Genetic Algorithms for Bus Scheduler Generation
CS252 Final Project Report
Trevor Meyerowitz + Claudio Pinello
{tcm, pinello}@eecs.berkeley.edu

Abstract:
This project attempts scheduling messages with deadlines at maximal efficiency onto a single bus. STRANG (Scheduling Tree language) is a new language for the hierarchical description of bus arbitration policies. It effectively expresses both time-sliced and pure priority driven schemes in a natural framework. Additionally it supports preemption and the generation of arbitrarily complicated priority trees. We have developed a trace-driven simulator that allows you to evaluate different policies specified in STRANG. We also present a genetic algorithm to automatically create schedules that are well suited for a given system.

1. Introduction:
Systems in today’s world are becoming increasingly more complicated. There are many designs that have hard or soft real-time constraints. The traditional example of this is an automotive control network. It must be completely robust and guarantee adherence to the hard real-time deadlines of a system. There are also emerging systems with soft real-time deadlines. Some possibilities are wireless system or a QOS (Quality of Service) networking applications such as streaming video. Whatever the system may be, there exists a need to interconnect their components. Given the increasing complexity of hardware designs, the growing number real-time applications, and the expanding performance gap between processors and their I/O speed, techniques for achieving efficient bus utilization are critical. Not only must the bus be of sufficient bandwidth, but it must also satisfy the deadlines real-time message. Each message in the system has a variety of attributes such as arrival time, deadline, size, and others. These can be combined to generate a priority function. Given a sample message trace, a description of the configuration of the system and a performance metric, different scheduling policies can be compared. The structure of the best schedule largely depends on the complexity and regularity of the traces as well as the overall structure of the system. Near-optimal performance should be achievable through an intelligent exploration of the design space. With the system we have developed we can: hierarchically describe bus schedules using STRANG, generate message traces based on statistical characterizations, simulate the bus schedules on these traces, and explore the design space using a genetic algorithm that modifies the scheduling tree.

1.1 Previous Work:
Our initial inspiration for this work was the structure of safety-critical automotive control protocols. The CAN (Control Area Network) bus [1] is a non-preemptive fixed priority bus with dynamic arbitration. Although not originally intended for safety-critical systems, CAN’s performance can be proven when bounds are placed on the frequencies of the messages. TTP (Time Triggered Protocol) [2] ensures reliability by statically reserving a unique time slot for every message. This is an excellent choice for completely static systems or where safety is of paramount concern, but it is very restrictive and performs on systems with a large number of sporadic messages. The latest automotive protocols Byteflight[3] and Flexray[4] improve upon TTP by splitting the system into dynamic and static portions. This is helpful, but provides little guarantee of performance of the non-static portion. We like this idea of dividing the protocol, but it is too limited for the wide spectrum of schedules that we wish to explore. We extend this flat division into a hierarchical protocol tree that allows

Dey and others introduce CAT’s (Communication Architecture Tuners) in [5]. They show that dynamic priorities can meet certain deadlines that are impossible to satisfy using fixed priorities. They present an example of an network controller that meet all of its deadlines if scheduled with a fixed priority and make all the deadlines if dynamically scheduled. In this example the priority is set to the message size times the deadline. This idea of creating a priority function based on a function of message characteristics is very powerful. This system also provides the mechanism for creating hardware FSM’s that implement the chosen protocol. This work is very interesting, but it requires some manual intervention and iteration. We feel that designers aren’t able to consider the entire design space and so are in need of automation. Other systems such as [6] and [7] look at the top level system topology issues such as bus sizes and organizations. We are concerned with the optimization of a single bus based on a given trace of tasks.
Genetic algorithms (GA's) introduced in [8] provide a reasonable way to explore a large search space filled with many non-obvious solutions. In [9] Emer and Gloy show the applicability of genetic algorithms to designing branch predictors. In [10] the authors use GA's schedule tasks in a real-time distributed system. Their approach treats the buses from a higher level as part of the entire system. We look at the actual techniques for scheduling the bus, for a given topology and trace.

1.2 Outline:
In section 2 we will explain the STRANG language and provide several examples of how it can be used to describe the protocols examined in the prior section. In section 3 we outline the characteristics of our genetic algorithm. In section 4 we discuss our results and the potential of the system. Finally in section 5 we draw conclusions from this project and lay out plans for future work.

2. Model of the Protocol

We aim at generating a tree of schedulers defining how messages should be treated at every communication cycle. Although the cycle of schedulers would repeat itself over and over, some of the schedulers may be dynamic. Moreover since we are optimizing the scheduler on a trace that contains a dynamic set of messages, this should result in schedule trees that can handle periodic messages as well as sporadic ones.

