrive at the same time they are non-deterministically ordered by the system. Because they are schedule dependent, we don't define message traces for extra nodes.

The j-th message to arrive at primary node p_i is called m_{i_j}.

$M_i(t)$ is a subset of the trace messages at primary node p_i up until time t, it includes all messages that have arrived at the node (they could have been transmitted or not).

The scheduling policy of a primary node can only be determined by its operation tree. The *operation tree* specifies the scheduling function for the given node. It can either be a custom tree or a predefined function.

2.1.3 Arbitration Nodes

Arbitration (or extra) nodes are nodes that have no message trace of their own, and receive messages from their children. These children can either be sender nodes, or other arbitration nodes.

Arbitration node x_i has a sequence of children, a sequence of durations, an allocation policy, and a scheduling policy. The scheduling policy is based on either the allocation policy, or, if this policy isn't time sliced, the operation tree. The scheduling policy of a time sliced node on its k-th slice is simply selecting the message from the k-th child. Because of this arbitration nodes need a notion of which mode they are in, we refer to this as the state of the node.

2.1.4 The Bus and the Arbitration Tree

All legal arbitration trees must have at least one arbitration node. There is the restriction that no arbitration node can be the parent of its predecessor. There is also the restriction that a valid tree can only have one top node. These conditions are enforced by requiring the arbitration structure to be a tree-like structure.

The bus represents the top node of the arbitration tree, along with the bus characteristics. The bus characteristics are: Cycle Time, Bandwidth (bits per cycle), and the overheads (in bits) for messages, arbitration, and preemption.

2.2 Scheduling Policies

$MZ_i(t)$ is the set of messages at node z_i at time t (for a given scheduling policy).

$mz_i(t)$ is the highest priority message at node z_i at time t (for a given scheduling policy).

Each node has a scheduling policy Z_i that selects the message submitted to the parent node arbitration. This policy chooses from the messages visible to the node at that moment.

$Z_i : MZ_i(t) \rightarrow mz_i(t)$, meaning that Z_i selects from all of the messages currently at the node, and decides which one to submit from the node's parent (or to the bus if it is the top arbitration node).

The bus also has a scheduling policy, which we refer to as Z_B is the scheduling policy of the bus. Z_B selects a message based on its: current state, the current time, and the messages submitted to it. It is the aggregate of the scheduling policies in the arbitration tree.

The message trace coming from the bus is called M_B and depends on Z_B.

$m_B(t)$ is the message being transmitted on the bus at time t. Note that if nothing is being transferred on the bus, then $m_B(t) = \emptyset$, where $\emptyset$ is the empty set.

2.2.1 The Operation Tree

The operation tree defines the scheduling policy for a non-time-sliced node. The priority can either be a predefined function (Such as EDF, FIFO, et al.), or as a custom function specified by the user. It is important to note that the predefined functions can all be represented as custom functions. The custom function is made up of variables based on the message characteristics, constants, and the +, -, and * operators. It is represented in a tree-style format. Below is the tree corresponding to the function $Priority = Deadline * (Size - 5.0)$. Examples of how to specify this are provided in the next chapter.

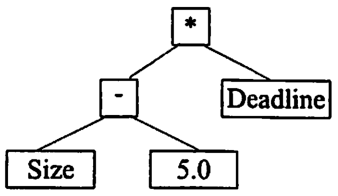

Figure 2.2: Sample Operation Tree

2.2.2 Time Slicing

Time slicing can only be applied to non-primary nodes. There are 4 different time slicing policies. "NONE" just does normal arbitration between the children. "ENFORCE" is like "NONE" in that it keeps the arbitration policy, but the node can only transmit messages that take under the number of cycles specified for that particular slot. For "TDMA" each child gets a certain number of cycles of control of the arbitration node. FTDMA is similar to TDMA, but each child only gets an initial cycle to control the node, and if it doesn't the next child is selected If the selected slot for FTDMA does have a message within the first cycle, then the child gets the additional cycles to control the arbitration node.

2.2.3 Preemption

Preemption occurs when a message of higher priority than the selected message arrives at a node with preemption enabled. There are 3 types of preemption: ABORT, SUSPEND (no resumption, you have to restart), SUSPENDwMEMORY (Like SUSPEND, but with resumption). For ABORT preemption the selected message is discarded and the higher priority message takes its place. For SUSPEND preemption the selected message returns to the queue, and the higher priority message takes its place. SUSPENDwMEMORY preemption is the same as suspend preemption, except the whole message doesn't need to be retransmitted, only the parts that haven't yet been transmitted.

2.3 Fitness Evaluation

When a trace is run on a particular arbitration tree, statistics are collected, and the tree is assigned a fitness based on the specified fitness evaluation function.

The below are statistics collected during the simulation that will be used afterwards for collecting fitness results.

- $F(m_{i_j})$ is the finish time of the message. If the message never transmitted or was aborted, then $F(m_{i_j}) = \infty$.
- $DM(m_{i_j})$ is a boolean function which is true if the deadline for m_i has been satisfied. It can be expressed as $F(m_i) \leq D(m_{i_j})$

Here is a list of the current fitness evaluation functions:

1. The number of missed deadlines.
2. The overall execution time
3. The average throughput of the bus

We are planning on expanding this list, and adding to the syntax so that these functions can be specified in the configuration file. Currently the only way to change the evaluation function is to modify the source code.

Chapter 3

Representing and Simulating Bus Schedules

In this chapter we explain how to specify a scheduling problem using STRANG. The first section provides and overview simulator and describes the environment. The second section describes the specification of scheduling trees. The third section follows with several example trees.

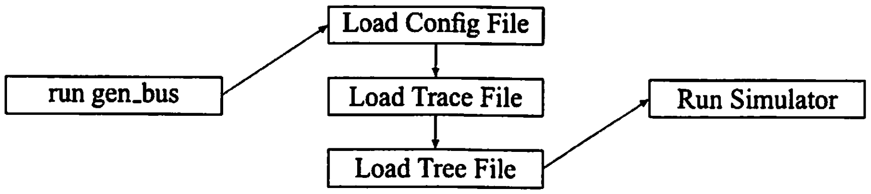

Figure 3.1: The Simulation Flow

3.1 The Simulator and Environment

3.1.1 Simulator Overview

The simulator is a trace-driven discrete event simulator. The simulator gets 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 dependant 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 serves 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. The behavior of the simulator will be discussed in greater detail, as parts of the system are further described in this chapter.

3.1.2 Configuration File

In the configuration file the fixed environment variables are defined for the simulator. These define the environment, as well as the overheads of the particular operations. The variables in the configuration file are: the number of primary nodes connected to the bus, the cycle time, the bandwidth (in bits per cycle), and the overheads (in bits) for messages, arbitration, and preemption. The cycle time is divided a certain amount, to provide a level of granularity smaller than that of the cycle (currently we use a cycle granularity of 10).

The below figure shows an annotated configuration file. This file consists of 4 numbers, the first one is the cycle time and the last three are the overheads (in cycles) for messages, arbitration, and preemption.

Cycle Time	Message Overhead	Arbitration Overhead	Preemption Overhead
0.01	4	3	0

Figure 3.2: **Sample Configuration**

3.1.3 Trace File

The tracė file contains a message trace as well as values for the number of primary nodes and the number of message types. The messages are presented in temporal order. Each message contains the following information: the message type, the sender node number, the recipient node number, the arrival time, the deadline, and the size of the message (typically in bits). These messages are read into the system, and then are placed into an event queue based upon their time information.

```
2.0 10 3
9 1 1 0.844416 10.8444 16
3 2 1 1.09819 6.09819 8
8 1 3 2.56932 12.5693 8
5 1 3 2.75021 7.75021 8
4 2 2 2.8431 7.8431 16
2 3 3 4.11124 9.11124 16
6 1 1 4.27312 24.2731 32
3 1 1 6.12031 11.1203 8
10 2 2 7.02415 17.0242 16
```

Figure 3.3: **Sample Trace (Actual)**

Mess #	Sender	Receiver	Arrival	Deadline	Size
9	1	1	0.844416	10.8444	16
3	2	1	1.09819	6.09819	8
8	1	3	2.56932	12.5693	8
5	1	3	2.75021	7.75021	8
4	2	2	2.8431	7.8431	16
2	3	3	4.11124	9.11124	16
6	1	1	4.27312	24.2731	32
3	1	1	6.12031	11.1203	8
10	2	2	7.02415	17.0242	16

Figure 3.4: **Sample Trace (Annotated)**

Figure 3.3 shows how a simple trace looks in STRANG. Figure 3.4 shows an annotated version of this trace. The first line of the trace indicates of the configuration, with the first number being the trace version, the next number is the number of message types, and the third being the number of primary nodes in the system.

Also provided is a program used to generate random traces of the proper format with certain distributions. This isn't specifically needed to run STRANG, but can be used generates random traces of data for scheduling onto the bus. The trace generator is more fully described in appendices A and B.

3.2 Tree File

This arbitration tree is specified using the scheduling tree syntax, as shown below in figure 3.5. First custom operation tree policies are specified. Next, the top level arbitration node is specified. Finally children of the top node are specified, these children can be additional arbitration nodes, or they can be senders.

```
[# Custom Op Trees]
(P [PolicyID1] [Operation Function])
...
(P [PolicyIDN] [Operation Function])
(A [PolicyID] [Preemption] [Alloc] [#children] (durations)
        (child_1)
        (S PolicyID Preemption SndrID)
        ...
        (child_n)
)
```

Figure 3.5: **Scheduling Tree Syntax**

3.2.1 The Operation Tree

The first line in Figure 3.2 indicates the number of custom operation trees specified. These aren't strictly necessary if the user utilizes the predefined policies.

Predefined Operation Tree Policies

The predefined policies are explained below. Notice that all of these policies can be specified using custom operation tress.

- **FIFO** - Messages are ordered strictly by their arrival times with the earliest arriving messages having priority.
- **LIFO** - This policy is the opposite of FIFO, where the latest arriving messages have priority.

- **FIXED** - This policy is only for arbitration (non-sender) nodes, where priority is purely based on the child location, with the first child having priority over all of the others.
- **EDF** - In this policy the messages with the earliest deadlines have priority.

Custom Operation Trees

Custom policies can be specified by operation trees instead of using the predefined policies. The options for specifying these are shown below in figure 3.6.

```
(P [PolicyID_number] [Variable])
(P [PolicyID_number] [Constant])
(P [PolicyID_number] [Operation Left_Child Right_Child])
```

Figure 3.6: **Operation Tree Syntax**

The operation tree represents the function that is used to describe the different policies used to sort between various messages at a particular node. Each policy is a function of the 9 different variables: arrival-time, deadline, message size, message-type, sender-id, receiver-id, the child order (only for arbitration nodes), the time elapsed since the arrival of the message, and the time until the deadline of the message. For a full explanation of this syntax see Appendix B.

The function can also use floating point constants, and addition, subtraction, and multiplication as operators. Division isn't used because it would be difficult to check divide by zero errors. We use prefix ordering to ease the of parsing the operation functions.

Operation Tree Examples

Below we have 3 custom operation tree examples. The first example tree has the ID of 1, and its value is the size of the message. The second tree has its value as the sum of the sender identifier and the receiver identifier. Finally, the third tree has the following value "Sender - (2.2* MessageSize)".

1. (P 1 size)
2. (P 2 + senderID receiverID)
3. (P 3 - sender * size 2.2)

3.2.2 Primary Node Syntax:

Primary, or sender nodes are specified as follows. They begin with the "S" keyword. This is followed by either the ID number of a custom policy function, or the name of one of the predefined policy functions. After this the preemption policy and the id of the sender node are specified. Note that each primary node can only be assigned one scheduling policy.

3.2.3 Arbitration Node Syntax:

The arbitration node uses the "A" keyword. Then the policy function (predefined or custom) is specified. After this the preemption policy is specified. Thirdly an allocation policy is specified. The four allocation policies, described in the next paragraph, are Allocate, None, Enforce, and Flex-allocate. All but the "None" mode use the durations that are specified for each of the children. The durations must be positive integers and represent multiples of the cycle time that is specified in the configuration file (See appendix for this). Finally the children are listed in order inside of the block, which is then closed by a parenthesis.

3.3 Example Arbitration Trees

All of the trees in this section have the exact same tree structure, and only vary in their arbitration policies. They each have 3 sender nodes and 1 arbitration node. In the below figure the scheduling policies are indicated by the dashed boxes bordering the nodes. We have non-time-sliced and time sliced trees, and end with an example with custom priority functions.

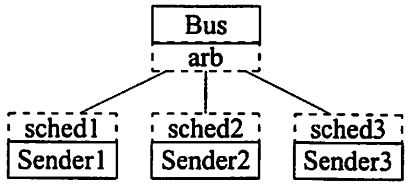

Figure 3.7: **Arbitration Tree Structure**

3.3.1 Simple Non-Timesliced Trees

The first example (Figure 3.8) is a LIFO (last in first out) tree, where each node uses this policy where the last arriving message receives priority. Neither preemption nor time allocation policies are used here. Note that it is perfectly possible for the children nodes to have different arbitration policies than the parent nodes. If the sender nodes had FIFO arbitration, then they would each present their oldest message to the arbitration node, which would then select the newest of these.