The scheduler tree is composed of basic arbitration nodes each of them arbitrating the bandwidth among a set of children, each of which can be either an Arbitration Node or a Sender Arbiter Node.

A sender node is associated with a specific sender and selects a message in the sender's buffer according to some policy. So the root node is always an Arbitration Node and the leaves are always Sender Arbiter Nodes.

Both Sender Arbitration Nodes and Arbitration Nodes use policies based on minimizing priority functions like:

- FIFO
- Earliest deadline first
- Best fit
- Shortest message first
- etc.

In addition to this, arbitration nodes may also have a notion of time so that we can model time-slicing protocols.

Arbitration Node Parameters:

The basic parameters that we identified for describing the scheduler tree nodes are:

policytree: a pointer to the tree that describes the priority function used to compare children; e.g. arrival time for FCFS, deadline for EDF, message size for shortest message first, combinations of such basic functions.

preemption_style: select whether a child with higher priority can preempt the current child; either NONE, ABORT, SUSPEND.

time_slots: a vector of floats defining the time slots, i.e. the maximum allotted time for every child. The real use of these time slots is described by the following parameter.

time_allocate: this parameter specifies how to use the time slots durations. It can either be NO, ENFORCE, ALLOCATE or FLEXALLOCATE. NO simply ignores them, the only duration constraint is the one coming from the arbitration node above if any.
ENFORCE uses them to determine the maximum time that can be used by a child but does not allocate time if the child does not have a message to send. Moreover all children are considered for transmission at every decision point, i.e. this is not a time-slicing arbiter hence it does not keep track of the ordering of its children nodes based on a cyclic schedule.

ALLOCATE assigns control to the specified child for the entire duration of the time slot even when the child has no message to send. This is the stricter time-sliced arbitration policy because at every one time there is only one possible child that is available for arbitration. This also means that the priority function of the arbiter is not used because there is only one child to choose from.

FLEXALLOCATE assigns control to the specified child for the entire duration of the time slot only when the child has a message to send. If the corresponding child does not have a message by the beginning of its time-slot, it must wait for the next round. Usually in this type of arbitration the child is given a minimum time to start transmitting (of the order of one bus cycle), much like a reservation slot, after which it lost the right to transmit. Also in this case the priority function is not useful because there is only one child at a time to choose from.

vectorofchildren: a list of children nodes; each child can be either an Arbitration Node or a Sender Arbiter Node (leaf).

2.1 Sender Arbiter Node Parameters:

policytree: a pointer to the tree that describes the priority function used to compare messages in the buffer of the associated sender node; e.g. arrival time for FIFO, deadline for EDF, any linear combination of such basic functions.

preemption_style: selects whether a message with higher priority can preempt the current message from the same sender; either NONE, ABORT, SUSPEND.

senderID: determines which sender it is attached to.

2.2 STRANG Language

A scheduler is the forest containing an arbitration tree and some policy trees. The tree structure can be described using the following language.

The root node must be an Arbitration Node. An Arbitration Node is described as

(A policytreeID preemption_style time_allocate #children
   (timeslot0 ... timeslot#children-1)
   child0
   ...
   child#children-1
)

Each child can either be another Arbitration Node or a Sender Arbiter Node.

A Sender Arbiter Node is a leaf of the tree structure and is described as

(S policytreeID preemption_style senderID)

A policytree is described as

(P policytreeID
   bintree
)
A bintree is a binary tree where non-leaf nodes (Operation Nodes) represent operators and leaf nodes (Terminal Nodes) represent operands. It operates on messages to assign a priority.

An Operation Node is described as

(operation
   operandLeft
   operandRight
)

an operand can be either another Operation Node or a Terminal Node.

A Terminal Node can be either: arrival_time, message_size, deadline, senderID, childID, allocated_time, messageID or a constant value.

An operation can be either: +, -, *

Finally a scheduler forest can be described as

NPT
[policytree]
Arbitration Node

where NPT is an integer indicating the number of policy trees in the forest.

Scheduler Examples
In this section we give a few examples of the kind of scheduling policies that can be expressed by this language.
In example a non-preemptive FCFS scheduler with two senders can be expressed like

1
  (P FIFO arrival_time)
  (A FIFO NONE NO 2
    (0 0)
    (S FIFO NO sender1)
    (S FIFO NO sender1)
  )

Similarly a TDMA scheduler would be expressed like

2
  (P FIFO arrival_time )
  (P FIXED childID)
  (A FIXED NONE ALLOCATE 2
    (0.001 0.002)
    (S FIFO NO sender1)
    (S FIFO NO sender1)
  )

Built-in Libraries
Some of the policy trees are so commonly used that we decided to allow the user to refer to them without explicitly defining the tree. We identified the following priority tree as worthwhile including.