The second example tree (Figure 3.9) is an EDF (Earliest Deadline First), where the message with nearest deadline is given priority.

```
0
(A LIFO NONE NO 3 (0.0 0.0 0.0)
        (S LIFO NONE 1)
        (S LIFO NONE 2)
        (S LIFO NONE 3)
)
```

Figure 3.8: **LIFO Tree**

```
0
(A EDF NONE NO 3 (0.0 0.0 0.0)
        (S EDF NONE 1)
        (S EDF NONE 2)
        (S EDF NONE 3)
)
```

Figure 3.9: **EDF Tree**

3.3.2 Simple Timesliced Trees

The two immediately below figures demonstrate two trees with time-sliced policies. For both of these trees each child has a slot of 40 cycles where it has full access to the bus. For the flexible TDMA allocation policy the if the first cycle of a slot doesn't have any messages, then the next slot is selected for the next cycle. This way better utilization can be achieved.

```
0
(A FIFO ALLOCATE NO 3 (40 40 40)
        (S FIFO NONE 1)
        (S FIFO NONE 2)
        (S FIFO NONE 3)
)
```

Figure 3.10: **TDMA Tree**

```
0
(A FIFO FLEXALLOCATE NO 3 (40 40 40)
        (S FIFO NONE 1)
        (S FIFO NONE 2)
        (S FIFO NONE 3)
)
```

Figure 3.11: **Flexibile TDMA Tree**

3.3.3 Trees With Custom Priority Functions

Here we have a tree with FIFO priorities on the senders, message-based priority at the intermediate node, and a priority function of the message size times the deadline at the top.

```
2
(P 1 messageID)
(P 2 * size messageID)
(A 2 NONE NO 2 (40 40)
        (S FIFO NONE 1)
        (A 1 NONE NO 2 (40 40)
                (S FIFO NONE 2)
                (S FIFO NONE 3)
        )
)
```

Figure 3.12: **Custom Tree**

Chapter 4

Exploring The Design Space

Part of the power of the tree representation is that it is easy to succiently represent a wide variety of schedules. Furthermore, it is quite easy to change an arbitration tree, be it drastically or incrementally, with slight edits. Using the simulator we can quickly evaluate the performance of different policies on traces. On top of this we have written a genetic algorithm to rapidly explore the design space and optimize the arbitration policy for a given trace.

Genetic algorithms first appeared in [8]. For a more modern description of them refer to [9]. The main idea is that solutions to the problem are represented as strands or trees of information that can easily be modified or bred. Through breeding and mutation of various generations large solution spaces can be effectively explored. They have been effectively applied in different areas including branch predictors [10] and distributed scheduling[11]. Our tree description of the arbitration policies and of the priority functions make them quite amenable to genetic algorithms.

4.1 Genetic Algorithm Overview

The genetic algorithm, shown in figure 4.1, explores the design space functions as follows. First the trace, configuration, and possibly an initial tree are loaded. After this the genetic algorithm runs for a prespecified number of times (or generations). Each generation has a specified number of trees that are generated and then each tree is simulated on the same trace, and then assigned a fitness value based on the simulation results. The top trees from the previous generation are saved, as are a certain number from the current generation. From the current generation the top few members are automatically passed on to the next generation, and then a small number of the remaining nodes are selected.

4.1.1 Configuration of the Genetic Algorithm

The configuration of the genetic algorithm plays a strong part in its effectiveness. The parameters include the number of new trees per generation, the number of trees kept at the end of the generation,

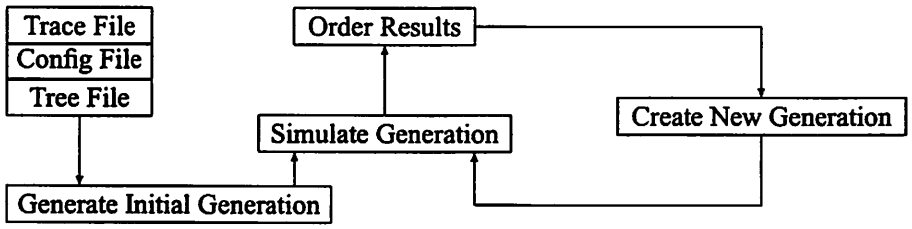

Figure 4.1: **The Genetic Algorithm Flow**

the number of survivors from the previous generation, and the number of winners from the current generation. Additionally, there are different rates for mutations and combinations that must be considered, as well as the number of operations done to generate each new tree (currently we only have 1 operation for each tree).

4.2 The Initial Generation

The initial generation is created with the hopes of covering a large area of the search space. The space is always seeded with a small set of trees, a FIFO tree, a Fixed Priority Tree, and a TDMA tree. If a starting tree has been specified on the command line, then it is also added to the array of seed trees. From here we generate the initial generation through a series of random breeding and mutating steps.

4.3 Evaluating a Generation and Creating a New Generation

All of the trees in the current generation are run through the simulator, and then they are ordered based upon their fitness values. From here the top several finishers are immediately copied to the next generation. Then a number of random trees are selected for the next generation as well. From these trees copied from the prior generation, a number of new trees are generated by mutating or breeding the copied trees.

4.3.1 Breeding Trees

Breeding takes 2 trees from the current generation duplicates them, and then breeds the duplicates to create new trees for the next generation. In STRANG there are two ways to breed arbitration trees. Either two operation trees can be bred together, or two arbitration trees themselves can be bred.

Breeding Operation Trees

Two arbitration trees are passed on to the operation tree breeding function. From these trees two nodes are selected for breeding. The operation trees from these nodes are passed into the operation tree breeding function. In this function one node from each operation tree is selected, and then these nodes (and any children that they may have) are swapped to yield new policy functions, that are applied to the particular nodes within the arbitration tree.

In the below figures we begin with 2 operation trees and show a valid breeding of them.

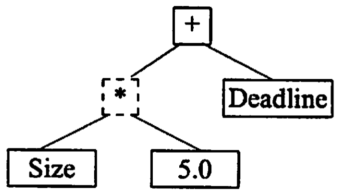

Figure 4.2: **Original Operation Tree 1**

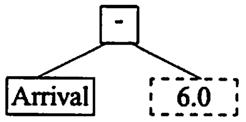

Figure 4.3: **Original Operation Tree 2**

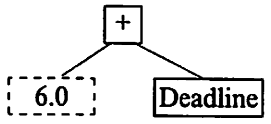

Figure 4.4: **Final Operation Tree 1**

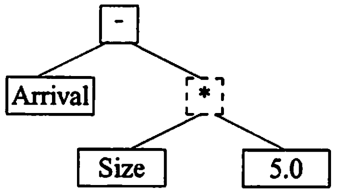

Figure 4.5: **Final Operation Tree 2**

Breeding Arbitration Trees

The other breeding option is to breed the arbitration trees themselves. For this two nodes are selected from each arbitration tree. This breeding can occur in two different ways either breeding of the individual nodes, or breeding of the branches rooted at these nodes. For breeding individual nodes the characteristics of the two nodes are exchanged, but no structural interchange occurs.

For breeding branches each branch of the two nodes selected *(Note: they cannot be the top of the tree)* is exchanged between the trees. This will often result in highly unbalanced trees, but is good at escaping local minima in the solution space. To avoid inconsistency if two non-sender nodes are bred, then each tree keeps its respective senders. Figures 4.6 and 4.7 are the 2 trees before breeding with nodes ArbNodey_1 and ArbNodex_2 being selected for breeding. Figures 4.8 and 4.9 show the respective trees after breeding the selected branches.

4.3.2 Mutating Trees

The other technique in genetic algorithms is to mutate nodes by slightly changing a single tree to encourage diversity. These changes can take two forms functional and structural mutations.

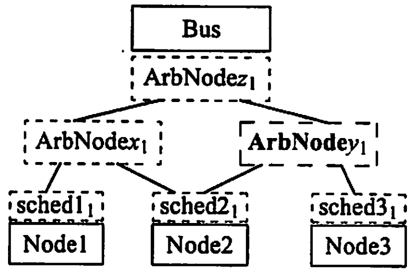

Figure 4.6: **Original Arb. Tree 1**

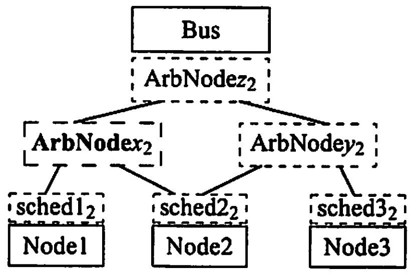

Figure 4.7: **Original Arb. Tree 2**

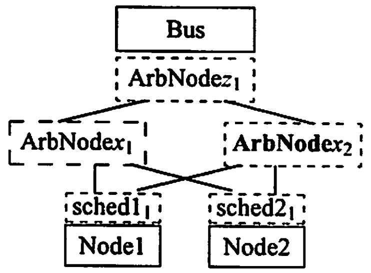

Figure 4.8: **Final Arb. Tree 1**

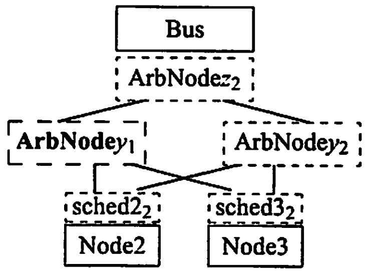

Figure 4.9: **Final Arb. Tree 2**

Functional mutations preserve the structure of the tree, and modify the parameters of nodes within the tree. Structural mutations modify the structure of the tree by adding, moving, or removing nodes within the tree.

Functional Mutations

Mutations can functionally change the individual parameters of nodes in the arbitration tree. The operation tree for a given node may be swapped for another one, or mutated itself. The preemption or allocation policies may be changed. The durations in an arbitration node might be changed. Finally, for sender nodes the sender that node is attached to may be changed.

Structural Mutations

Structural mutations involve adding, removing, or moving nodes in the arbitration tree. To start with a particular node (typically an arbitration node) is randomly selected from the chosen tree, then one of the below structural mutations is chosen.

The most basic operations are the addition or removal of children of the arbitration node. The removed node is deleted if it has no other parents. Added children are randomly selected sender nodes. Another very basic operation is swapping the positions of two children in the tree, this can affect fixed priority and time slotted schedules.

The final two structural mutations are the combining children and collapsing nodes. The combine operation takes a number of children away from the selected arbitration node and puts them under a newly generated arbitration node. The new arbitration node is then added as a child to the selected arbitration node. Collapsing a node is the opposite of combining children. The selected arbitration node is removed and all of its children are added to its parent[1].

4.3.3 Preventing Illegal Cases

Sometimes the genetic algorithm will generate trees that cannot be simulated. One example is having a TDMA with all of the durations being 0 (this leads to infinite execution). Illegal trees are deleted, and the program several times attempts to generate legal trees in their place before giving up.

4.4 Genetic Algorithm Status

The genetic algorithm does currently work. It hasn't been optimized for good performance as of yet. The percentages and rates of different tree generation strategies need to be evaluated. Furthermore, we probably want to introduce bounds on the size that the trees can become. Finally, and most importantly, the genetic algorithm really needs an estimator of the overhead of a given policy, so as to fairly compare policies.

4.5 Other Approaches

Genetic algorithms are by no means a panacea. They can often over-optimize for a given trace. Additionally they sometimes produce unnecessarily complicated solutions that often must be pruned to be realistically used. Even if our genetic algorithm proves not to be useful, one can apply other techniques to it such as simulated annealing, branch and bound search, and heuristic schedule generation. Much of this depends on the trace presented to the system. If it is fully periodic, then it may be quite easy to find an exact solution. However, if the messages are non-periodic it is difficult to prove much about their behavior in realtime systems. If worst case bounds can be determined for the message periods, then it is possible to analyze the results. If not, we must fall back to simulations.

[1]Obviously the top node of the arbitration node cannot be selected to be collapsed

Chapter 5

Examples and Results

In this chapter we provide a set of examples that have been run using the tool. With these we verify the simulator by comparing it with the results of previous work, and also go beyond it using the hierarchical scheduling description. We begin with a simple example. Then we move on to evaluating multiple solutions to the SAE automotive control benchmark. After this we present a VoIP(Voice Over IP) networking example. Next, we evaluate the genetic algorithm on the CAN solution to the SAE automotive benchmark. We finish with a discussion of these results.

5.1 A Simple 3 Node System

Here we have a simple example where there are 3 nodes that send messages between one another. Each of the messages has a source, a destination, an arrival time, a size, and a deadline. There are 10 types of messages for this example. The actual trace is shown below in figure 5.1.

The impact of different schedules is readily apparent here. The treeLIFO scheduling tree shown below misses one deadline, while the treeEDF using the Earliest Deadline First scheduling technique doesn't miss any deadlines. Additionally we also use this example to compare trees using TDMA and Flexible-TDMA policies. The configuration policy, the message trace, and all of the trees are taken from chapter 3.

Schedule	**Deadlines Missed**	**Exec Time (cycles)**
treeLIFO	1	1668
treeEDF	0	1668
treeTDMA	0	2760
treeFTDMA	0	2440

Figure 5.1: **Simple Trace Results**

5.2 The SAE Automotive 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 [1]. The SAE benchmark has 53 message types that travel between 7 nodes in a system, as shown below figure 5.2. The messages are either sporadic or periodic, each with required deadlines, jitter, and average period. In this section we replicate the results for TTP and CAN solutions to this benchmark, and finish by evaluating two novel solutions (CAN with EDF arbitration, and TTP with shared clustering). For all of the results we use 5 second long message traces, these traces vary based on the clustering of the messages.

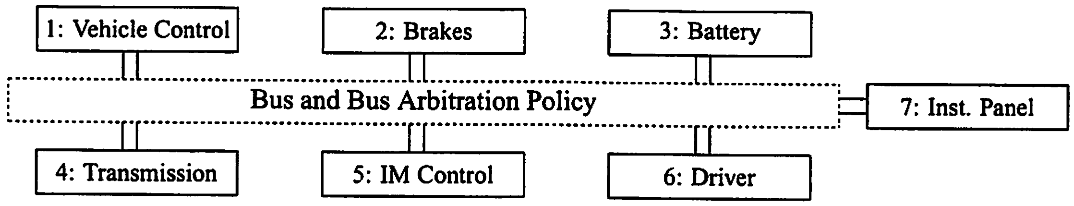

Figure 5.2: **Physical System**

5.2.1 A CAN Bus Solution

Here we use the simplified solution from [1], where different messages are grouped together to reduce the total number of message types to 17. Also, we follow their solution and use purely periodic signals with a period of 20ms to represent the worst case for sporadic messages. The CAN bus is shown below in figure 5.3, it is a simple bus where everything's priority is based on the message id number, where smaller numbers have higher priorities.

```
1
(P 1 messageID)
(A 1 NO NONE 3 (0 0 0 0 0 0 0)
        (S 1 NONE 1)
        (S 1 NONE 2)
        (S 1 NONE 3)
        (S 1 NONE 4)
        (S 1 NONE 5)
        (S 1 NONE 6)
        (S 1 NONE 7)
)
```

Figure 5.3: **CAN-style Scheduling Tree**

The CAN Bus Tree

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. Since each node is aware of the values on the bus, every node knows which node has received control of the bus, and the other nodes then defer to that node. To represent a CAN-like bus we merely have to specify the priority based on the message ID number, and specify an estimated overhead of 55 bits[1].

CAN Results

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 5 second long trace that consists of 7365 messages.

Category	CAN(100Kbps)	CAN(125Kbps)	CAN(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 5.4: **Regular CAN SAE Results**

5.2.2 A TTP Bus Solution

In [22] 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[2]. Following Kopetz's solution, we also leave out the instrument panel messages in this benchmark.

We use the full 53 message traces, with the sporadic messages have a strict period of 50ms. We use the same trace for all TTP and modified TTP results. This trace is 5 seconds long and it contains 12481 messages.

TTP Tree Representation

Figure 5.5 shows the text used to enter the tree into STRANG. Figure 5.6 shows a more intuitive graphical representation of the same arbitration tree. In it the arbitration node for the bus is a time

[1]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.

[2]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 and it goes for the message overhead (20 cycles in this case). This allows us to have the implicit clustering of the messages, while still modelling the overhead imposed by TTP.

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 has a time of 20 and isn't connected to any node, this is because it is connected to the dummy node (sender 8), which is used to model the overhead of a TTP solution using clustering without knowing the exact clustering technique.

```
0
(A FIFO NONE ALLOCATE 12 (40 20 40 20 16 20 16 20 8 20 8 20)
        (S FIFO NONE 1)     (S FIFO NONE 8)
        (S FIFO NONE 2)     (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 5.5: **TTP-style Arbitration Tree (In STRANG)**

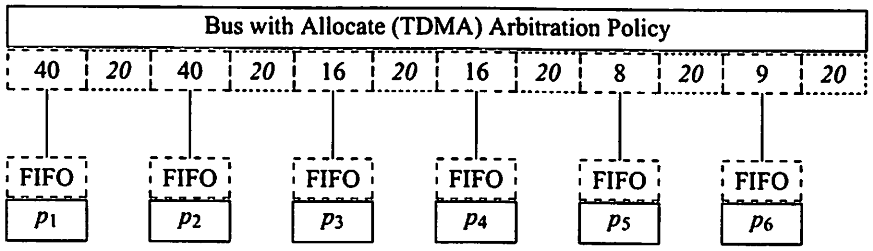

Figure 5.6: **TTP-style Arbitration Tree (Graphical)**

TTP results

Here we have results for the SAE automotive benchmark running on a TTP bus at 3 different speeds. Every instance automatically has an overhead of at least 48% because each time slice has 20 cycles devoted to the CRC and TTP overhead.

Category	**TTP(100Kbps)**	**TTP(125Kbps)**	**TTP(250Kbps)**
Deadlines Missed	0	0	0
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 5.7: **TTP SAE Results**

5.2.3 Other SAE Solutions

Here we describe two improvements upon the CAN and SAE solutions. The first is a method for adding EDF scheduling to CAN. The second shares some of the TTP time slots to improve the minimum deadline slack.

CAN with EDF

We model the work done in [17], where a CAN bus protocol is modified to Earliest Deadline First Arbitration instead of fixed message-id protocol. We keep the same CAN trace, and add 3 to the message overhead to account for the added complexity of EDF priority resolution. The tree is exactly the same as the CAN tree with EDF replacing MessageID as the priority function. The CAN-EDF tree is simulated on the same trace as the CAN trace.

Category	CEDF(100Kbps)	CEDF(125Kbps)	CEDF(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 5.8: **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 is much improved (from 1.28ms to 2.74ms).

Shared TTP Description

To try to increase the flexibility of the TTP solution, we have made it so that the 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. In their respective slots senders 1 and 2 have priority. We selected these by examining the message results from TTP and noticing that messages from senders 5 and 6 had the least slack, and those from 1 and 2 had the most. By doing this we hope to increase the minimum deadline slack, a good indication of how jitter-proof the arbitration scheme is. To model the increased complexity of shared slots we add an overhead of 4 cycles to each of the shared slots.

Shared TTP Results

Look at Figure 5.11 for these results. These results are taken using the same trace as the TTP results. 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 ALLOCATE 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 5.9: **Shared TTP-style Arbitration Tree (In STRANG)**

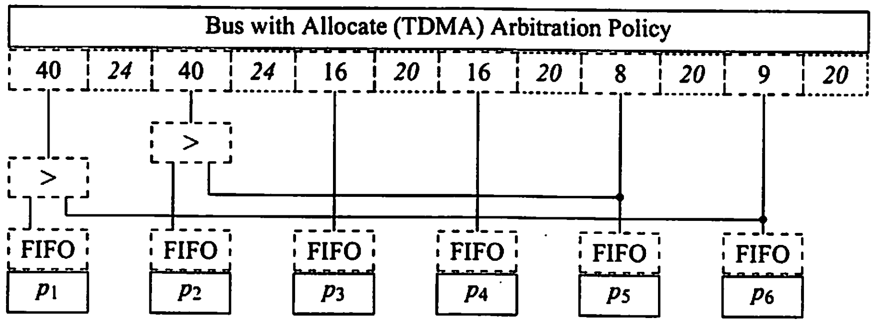

Figure 5.10: **Shared TTP-style Arbitration Tree (Graphical)**

Category	STTP(100Kbps)	STTP(125Kbps)	STTP(250Kbps)
Deadlines Missed	0	0	0
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 5.11: **Shared TTP SAE Results**

5.2.4 SAE Results Discussion

TTP is more effective than CAN for lower bandwidths. If the bandwidth is higher, than 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 nodes. Figure 5.12 shows the minimum deadline slack for all of the nodes,

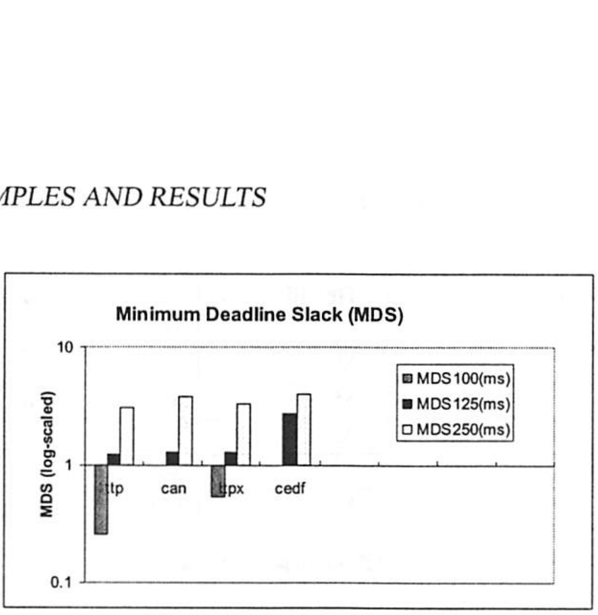

Figure 5.12: **Minimum Deadline Slack for SAE benchmark results**

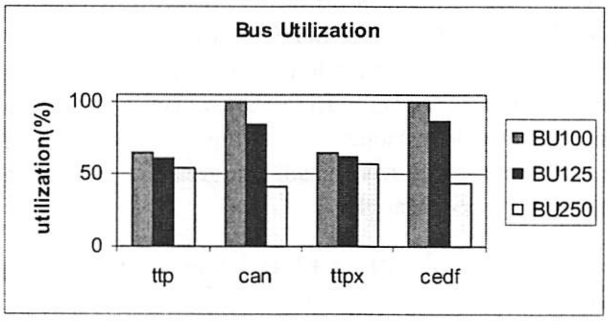

Figure 5.13: **Bus Utilization for SAE benchmark results**

on a logarithmic scale (the real values range from -5072ms to 3.8ms). Figure 5.13 lists the bus utilizations of the different solutions.

5.3 Voice over IP Benchmark

We have taken 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 (RTP, UDP, and IP) overheads from [32]. 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. Each packet has an overhead of 40 bytes. 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 VoIP stream's performance is considered acceptable if its one way total latency is less than 150ms.

Because there may be a more hops to travel to reach the final destination, we set the deadline for each sample to be 100ms. The actual deadline for each stream is equal to 100ms minus the queueing delay. For this benchmark we compose 6 G.729A VoIP streams with various sample clustering sizes

Number	Samples per Packet	Size(bits)	Deadline(ms)	Period(ms)	Jitter(ms)	Kbps
1	1	400	85	10	0.1	40
2	4	640	55	40	0.1	16
3	4	640	55	40	0.1	16
4	5	720	45	50	0.1	14.4
5	5	720	45	50	0.1	14.4
6	8	960	15	80	0.1	12

Figure 5.14: **VoIP Message Streams Data**

sharing a single 128kbps, with 0 overhead for arbitration. We have 1 stream that carries a single sample per packet, 2 that carry 4 samples per packet, 2 that carry 5 samples per packet, and 1 that carries 8 samples per packet. These streams are shown in figure 5.14. 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 longest periods have the highest priorities. ended up missing some deadlines. The *DMS* solution orders the messages, where the ones with the shortest deadlines get the highest priorities, this happens to be the exact opposite ordering of the *RMS* scheme. Note that because the messages have deadlines greater than their periods, none of the theoretical guarantees about *RMS* or *DMS* apply.

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

Figure 5.15: **Voice over IP Benchmark Results**

As expected, EDF provided the best result with a minimum deadline slack of 5.07ms. The FIFO policy 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 fixed priority solution actually misses some deadlines. We haven't yet explored preemptive solutions, or time-triggered solutions.

5.4 Genetic Algorithm Results

It is hard to do a real comparison here because TDMA and fixed priority protocols should have different overheads. Instead we'll just run the algorithm on the CAN solutions and see what comes out in the various generations.

We take the CAN-SAE benchmark solution at 125Kbps and optimize it using the genetic algorithm. In order to have the genetic algorithm optimize both the we use the following cost function:
Fitness = *(Num Deadlines missed * 1000) - minimum deadline slack.*

For each generation we create 2000 new trees and simulate them. In addition to this we keep the top 50 winners, and the top 50 from the previous generation. The results are shown below in figure 5.16, and one of the winning trees is shown in figure 5.17.

Category	CEDF	Gen 1	Gen2	Gen3
Min. Slack(ms)	2.87	1.67	2.08	2.87

Figure 5.16: **Genetic Algorithm Results**

```
1
(P 1 messageID)
(A EDF NO NONE 3 (0 0 0 0 0 0 0)
        (S 1 NONE 1)
        (S 1 NONE 2)
        (S 1 NONE 3)
        (S 1 NONE 4)
        (S 1 NONE 5)
        (S 1 NONE 6)
        (S 1 NONE 7)
)
```

Figure 5.17: **Genetically Generated Tree**

As can be seen, the algorithm quickly converges on a partial-EDF solution, where each of the sender nodes are fixed priority, and only the top level arbitration node has EDF arbitration. This achieves the same minimum slack as a fully EDF arbitration tree, and possibly has a lower cost.

Chapter 6

Final Words

6.1 Conclusions

In this report we motivated and formulated the problem of scheduling realtime messages on a shared bus, as well as shown the benefit of using hierarchical arbitration policies for optimizing such a schedule. From there we presented a language that, along with a simulator, can represent a wide variety of trees, and simulate them using message traces. Furthermore, we've presented a genetic algorithm for automatically exploring the design space. The tool has been exercised on several non-trivial examples and has shown results consistent with the literature. Additionally the tool has been used to improve upon the solutions from the literature. The genetic algorithm has also been evaluated and has shown interesting initial results.

6.2 Limitations

Despite the promising results, this work has a number areas that need improvement. Some of these are related to the tree language itself, others involve the simulator, and still others come from the genetic algorithm.

STRANG assumes full knowledge of the system, where each node can keep an arbitrarily large set of messages continually ordered based on certain criteria, and that each message contains the full information about itself. These assumptions can lead to unimplementable schedules, and often times scheduling is based on an a subset of the information used in STRANG. Furthermore, the evaluation policy of a priority function may be too complicated to realistically implement, and STRANG can't currently estimate the complexity of particular arbitration policies.

Another problem is that the configuration for a single run is considered fixed. This includes the message, arbitration, and preemption overheads, as well as the cost function. The cost function isn't currently changeable from the configuration file (or command-line), and have to be modified in the actual source code. The overheads should probably be derived from the actual scheduling tree using some sort of heuristic (e.g. dynamic arbitration might require more overhead than TDMA). Additionally, this can't model techniques such as pipelining buses, and clustering or splitting messages. Finally, the assumption of fixed transfer times may be unrealistic because a message between two close nodes may move faster on the bus than one between two distant nodes.

6.3 Future Work

We will now present future directions for this work. We begin with simple, but powerful additions. We then continue with cosmetic and usability changes to the framework. Finally, we address expanding to different optimization techniques, and extending the work for other domains.

The scheduling tree structure is extensible, and fairly natural to work with. Two simple and needed extensions are the addition of round robin scheduling, and token bucket models. These two could be added as new time allocation policies. Constructs should be added for specifying non-trivial overheads, and for specifying custom cost functions. Additionally, the ability to estimate the complexity of a custom cost function needs to be added to make the genetic algorithm's results more meaningful.

The syntax of STRANG was designed to ease parsing and to allow the genetic algorithm easily operate. Unfortunately, the resultant syntax is somewhat difficult to work with. The syntax should be revised to make it easier to work with, and also more flexible. Often bus-traffic directly depends on prior performance. The simulator should be modified so that it is easily interfaced with other simulators, and system-level design environments such as [12]. Additionally it could be integrated with the constraint-driven communication synthesis, protocol synthesis tools such as Ulysses[28], and interface synthesis [24] to create a relatively complete bus-based real-time system design flow.

Right now STRANG only supports one shared resource at a time. It would be interesting to look at expanding STRANG to handle multiple resources, and to explore the system design problem at a broader level. Other work includes synthesizing protocols for the given arbitration schemes, and adding more flexible mode switching mechanisms.

The genetic algorithm is only one technique for exploring the design space, and it tends to be arbitrary and time-consuming. For certain cases it may be possible to exactly solve the problem either analytically, or through a branch and bound search. Also, there's a large opportunity for developing heuristics for more rapidly arriving at a solution.

Finally, STRANG can be applied to more than just buses. It could quite easily be generalized to evaluate scheduling policies in a variety of other domains. Possible domains include: general hardware resources, operating system scheduling, QOS Network Routing, or other scheduling problems. Also, we are considering extending the language for use with other communication primitives such as FIFO's and crossbar switches.

6.4 Acknowledgements

I would like to thank many people for their support and assistance in completing this report. First, I thank my advisor, Professor Alberto Sangiovanni-Vincentelli, for his support, patience, insight, and time. I thank Professor Edward A. Lee for being the second reader and for his helpful suggestions. I thank Claudio Pinello for his help in collaborating on the brainstorming and initial development for this project. Julius Kusuma deserves great mention for his motivating encouragement and proof-reading. I thank Michael Shilman for his careful proofreading, advice, and suggestions. Finally, I thank Gregor Goessler for his proofreading, support, and suggestions for related work.

Bibliography

[1] Tindell, K.; Burns, A.; "Guaranteeing Message Latencies on Controller Area Network (CAN)", Proceedings 1st International CAN Conference, Mainz, Germany, September 1994

[2] Kopetz, H.; "A Comparison of CAN and TTP", Annual Reviews in Control, vol.24, Elsevier, 2000. p.177-88.

[3] Lahiri, K.; Raghunathan, A.; Lakshminarayana, G.; Dey, S.; "Communication Architecture Tuners: a Methodology for the Design of High-Performance Communication Architectures for System-On-Chips" Proceedings 2000. Design Automation Conference. p.513-18.

[4] Yen, T; Wolf W.; "Communication synthesis for distributed embedded systems", Proc. Int. Conf. Computer Aided Design, pp. 288-94, Nov. 1995

[5] Gasteier, M.; Glesner, M.; "Bus-Based Communication Synthesis on System Level.", ACM Transactions on Design Automation of Electronic Systems, vol.4, (no.1), ACM, Jan. 1999. p.1-11.

[6] Ortega, R.; Borriello, G; "Communication Synthesis for Distributed Embedded Systems", Proc. of Int. Conference on Computer Aided Design, June 1998, p. 437-444

[7] Renner, F.; Becker, J.; Glesner, M.; "Automated Communication Synthesis for Architecture-Precise Rapid Prototyping of Real-Time Embedded Systems", Proceedings 11th International Workshop on Rapid System Prototyping, 2000, p.154-9

[8] Holland, J; "Adaptation in Natural and Artificial Systems", Ann Arbor - The University of Michigan Press, 1975

[9] Ebeling, W.; Rose, H.; Schuchhardt, J.; "Evolutionary strategies for solving frustrated problems", IEEE 1994.

[10] Emer, J; Gloy, J; "A Language for Describing Predictors and its Application to Automatic Synthesis", Procedings of the International Symposium on Computer Architecture (ISCA), 1997

[11] Monnier, Y.; Beauvais, J.-P.; Deplanche, A.-M. "A Genetic Algorithm for Scheduling Tasks in a Real-Time Distributed System." Proceedings. 24th EUROMICRO Conference, (vol.2),. p.708-14

[12] The Metropolis Group Web Site: http://www-cad.eecs.berkeley.edu/ polis/metro

[13] ByteFlight Consortium Web Site: http://www.byteflight.com

[14] FlexRay Consortium Web Site: http://www.flexray-group.com

[15] Lee, E.; Sangiovanni-Vincentelli, A.; "A framework for comparing models of computation", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.17, (no.12), IEEE, Dec. 1998. p.1217-29

[16] Nicols, K.; Jacobson, V.; Zang, L.; "A Two-bit Differential Services Architecture for the Internet", Internet-Draft, 1997

[17] Di Natale, M. "Scheduling the CAN bus with earliest deadline techniques", 21st IEEE Real-Time Systems Symposium. IEEE. 2000, pp.259-68.

[18] Isovic D.; Fohler G.; "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints." 21st IEEE Real-Time Systems Symposium. IEEE. 2000, pp.207-16.

[19] Yeh, C.-H.; "Scalable QoS supports for multimedia applications in the next-generation Internet," Proc. IEEE Real Time Technologies and Applications Symp. (IEEE RTAS'01), May/Jun. 2001, pp. 39-50

[20] Abeni, L.; Buttazzo, G.; "Hierarchical QoS management for time sensitive applications.", Proceedings Seventh IEEE Real-Time Technology and Applications Symposium, IEEE Comput. Soc. 2001, pp.63-72

[21] West R.; Poellabauer C.; "Analysis of a window-constrained scheduler for real-time and best-effort packet streams." 21st IEEE Real-Time Systems Symposium. IEEE. 2000, pp.239-48

[22] Kopetz, H.; "A solution to an automotive control system benchmark", Proceedings of 1994 Real-Time Systems Symposium, pp.154-8.

[23] Wang, C-Y;, Hsu, C-C; Huang, Y.; "VORAL: a system for Voice over IP Routing in Application Layer"; Proceedings Seventh IEEE Real-Time Technology and Applications Symposium. IEEE Comput. Soc. 2001, pp.165-70.

[24] Passerone, R.; Rowson, J.; Sangiovanni-Vincentelli, A.; "Automatic Synthesis of Interfaces between Incompatible Protocols, 35th Design Automation Conference, June 1998.

[25] Pinto, A.; Carloni L.; Sangiovanni-Vincentelli, A.; "Constraint-Driven Communication Synthesis", 39th Design Automation Conference, June 2002

[26] Balarin, F.; Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; Passerone, C.; Sangiovanni-Vincentelli, A.; Sentovich, E.; Suzuki, K.; Tabbara, B.; "Hardware-Software Co-Design of Embedded Systems: The POLIS Approach", Kluwer Academic Publishers, Boston/Dordrecht/London, 1997

[27] Balarin, F.; Lavagno, L.; Murthy, P.; Sangiovanni-Vincentelli, A.; "Scheduling for Embedded Real-Time Systems", IEEE Design and Test of Computers, vol. 15 (no. 1), Jan.-March 1998

[28] Sgroi, M.; Sangiovanni-Vincentelli, A.; Berkeley EECS Research Summary, "Ulysses: Methodology and Tools for Protocol Design", http://buffy.eecs.berkeley.edu/IRO/Summary/02abstracts/sgroi.3.html

[29] Pop, T.; Eles, P.; Peng, Z. "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems", Proceedings of the Tenth International Symposium on Hardware/Software Codesign. ACM. 2002, pp.187-92

[30] Kaneko, H; Stankovic, JA; Sen, S; Ramamritham, K "Integrated scheduling of multimedia and hard real-time tasks", Proceedings. 17th IEEE Real-Time Systems Symposium1996, pp.206-17.

[31] The Network Simulator Web Site: http://www.isi.edu/nsnam/ns

[32] Wright, D.J., "Voice Over Packet Networks", John Wiley & Sons Ltd., Chichester, 2001

Appendix A

Running STRANG

A.1 Running The Simulator

There are three ways to run the simulator from the command line. It can be run on a single tree, run with the genetic algorithm that is seeded by one tree, or run by the genetic algorithm with no seeds. Here are the 3 syntaxes:

- **Run the simulator:** gen_bus trace_file config_file tree_file anything
- **Run the genetic algorithm (with a seed):** gen_bus trace_file config_file tree_file
- **Run the genetic algorithm (without a seed):** gen_bus trace_file config_file

A.2 Running The Trace Generator

Format: gen_trace in_file [duration] [random seed]

Appendix B

The STRANG Syntax

B.1 Configuration File

[cycle time] [bandwidth(bits per cycle)] [message overhead (in bits)] [number of senders] [arbitration overhead (in cycles)] [preemption overhead (in cycles)]

B.2 Trace Generation File

```
[# of traces] [# of nodes] [trace#1] [source] [destination]
[size] [period] [deadline] [jitter] [mode] ... [trace#n] [source]
[destination] [size] [period] [deadline] [jitter] [mode]
```

The three modes for timing distribution all include jitter and are:

R - Regular occurrences (plus jitter)

P - Poisson distribution (plus jitter)

F - Flat (Uniform) probabilistic distribution (plus jitter)

B.3 Trace File

version, config info: 2.0 [number of message types] [number of senders]
message syntax: [message type] [sender] [recipient] [arrival time] [deadline] [size]
...

B.4 Tree File

```
(P [PolicyID1] [Operation Function]) ... (P [PolicyIDN] [Operation
Function])
     (A [PolicyID] [Preemption] [Alloc] [#children]
        (durations)
        (child_1)
        (S PolicyID Preemption SndrID)
        ...
        (Arbnode...) (child_#children)
     ...
)
```

Built-in Operation Functions: FIFO, FIXED, EDF, LIFO

Allocation Policies: NONE, ALLOCATE, FLEXALLOCATE, ENFORCE

Preemption Options: NONE, ABORT, SUSPEND, SUSPENDwMEMORY

Figure B.1: Arbitration Tree Syntax

B.4.1 Operation Tree

```
(P [PolicyID_number] [Variable])
(P [PolicyID_number] [Constant])
(P [PolicyID_number] [Operation Left_Child Right_Child])
```

Operations: +, -, *

Variables: arrival, deadline, size, senderID, childID, receiverID, messageID, allocatedTime, sinceArrival, untilDeadline

Constants: floating point numbers.

Figure B.2: Operation Tree Syntax

The operation tree represents the function that is used to describe the different policies used to sort between various messages at a particular node. Each policy is a function of the 11 different variables: arrival-time, deadline, message size, message-type, sender-id, receiver-id, time elapsed since the arrival of the message and time until the deadline of the message. The function can also use floating point constants, and addition, subtraction, and multiplication as operators. Division isn't used because it would be difficult to check divide by zero errors. We use prefix ordering to ease the of parsing the operation functions.

Appendix C

Class Definitions

Here are brief descriptions of each class in the source code for STRANG.

BusGenerator - genetic algorithm (breeding and mutation) and main functions

Simulator - simulates a particular schedule

Bus - The Bus

Sender - a sender in the simulator

Message - a message being sent on the bus, has deadline, preemptability, size, sender, destination, etc.

ArbitrationTree - the arbitration tree

ArbitrationTreeNode - base class for arbitration tree nodes

ArbitrationNode - arbitration node (has 2 or more children)

SenderNode - sender node, orders messages on the sender.

OperationTree - tree for calculating fitness at a given arbitration tree node

OperationTreeNode - node in operation tree for calculating a given fitness

OperatorNode - +, -, *, and 2+ children

VariableNode - arrival, deadline, size, senderID, childID, receiverID, messageID, allocatedTime, sinceArrival, untilDeadline

ConstantNode - A floating point constant value

Heap - heap for storing the priorities of the different messages(aka the event queue)

Appendix D

File Descriptions

D.1 Config Files

config - one sender

config2,3,4,configFIFO,configTDMA - three senders (config2 and config4 are identical)

configCAN1,2 - CAN bus configurations (7 and 17 senders respectively)

configTTP1,2 - TTP configurations (same as CAN configurations)

config_sae_can - Configuration for the SAE automotive example for CAN bus

config_sae_ttp - Configuration of the SAE automotive example for TTP and shared TTP buses

config_voip - Configuration for the voice over ip example.

D.2 Tree Files

tree - just has one node

tree2, treeFIFO - 3 nodes pure fifo

tree2p - 3 nodes, fixed priority top level, other goodies, such as abort preemption.

tree3,4,5,treeRcv,treeSize - 3 nodes different priority schemes.

treep - one tree with ABORT and SUSPEND preemption

treeCAN1,2 - 7 (and 17) nodes, EDF scheduling

treeEDF - 3 nodes, earliest deadline first, suspend preemption.

treeEnforce - 3 nodes, fifo, enforces sizes

treeFTDMA - 3 nodes Flexibile allocation of time

treeLIFO - 3 nodes, lifo everywhere

treeTDMA - 3 nodes FIFO arbitration with TDMA

treeTDMA7 - TDMA with 7 nodes

treeTDMAEDF - TDMA with 3 nodes, EDF instead of FIFO at the senders

treeTTP1,2 - TDMA with 7 and 17 nodes respectively

tree_sae_CUSTOM - SAE with FIFO and 1 level of additional hierarchy.

tree_sae_FIFO - SAE with FIFO, 1 level of hierarchy

tree_sae_FIXED - SAE with fixed priority

tree_sae_FTDMA - SAE with flexible TDMA

tree_sae_TDMA - SAE with TDMA

tree_sae_CAN - SAE with CAN

tree_sae_CEDF - SAE with CAN implementing EDF

tree_sae_TTP - SAE with TTP

tree_sae_TTPx - SAE with shared TTP

tree_voip_edf - Tree for VoIP example with EDF

tree_voip_fifo - Tree for VoIP example with FIFO

tree_voip_fixed - Tree for VoIP example with RMS fixed ordering.

tree_voip_fixed71 - Tree for VoIP example with DMS fixed ordering.

D.3 Trace Files

custom.tr, simple.tr - 3 senders, 10 types

simpler.tr, simplest.tr - 1 sender, 3 types

sae_short.tr, sae_shortest.tr, sae_simp.tr, sae_simp_short_ln.tr - 17 types, with poisson distributions

sae_full.tr - 53 types, messages with poisson distributions

sae_periodic_full.tr - 53 types, messages are all periodic

sae_periodic_simp.tr - 17 types, messages are all periodic

voip.tr - The trace used by the VoIP example.