(P FIFO arrival_time)
(P FIXED childID)
(P EDF deadline)
(P SHORT size)
(P BEST (+ 0 size))

Note that BEST (for BEST-FIT) is really computing -size thus aiming at the largest possible message. However the priority function (tree) is applied only to messages that are shorter that the allocated time hence that have a right to be transmitted.

The list can be obviously extended by using more complex trees and/or defining new attributes for the messages.

Results/deliverables

As partial results of this project we have the following deliverables:

- Definition of a language for describing hierarchical schedulers in a compact and modular way.
- A trace generator.
- A discrete-event simulator for the bus+scheduler tree.
- A partially operational Genetic Algorithm to explore the design space automatically.

A description of the STRANG language is offered in a previous section. The trace generator reads-in a file describing the message types in terms of their statistical characterization, like arrival time distribution, periodicity, jitter variance, inter-arrival time. It also takes information like sender and receiver identifiers and deadlines. The result is a trace that is a possible execution of the described bus system. For its nature we cannot simulate or use dynamic bus traffic with this setup, because message arrivals to the bus system do not depend on the arrival/delivery of previous messages. To have a dynamic trace we need a dynamic model of the senders/receivers instead of their simple statistical characterization.

The discrete-event simulator\(^1\) first parses in a scheduling tree written in the STRANG language and then reads-in a trace file. Finally the traffic on the bus is arbitrated using the scheduling tree and statistics are collected to measure the fitness of the tree. This simulator can be used separately from the genetic algorithm to have a fast evaluation of the fitness of complicated scheduling policies. The combination of the STRANG language and of its simulator makes the “manual” exploration of the design space much faster, reducing the burden of describing your own policy in general-purpose programming languages.

Unfortunately we haven’t been able to fix some segmentation faults caused by the genetic algorithm. In particular we can only explore changes in a sub-set of the parameters and topologies of the scheduling tree.

Genetic Algorithm

We developed a partially operational genetic algorithm to explore the design space of the scheduling trees.

Brief introduction to GA’s

The basic idea behind this type of optimization search is the evolutionary theory that so accurately explains the diversity of successful life forms in nature. It is based on the notion of genotype, a coded description of an individual (phenotype), that can be varied to obtain new individuals. The genome is the space of the genotypes for a given species. Success of an individual to survive its environment is usually measured by a fitness function. The most fit individuals tend to have higher chances of surviving and especially of breeding, thus perpetuating a large part of their genotype to the offspring. In particular the offspring is a mix (cross-over) of the genotypes of the parents. Other times it is convenient to introduce random “mutations” to the genotype so that the new individual may get properties that are not found in any of the parents.

Genetic algorithms have a lot in common with other optimization searches like simulated annealing[8ebeling93]. One of the differences is that genetic algorithms tend to be less sensitive to local

---

\(^1\) For the readers that were at the presentation on Tuesday: the simulator was actually working fine also on long traces but we were using a bad-formatted trace that was using more senders than declared. There is a check now to avoid seg-faults in these cases and provide more informative responses instead.
minima in the cost function because the function from the parameters space to the fitness space is usually strongly non continuous and discrete variations in the genotype (like those coming from a cross-over or mutation) can often bring the system away of a local minima. This behavior at the same time makes the convergence of genetic algorithms less than obvious for many systems.

3. Applying Genetic Programming to the scheduler synthesis

Our genome is the set of trees that we can be described in the STRANG language. We first generate a random population (200 individuals in our examples) and then evaluate their fitness using the simulator as illustrated in Figure GA. We then produce a new generation of trees favoring re-use of the genotypes of the most successful individuals. At the same time we want to give a chance to all individuals to participate in future generations, so to increase diversity and avoid a homogeneous set of good individuals that are incapable of generating interesting new individuals. The mechanisms that we use to produce the next generation are then

- Survival;
- Mutation;
- Breeding;

We implemented survival by “copying” the five best-fit individuals to the next generation, as illustrated in Figure Survival.

We implemented a variety of mutation mechanisms, all occur in two phases: first a copy of an individual is selected at random among the best 20 individuals, then it is modified and stored in the new generation. The different mutation mechanisms that we have coded are:

- Select a node and modify its properties (parameters of the STRANG language)
• Select a node and collapse all the corresponding sub-tree, like in Figure Prune.
• Select a node, add an Arbitration Node child and move some of the children to the new Arbitration Node, like in figure InsertNode.
• Select a node and add a random sub-tree, like in Figure Add Tree.
• Select a node and swap two of its children.
• Select a node, kill one of its child and then adopt the grandchildren.

The other very important mechanism is breeding and we implemented it by crossover, i.e. we select one node from the first parent and one node from the second parent and then exchange the two sub-trees to obtain two offsprings, like in Figure Crossover.
These mechanisms are used to generate a total of 200 individuals starting from the best 20 individuals of the previous generation. The entire process is repeated a number of times until the improvement in the fitness of the best individuals is zero or very small. The resulting best individuals are supposedly able to schedule messages from traces that have similar characteristics as the one used to select them. In evolutionary terms these individuals are likely to be well-fit to other environments similar to the one where they evolved.

It is important to notice that most of these mechanisms can be applied to the policy trees as well, so that we can mutate a policy tree or cross-over two distinct policy trees. We have implemented also these mechanisms in the C++ code for the genetic algorithm.

4. Benchmarks

We selected a benchmark from the SAE (Society of Automotive Engineers) that describes an electric vehicle control system.

The system consists of 7 nodes connected by a bus. They exchange a set of 53 message types with different periodicities and deadlines (some are defined to be sporadic). This test bench has been used to evaluate both a solution based on a contention based protocol (CAN controller area network) and a solution based on a time-sliced protocol (TTP time triggered protocol). We plan on using their results as a baseline for comparison of our solution.

We used the statistical characterization provided by the SAE to run our trace generator.

4.1 Results

We used the SAE trace to measure the throughput of a can-like hand written scheduler and measured a value of 1.46 messages per millisecond.

We also measured throughput for a genetically improved version of can and obtained 2.7.

The solution used a fixed priority instead of EDF.

And for a mutation not seeded w/ can we got a version with a fitness of 2.88.

It used size as the priority metric.

5. Conclusions and Future Work:

Although our project isn’t fully completed we are very close to having a fully functional system. Because of the few remaining bugs it is difficult to say what the potential for improvement is using this system. We have shown that the STRANG language can naturally describe a wide variety of bus scheduling policies. The simulator is stable enough to test a wide variety of functions. The genetic algorithm is completely coded, but it has a few key bugs remaining that currently limit our use of it.

5.1 Conclusions:

This project proved to be an incredibly interesting endeavor. Our research indicates that it is a unique approach, which we believe has much promise. The two major successes are the STRANG language and the simulator. We have shown the language to be easily understood and powerful enough to describe
most scheduling systems. Our genetic algorithm can function if only operation tree mutations and breedings are applied. Even with this we did achieve some gains, but the true power of this approach remains unclear. Even if the genetic algorithm isn’t powerful, the framework is in place to design and test bus arbitration schedules. This in itself is a noteworthy achievement.

All in all this was a very rewarding project. It required us to think deeply about what we wanted to accomplish, what made sense, and what we could do in the allotted time. We read a large number of papers to investigate the area, and believe that this is something novel and useful. Additionally we have a greater appreciation for event driven simulators and genetic algorithms having written one of each for this project.

PS: We should’ve run more simulations instead of trying to fix the bugs that appeared at the last minute.

5.2 Future Work:

Our work, while significant, has just scratched the surface of this area. We plan on debugging the genetic algorithm and then running many different tests upon it to determine the best parameters to use. Also currently there is no restrictions upon the algorithm to take into account hardware complexity. Such metrics could be bounding the number of tree entries, the sizes of operation trees, or incorporating hardware complexity as a cost metric. Additionally we should explore additional fitness metrics and making cost functions more customizable. We do believe that we have a relatively complete genetic algorithm for this system, but more debugging and testing is necessary.

The simulator and the STRANG language are relatively complete. STRANG should probably be revised to be more readable by allowing operation trees of arbitrary names and expressing functions in natural form instead of prefix notation. Also, currently the simulator doesn’t support preemption on sender nodes and its code should be cleaned up and optimized more. Another limitation is the fact that the traces that we currently use are static and don’t depend upon the performance of the bus. This limits the sorts of traces that we can meaningfully investigate. Given the object-oriented design of our simulator, it shouldn’t be difficult to hook it up to computational components that generate traffic based on the performance of the bus.

We would also like to use our simulator and the STRANG language as a technique for investigating more formal methods of determining schedules. One of the nice features of this project is that it has many facets and so if one part isn’t successful, there are others to build off of. This system provides a nice intermediate level of abstraction between the top-level and very general system view of buses and the wire level view of buses.
References:


