CODE GENERATION FOR MULTIRATE DSP SYSTEMS IN GABRIEL

by

Stephen How

Memorandum No. UCB/ERL M94/82

26 October 1994

COVER PAGE

CODE GENERATION FOR MULTIRATE DSP SYSTEMS IN GABRIEL

by

Stephen How

Memorandum No. UCB/ERL M94/82

26 October 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CODE GENERATION FOR MULTIRATE DSP SYSTEMS IN GABRIEL

by

Stephen How

Memorandum No. UCB/ERL M94/82

26 October 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

CODE GENERATION FOR MULTIRATE DSP SYSTEMS IN GABRIEL

Stephen How

Master's Degree Report
UC Berkeley

ABSTRACT

Gabriel's code generator allows for rapid, real-time prototyping of DSP systems described as Syncronous Data Flow graphs. The static scheduling of the SDF graphs allows for efficient buffering between actors and a simple in-line code compilation technique. Multirate SDF graphs presented a problem for in-line coding because of redundancy. A technique for code generation of multirate systems is developed using a specialized scheduler, iteration, and a modified static buffering scheme. A 2400bps voiceband data channel is developed as an example for the multirate code generator.

1. Introduction

Gabriel's capacity to automatically generate assembly code for DSP systems described as syncronous data flow graphs makes it very powerful as a fast-prototyping system. In SDF graphs, actors representing DSP functions consume and produce a fixed number of samples on the data arcs connecting them. The SDF model facilitates a systematic method of scheduling the actors (DSP code) on the processors of the DSP hardware [1]. Because of the syncronous nature of the data flow model, DSP code for the entire graph may be compiled together and downloaded into the DSP program memory. The code Gabriel produces is an in-line placement of the actor codeblocks in the sequence of the schedule [2]. This implementation uses a low overhead *static buffering* scheme which provides good runtime efficiency. However, in-line code can be very inefficient in terms of program length, especially for DSP systems with several sample frequencies. Many DSP applications are multirate, with unirate subsystems producing or processing larger blocks of data. These systems are easily described to Gabriel in an SDF graph, but the generated in-line code may be impractically long.

This report documents a method of producing efficient *iterated in-line code* for multirate graphs presented to Gabriel. This method retains in-line code and static buffering for runtime efficiency, and places loops around repetitive sequences of the schedule. Arbitrary graphs are handled, and an attempt is made to organize loops to minimize runtime overhead. The system is currently only implemented for single processor code generation, although a simple version for the multiprocessor case may be modified from the existing code. This multirate capability is implemented as a set of additions to the Gabriel code, rather than modifications to it. These functions required to implement iteration are arranged as a preprocessor to the original code generator as shown in the block diagram of figure 1. The organization of the report roughly follows the sequence of the preprocessor after an initial description of iterated in-line code.

As a typical example of a real-time application using several sampling rates, a 2400bps voiceband data channel was developed on Gabriel and implemented on a

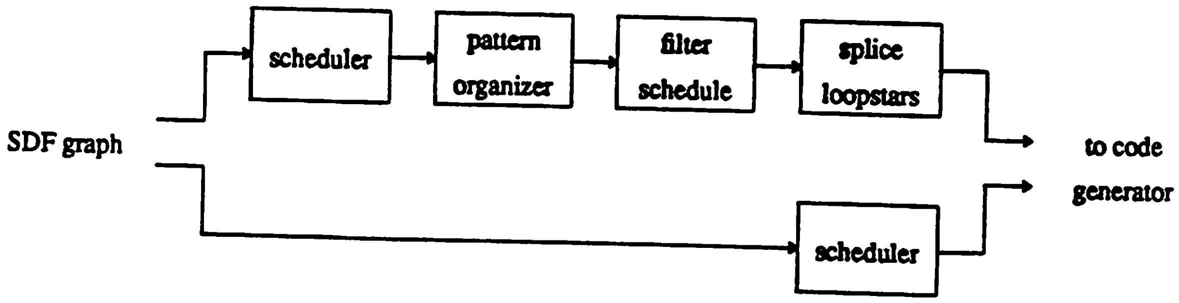


Figure 1. Code generation flow diagram. The top path includes pre-processing for the coding of multirate graphs. The bottom path is the original code generation method.

DSP56000 processor with a memory mapped A/D converter. An algorithm for baud rate timing recovery was developed, and has a geometeric interpretation described in the appendix.

2. Problems of In-line Code

This section first reviews the techniques of static buffering and in-line code originally used by Gabriel. The shortcomings of this approach are shown for multirate SDF graphs and, as an alternate form of generated code, iterated in-line code is proposed. The details of its implementation are described in the following sections.

2.1. In-Line Code

Gabriel uses an in-line technique for code generation. From the SDF universe, where each star produces and consumes a known number of tokens per firing, a schedule of star firings is constructed. The schedule is then directly translated into code by subsituting instances of the stars with their respective code templates.

Code templates are the blocks of DSP assembly code that perform the function of a star. Typically, stars consume (input) and produce (output) samples on the data arcs using symbolic absolute memory references:

```
move  x:in,a
{body of code}
move  a,x:out
```

When code is being generated by Gabriel, the absolute addresses are computed and subsituted for "in" and "out" in each instance of the star.

The data buffering between stars can be handled using absolute memory addressing due to the static buffering technique, a benefit of the SDF model. A static buffer representing a data arc is constrained to a have a certain length, so that each seperate invocation of a star references the same buffer locations. In the example of figure 2, the arc is implemented as a static buffer of length 6. At least 3 locations are needed to hold the samples written by A before they are read by B. The buffer is shown after AB. Note at the end of the schedule, after ABABB, the pointers return to their initial positions. In general, the length of a static buffer, L, must divide into the total number of samples produced on that arc in one period, N, an integral number of times. This is

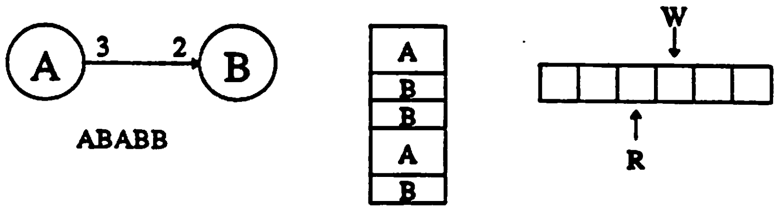


Figure 2. A simple SDF graph, its schedule, in-line code, and static buffer. The pointers are shown after one firing of A and B.

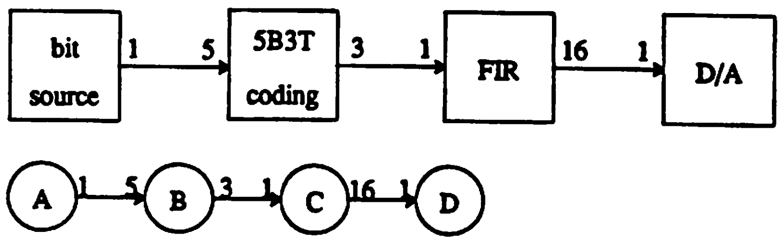
the number of times the r/w pointers wrap around the buffer in one period of the schedule. From this set of possible L's, the smallest length larger than the minimum required is usually chosen.

2.2. Iterated In-Line Code

In-line code is a simple, run-time efficient way to implement an SDF universe. Since buffer management is handled at compile-time, the only overhead is the copying of samples from the data buffer to an arithmethic register. However, because in-line code is simply a direct translation of the schedule, a long, repetitive schedule will result in long, redundant code. This is generally the case for systems with interpolation, decimation, or other sample rate changes. For example, nested sample rate changes are common, as in the simple graph of figure 3 which represents a PAM transmitter with redundancy coding. A directly coded schedule would result in 48 repetitions of the D/A star codeblock, and 12 instances of the FIR. A similar transmitter, without coding, required about one-half of the DSP's program space in the lab's test setup. Any coding of the bit stream would have produced code exceeding the available program memory. It's easy to imagine other SDF examples with high sample rate ratios involving complicated stars producing prohibitively long code (e.g. subband coding using FFT's).

A solution was sought that would work with Gabriel's existing code generator for simplicity, and also for the efficiency of the static buffering scheme. Many methods for just reducing the program space requirement can be imagined where the star's codeblocks reside in memory as subroutines, and the main program would consist of calls following the schedule. Buffer pointers could be computed at compile-time and passed in the subroutine calls, or buffers could maintain their own r/w pointers during runtime. However, the overhead required of these schemes is expensive in real-time applications, especially in fine-grain SDF programming.

In the modified code generation technique, repetitions of sequences in the schedule are coded as software loops around in-line code. First, a schedule is put into abbreviated form in which repeated patterns are replaced by a period of the pattern and its repetition factor. This form is called the iterated form of the schedule. In



AAAAABCDDDDDDDDDDDDDDDDCDDDDDDDDDDDDDDDDCDDDDDDDDDDDDDDDD

Figure 3. A PAM transmitter as an example of a graph with multiple sample rates. Its long schedule is representative of its in-line code.

figure 3 for example, the flat schedule is abbreviated as (5A)B(3C(16D)). As in-line code looks very much like its schedule, iterated in-line code will look very much like the iterated schedule with the parenthesis subsituted with code implementing the loops. Most importantly, the code will be of a manageable size, and will allow for previously impractical multirate applications to be implemented.

3. Scheduling

The first problem in generating efficient code for a multirate universe is to obtain a compact, iterated form of the schedule. Any valid schedule can be grouped into an iterated form, but since loops require runtime overhead, it is desireable to minimize the number of loops, and maximize their iteration count. In general, it is more desireable to code a compact, orderly form of a schedule than a long, scattered one. In addition to being more efficient in runtime and storage, the code is more readable when in iterated form.

3.1. Minimum Buffer Size Scheduling

Initially, Gabriel's existing scheduler was used in the multirate code generation path in figure 1. This scheduler was concerned only with keeping buffer sizes minimized, and the resultant iterated schedules were very inefficient. To minimize buffer sizes, the scheduling algorithm preferentially fires stars whose output buffers contain less than the number of samples required by its dependents. A fireable star that would produce excess samples on these arcs is scheduled only if it is the only fireable star. In this manner, an attempt is made to data arcs are kept as small as possible. However, the schedules produced by this algorithm are not satifactory for general multirate systems. Consider the multirate graph in figure 4. A compact, orderly form for the iterated schedule might be HI(2(5ABCD)EFG). However, the old scheduler produces ABCFHIABDACD(2BCDA)BCDEGF(5ABCD)EG. The main fault of this scheduler is its inability to recognize natural flow in the graph. The path ABCD should be treated as a whole, where no star should fire unless the entire sequence can be scheduled. Because of this, the old scheduler tends to stagger samples in the data path with sequences like ABCABAF ... and then ... DCDBCDABCD ... . This problem is seen to be even more severe in a practical graph representing a QAM data transmitter. The scheduler produces an iterated schedule that is still 2/3 as long as the flat version. By comparison, a hand-scheduled iterative form would be about 12 times shorter than the flat schedule.

3.2. Multirate Scheduling

A new scheduling algorithm was developed that generates a highly ordered, flat schedule by recognizing the overall flow of the multirate graph. A pattern organizer later groups this schedule into a compact iterated form. This algorithm is described using the examples of figure 4 and figure 5, and a general argument for its effectiveness is given.

First, the scheduler partitions the multirate sdf graph into groups of *connected stars of the same frequency*. The frequencies of the stars are calculated by Gabriel at connect-time, and represent the number of times a star fires in a period of the

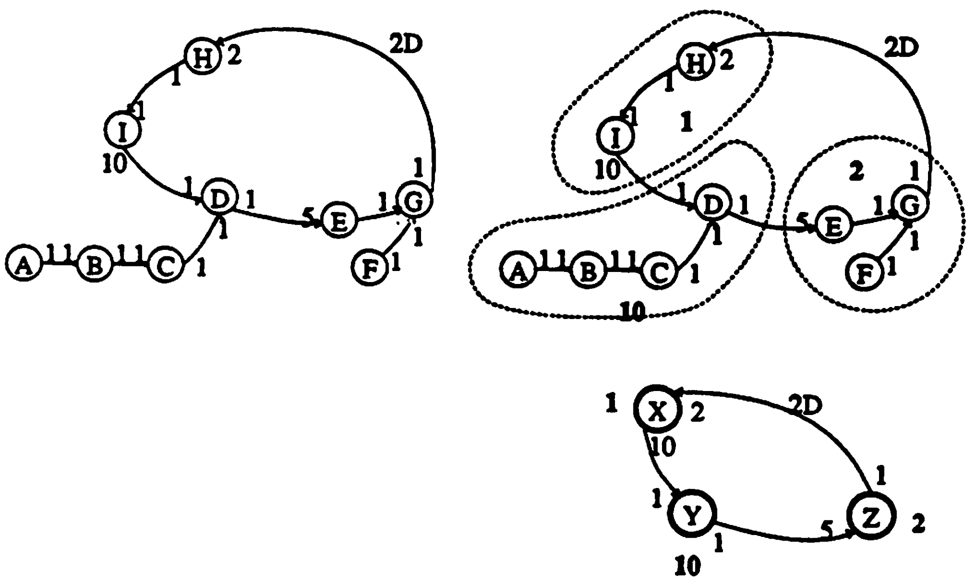


Figure 4. Multirate SDF graph. The graph grouped into connected stars of the same frequency. 2D stands for two delays on the arc.

schedule. Consider the example in figure 4. The groups are identified by a dotted encircling and their sample rate is noted inside in bold. Assuming the multirate graph has consistent sample rates (i.e. a schedule exists), the groups partition the sdf graph into a simplified universe where the groups are galaxy-like structures. Most importantly, because these groups consist of connected stars of the same frequency, only one firing of each star in the group is required to fire the quasi-galaxy. Furthermore, an ordering of these stars exists so that the buffer sizes for the internal arcs are not increased. This one-pass schedule of the group stars is valid whenever the quasi-galaxy is fireable, and will be referred to as a sub-schedule.

Then, if a multirate graph is partitioned, and a schedule exists for the simplified graph, sub-schedules for the groups may be subsituted into the simplified schedule. In figure 4, the simplified schedule is X(2(5Y)Z) and subsituting ABCD for Y, HI for X, and EFG for Z, the compact schedule of HI(2(5ABCD)EFG) is obtained.

A more complicated situation exists when the simplified graph is initially deadlocked. This can occur for universes that are initially schedulable, but after partitioning, the delays that provide starting points become internal to the quasi-galaxies. For example, in figure 4, if the two delays "2D" were removed and 1D was placed on the arc between H and I, a schedule would still be exist for the graph. However, on the level of the simplified graph, the 2D delays are gone and the graph is deadlocked. Assuming a schedule exists for the actual graph, the scheduler must find a set of stars to fire to break the deadlock. There are fireable stars within the groups, however if they don't help break the deadlock, then they only disorganize the sequence. Firing a star without firing the rest of the stars in its group means that later the rest of the group will fire without it. This tends to make the iterated schedule less compact, and decrease the efficiency of the code. The result would be similar to the original scheduler's output. In the modified, deadlocked example of figure 4, initially stars A, I, and F are fireable. It's clear that A and F are fireable stars that do not help to break deadlock, while firing I immediately breaks the deadlock. In general, the scheduler needs to "initialize" the graph by firing sequences inside groups that produce samples on their output arcs. After the delays are thus pushed out, scheduling of the groups can continue.

Of course, there are many other examples of simplified graphs that are initially deadlocked, and cannot be "initialized" as above. Consider the graph in figure 5 which is initially deadlocked in its simplified form. By searching for sequences that produce samples on quasi-galaxy output arcs, AB is identified. Notice that even if AB is scheduled, the same condition will exist for the next 4 schedulings of AB. Meanwhile, the arc between A and F is unnecessarily accumulating samples. We are forced to search for deadlock-breaking sequences at each top-level scheduling step.

At this point, a general solution to the deadlock problem would be to break the group containing the required delay or source into separate groups. A new group is formed that contains only the deadlock breaking sequence. The rest of the stars in the original quasi-galaxy are re-partitioned into groups of connected stars. This is done in figure 5, where the rate 10 group was re-partitioned after removing AB. The result being a simplified, schedulable graph. Also note that in the resulting schedule (2(5ABF)D)(10C) is more efficient because the buffering between A and F is minimized. The general algorithm for the multirate scheduler is outlined in below:

```
partition graph
loop
  (while not_deadlocked
        schedule_groups)
  (if    done, return
   elseif no_fireable_stars, error)
   else   re-partiton_group)
```

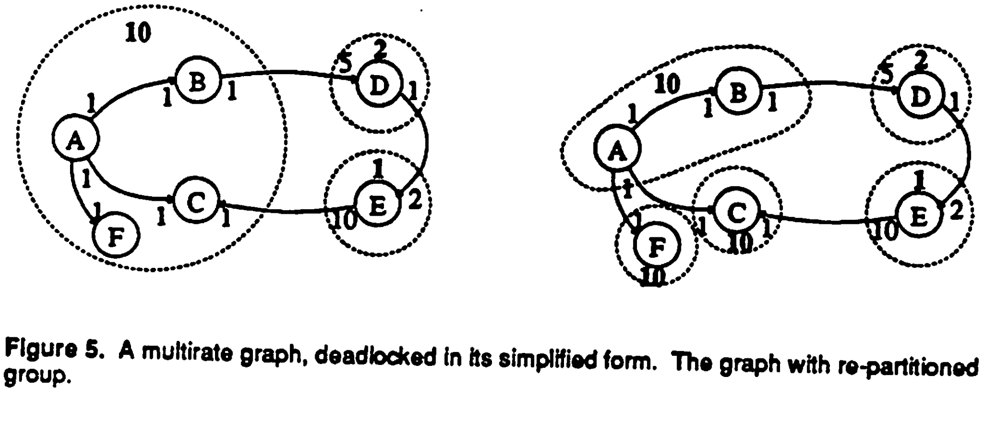


Figure 5. A multirate graph, deadlocked in its simplified form. The graph with re-partitioned group.

go loop

3.3. Top Level Scheduling

It has been assumed that the algorihm above produces a schedule with a high degree of order. Order is actually more dependent upon the top level scheduler than the group sub-scheduler, since the fixed sub-schedules are just subsituted for the group instances. Consider the simplified graph of figure 6. We expect a schedule of (7Z)(5Y)(3X)A although XYZXYZXYZYZYZZZA or some worse schedule would also meet the minimum buffer size criterion. Note that if the scheduler checks for fireable groups in a fixed order, this problem is avoided. Currently, the scheduler sequences through a list of groups arranged in descending order of frequency, checking if a group is preferentially fireable. After scheduling a group, the search restarts at the beginning of the list. In the case when a group is broken apart, the new groups are placed in the list in the position of the previous group.

3.4. Organizing Iteration in the Schedule

The product of the multirate scheduler is a flat schedule which contains repetitive sequences of stars. The repetition is a product of the sample rate relationships of the simplified, partitioned graph. The last scheduling step is to reduce the flat schedule to the compact iterated form. This task is merely one of pattern matching, and doesn't require any information from the graph. However, in the next section on buffering, the iterated schedule will again be interpreted in terms of the graph.

The algorithm for pattern orgainization maximizes the degree of iteration in terms of minimizing the length of the output schedule. This criterion is implemented as a preference to represent ABABABABAB as (4AB) rather than (2ABAB), for instance. The basis for the algorithm is a function that finds repetition in a sequence, which is called recursively to form nested loops. The algorithm is straightforward, and the details can be found in the code.

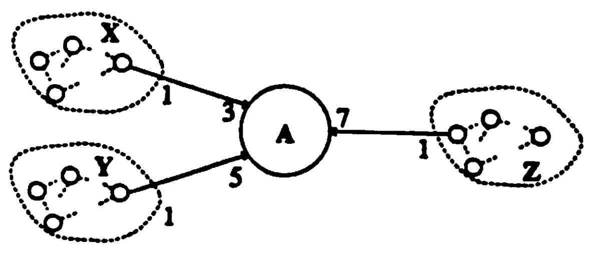


Figure 6. A simplified graph of a multirate universe. The scheduling of the galaxies defines the compactness of the iterated schedule.

4. Code Generation

The previous section describes how information from the SDF graph is used to produce a compact schedule for a multirate universe. This section covers the problems of adapting static buffering to iterated in-line code. In static buffering considerations, the information required from the graph is different from the partitioning of the previous section. Galaxies will again be encircled, but the stars inside are not necessarily of the same rate.

4.1. Local Buffers

The task of generating iterated in-line code is more difficult than generating simple in-line code because of the combined usage of static and non-static buffering. As a simple example of iterated in-line code, consider figure 7. Because of the 3 delays on the arc between B and C, its schedule is CA(2BCCBC)BCCBD. The idea of iterated in-line code is to code (2BCCBC) as a loop of 2 around in-line code for BCCBC. First note that for the in-line code for BCCBC, B will need to read from 2 fixed buffer locations, and C will write to 3 fixed locations. However, B would normally be connected to A by a buffer of length 6, and C connected to D by a buffer of length 9. A solution to this problem is shown in figure 7 where two shorter arcs are temporarily subsituted during the loop. These temporary *local buffers* are drawn as dotted arcs, and the large arrows indicate the samples that must be copied into and out of them for each iteration. This copying is done at runtime by the code implementing the loop.

In general, iterated in-line code uses static buffering almost identical to the usual in-line code except inside loops certain buffers are subsituted by local, temporary buffers. At the top level of the schedule, all buffers are static buffers, and code generation is the same as in-line. When a loop is encountered, buffers are subsituted before in-line code is produced for the loop. The buffers to be subsituted are identified by drawing a circle around the stars of the sub-schedule. The data arcs entering the circle are *loop input* arcs, and the those leaving are *loop output* arcs. These input and output arcs will be replaced by the local buffers during code generation in that loop. Also, for the sub-schedule to be correctly coded as in-line code,

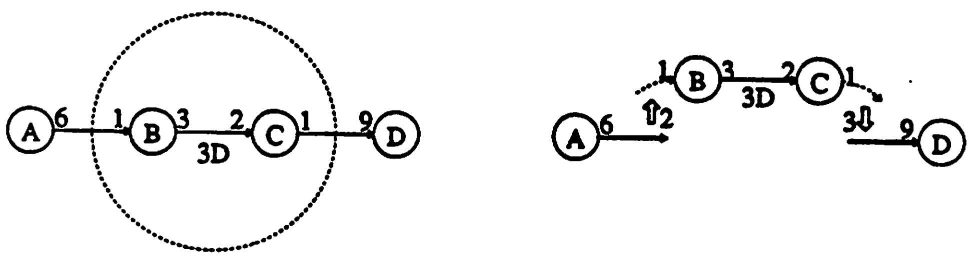


Figure 7. Buffering requirements for iterated in-line code.

these encircled stars must form a galaxy. This is required because the internal arcs are implemented as a set of fixed memory locations in the in-line code of the loop. No net samples can accumulate on these arcs during an iteration.

4.2. Loopstars

Once the loop inputs and outputs have been identified, local buffers must be subsituted for them before code is generated for the loop. In addition, samples from the input buffers must be copied to the local arcs before the loop body, and samples must be copied from the local buffers to the output arcs after the loop body. Both these functions are accomplished by *loopstars* which are subsituted for the parenthesis and iteration factors of the iterated schedule. In the example of figure 7, CA(2BCCBC)BCCBD becomes CA !top1 BCCBC !end1 BCCBD. This flat schedule is sent to Gabriel's in-line code generator. The loopstars !top1 and !end1 need to have compiler directive-type instructions to subsitute the buffers, and code generating instructions to write the copying code. Note that these stars are spliced into the schedule and not the graph, as with interprocessor communication stars. The splicing of loopstars occurs before code generation as shown in figure 1, and in this step all local buffers are identified, their sizes are determined, and their memory allocated. The read/write pointers for the copying of samples from/to the loop input/output buffers are also allocated. Once these parameters are determined for the loopstars, they are used during in-line code generation as:

```
!topstar
  save loop input and output buffers
  subsitute local buffers
  emit_code " do      #" iteration_factor , !loop
  emit_code {for each input
                copy input samples to local buffers
                update read pointer}

{ in-line code for loop }

!endstar
  emit_code {for each output
                copy local samples to output buffer
                update write pointer}
  emit_label !loop
  restore loop input and output buffers
```

The actual DSP assembly code implementing the loopstars is made as efficient as possible by making an attempt to implement the loop input read pointers and the loop output write pointers as DSP56000 address registers. While the loopstars are being spliced into the schedule, all the stars inside a loop are checked for unused address registers. This is done from the innermost loop outward, and the unused registers are first allocated to the inner loops. This is the most efficient allocation since the innermost loop is run the most often. The star must contain a declaration of its address register use (see manual page for "defstar"), or it is assumed all the address registers are affected by the star.

The local buffers subsituted by the loopstars are similar to local variables used in recursion. The iterated in-line code analogy to recursion is nested looping. An example of local buffers and nested looping is shown in figure 8. The three drawings represent the buffers that exist on the top level, in the first loop, and in the nested loop. In going from the first loop to the nested loop, a second local buffer is generated for C, and the code generator thinks that C's first local buffer is the loop output buffer. Note that the samples generated by C are first copied from the second local buffer to the first 3 at a time, then when 9 samples fill the first local buffer, they are copied again onto the static buffer. This double copying is characteristic of nested loops where there are two consecutive parenthesis. A more complicated compiler would eliminate the unnecessary copying.

4.3. Global Buffers

So far it has been assumed that leaving the internal arcs of a loop as the static buffers of the level above is valid. It is necessary that the arcs buffer data correctly between star invocations inside the loop and between star invocations outside the loop. The static buffer must also pass data between stars straddling a loop. To meet the conditions for static buffering inside a loop, no net samples can be accumulated on the arc. This condition is checked by a function in the block entitled "filter schedule" in figure 1. This function is passed an iterated schedule and returns a valid iterated schedule where all the buffers internal to a loop are static. This check is done recursively to allow for nested looping. For example, consider the graph in figure 2 with 2D placed on the arc. The schedule is BABAB. However, (2BA)B is not valid since the internal arc in the loop accumulates one sample per period. The exploded schedule BABAB is returned. If A and B were actually loops themselves, then BABAB will still contain those loops. In this manner, the maximum looping is maintained.

The above filter on the iterated schedule ensures that at all levels, internal arcs can be implemented as static buffers. This means that buffering will be correct on these arcs for star invocations within the same loop. But the global static buffers must

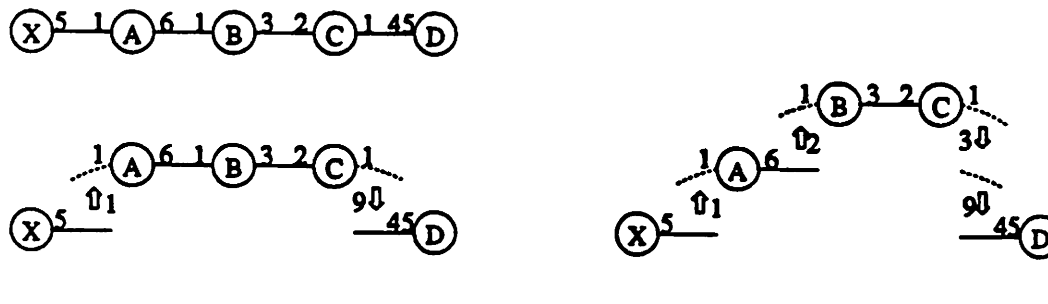


Figure 8. An example of a multirate graph with nested iteration. The temporary local buffers are shown as dotted arcs on hte top level; the first loop; and the nested loop.

also pass data between stars inside and outside of a loop. In the example of figure 7, the schedule CA(2BCCBC)BCCBD has firings of B and C straddling the loop. For the buffering between stars B and C to always be correct, it is necessary that a common static buffer size exist for all levels. In this example, B and C require a buffer of size of 6 inside the loop, and also in the top level CA()BCCBD. The first firing of C outside the loop consumes 2 samples from the arc, leaving one, and passes the buffer pointers to the loop. The loop then correctly processes the samples on the arc. Since the buffer is static in the loop, the buffer pointers exit the loop as they entered it. Then, the rest of the top level schedule continues with one sample on the arc as expected. In general, if a loop uses a static buffer for an internal arc and both required buffer sizes are the same, then the loop has no effect on the buffer pointers as seen by the outer level. However, the loop uses the samples on the arc initialized by the outer level, and leaves the most current samples on the arc when it exits. This compatability of global buffer sizes is also checked in the "filter schedule" block, because the incompatable subschedules are returned exploded. After this block, a valid iterated schedule is sent to the rest of the code generator.

5. Example - 2400BPS Data Channel

As an example of code generation for a multirate graph, a 2400bps 4PSK voiceband channel was designed and implentented on Gabriel. The transmitter for the channel consisted of a Motorola DSP56000 evaluation board connected to a modified Magnavox CD player. The receiver was implemented on a seperate Motorola board with an Ariel 12 bit A-to-D converter. Several different channels were used including a DC connection, a Gabriel-based telephone channel simulatior, and an wireless IR channel. The development set-up is shown in figure 9 with a DC channel.

5.1. The Transmitter

A PAM scheme was chosen that would be roughly suitable for use on a phone line. Initially, a baseband binary antipodal channel was developed, then the adaptive filter, timing recovery scheme, etc. were modified for a 4PSK constellation. A symbol rate of 1225Hz was selected based on the CD player's 44.1kHz clock (1225Hz =

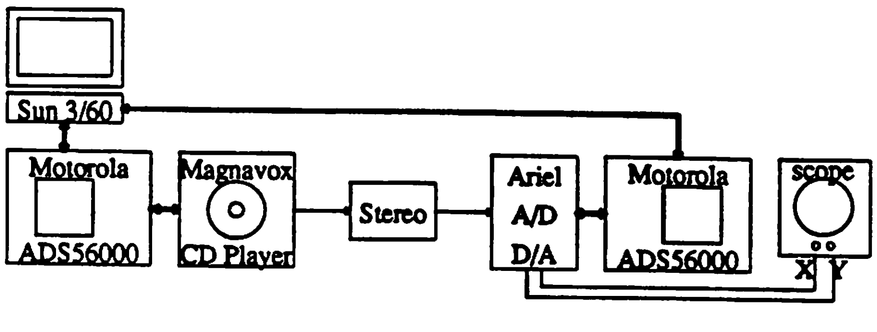


Figure 9. The hardware setup for the voiceband data channel example.

44.1kHz/36). A 100% excess bandwith raised cosine pulse was used for the transmit pulse. This pulse was rectangularly windowed to contain exactly 2 zero-crossings on either side of the pulse. The carrier frequency was set at 3/2 times the symbol rate, approx 1800Hz. This frequency spectrum was chosen to approximate a map needed for full a duplex transmission (and without any knowledge of standards like v22.bis.)

The Gabriel universe for the transmitter is shown in figure 10. The graph is partitioned into groups of the same frequency. In the transmitter, most of the computation is done at the slower rate of 14.7kHz and then interpolated to 44.1Khz by a zero-order hold for 3 samples. This is done to conserve processor time, since oversampling the transmit pulse at 12 samples per baud interval is sufficient to keep the replicated spectrums far out of band. Also, if 36 samples were used as a baud interval, the interpolating FIR filter implementing the transmit pulse would have 144 taps as well as operating at 44.1kHz. The interpolation also provides a good example of a nested schedule since it adds an intermediate 14.7kHz to the 44.1Khz CD rate and the 1225hz baud rate.

Three schedules are shown in figure 11. The first is the flat product of the original scheduler. The length of the schedule is indicative of the DSP code, which loads as 2.2k words in the Motorola's 4k word memory. The second schedule is the iterated form of the first, and shows that an optimized multirate scheduler is needed. Also, grouping patterns for long schedules becomes impractical as the pattern recognizer can require on the order of N^2 comparisons, where N is the number of stars in the schedule. Especially in Lisp, scheduling the simplified graph is many times faster. Finally, the output of the multirate scheduler is listed, and its iterated in-line code is 266 words long.

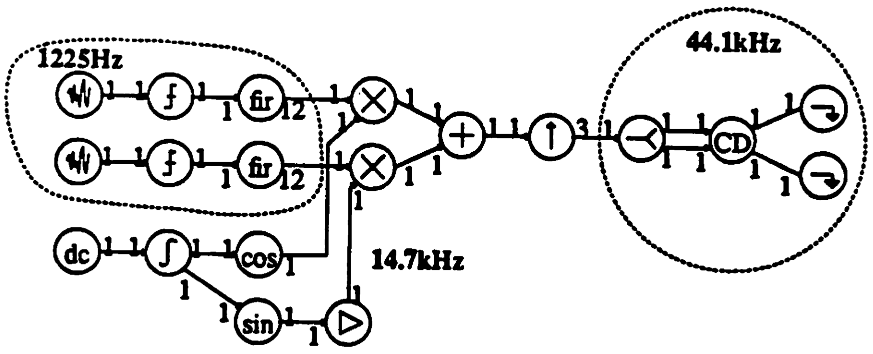


Figure 10. The gabriel universe for the QAM transmitter. The CD star performs the D/A at 44.1Khz.

```
(|56noise1| |56quantizer1| |56noise2| |56quantizer2| |56dc1| |56integrator1| |56fir2| |56fir1| !af1 |56dc1|
|56integrator1| |56cos1| |56sin1| !af1 |56dc1| |56integrator1| |56mult1| |56cos1| |56gain1| |56sin1| !af1
|56dc1| |56integrator1| |56mult2| |56gain1| |56sin1| |56add1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| !af1 |56dc1| |56integrator1| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| |56add1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| !af1 |56dc1| |56integrator1| |56mult1| |56mult2| |56cos1|
|56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56dc1| |56integrator1|
|56upsample1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56dc1| |56add1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| !af1 |56integrator1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| |56dc1| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1| |56mult2|
|56cos1| |56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56integrator1|
|56upsample1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56dc1| |56add1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| !af1 |56integrator1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| |56dc1| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1| |56mult2|
|56cos1| |56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56integrator1|
|56upsample1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56dc1| |56add1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| !af1 |56integrator1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1|
|56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56upsample1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1|
|56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2|
|56mult1| |56mult2| |56cos1| |56gain1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2|
|56mult1| |56mult2| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56upsample1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af2
|56magnavox1| |56black_hole1| |56black_hole2| !af2 |56magnavox1| |56black_hole1| |56black_hole2|)

(|56noise1| |56quantizer1| |56noise2| |56quantizer2| |56dc1| |56integrator1| |56fir2| |56fir1| !af1 |56dc1|
|56integrator1| |56cos1| |56sin1| !af1 |56dc1| |56integrator1| |56mult1| |56cos1| |56gain1| |56sin1| !af1
|56dc1| |56integrator1| |56mult2| |56gain1| |56sin1| |56add1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| !af1 |56dc1| |56integrator1| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| |56add1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| !af1 |56dc1| |56integrator1| |56mult1| |56mult2| |56cos1|
|56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56dc1| (5 |56integrator1|
|56upsample1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56dc1| |56add1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| !af1) |56integrator1| |56upsample1| !af2 |56magnavox1| |56black_hole1|
|56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1| |56mult2| |56cos1|
|56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| !af1 |56upsample1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56mult1|
|56mult2| |56cos1| |56gain1| |56sin1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2|
|56mult1| |56mult2| |56cos1| |56gain1| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2
|56magnavox1| |56black_hole1| |56black_hole2| |56add1| !af2 |56magnavox1| |56black_hole1| |56black_hole2|
|56mult1| |56mult2| !af2 |56magnavox1| |56black_hole1| |56black_hole2| |56upsample1| !af2 |56magnavox1|
|56black_hole1| |56black_hole2| |56add1| (2 !af2 |56magnavox1| |56black_hole1| |56black_hole2|) |56upsample1|
(3 !af2 |56magnavox1| |56black_hole1| |56black_hole2|))

(|56noise2| |56quantizer2| |56fir2| |56noise1| |56quantizer1| |56fir1| (12 |56dc1| |56integrator1| !af1
|56cos1| |56mult1| |56sin1| |56gain1| |56mult2| |56add1| |56upsample1| (3 !af2 |56magnavox1| |56black_hole2|
|56black_hole1|)))
```

Figure 11. The schedules of the QAM transmitter: the original scheduler output, it's iterated form, and the output of the multirate scheduler.

5.2. Receiver

The receiver was also implemented on a single processor Motorola evaluation board connected to an Ariel A/D and D/A board. The analog front-end of the receiver consisted only of the 12 bit A/D converter which was triggered directly by a signal from the DSP56000's SSI port. No external anti-aliasing or external pll circuitry was required. The basic architecture of the receiver is shown in the simplified diagram of figure 12. The signal is first filtered by a matched analytic bandpass filter and then demodulated by the nominal carrier frequency before entering the timing and carrier recovery loops. A decimating complex adaptive filter using the lms algorithm was written to implement a fractionally spaced equalizer. The actual Gabriel universe implementing this receiver is included in Appendix A which describes some details of the timing recovery. The partitioned graph of connected groups of stars of the same frequency is shown in figure 13. This simplified graph is initially deadlocked, and the 1225Hz group is be re-partitioned into 9 groups after pulling the delays out. The resulting schedule is listed in figure 14, again compared to the original scheduler. In this example, the receiver universe doesn't contain as high a sample rate ratio as the transmitter did, and the benefits of the scheduler aren't as dramatic. However, if any coding was added to the channel, iterated in-line code would again be critical in implementing the system.

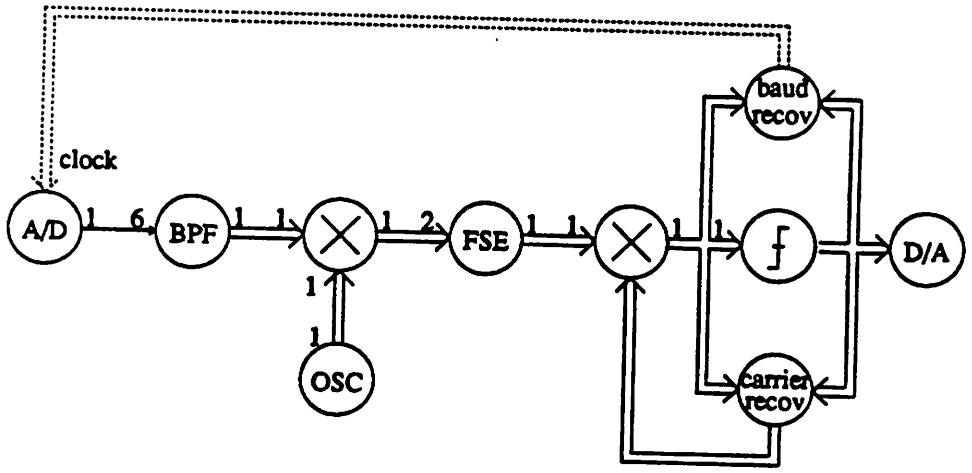


Figure 12. Simplified block diagram of the receiver implemented in Gabriel.

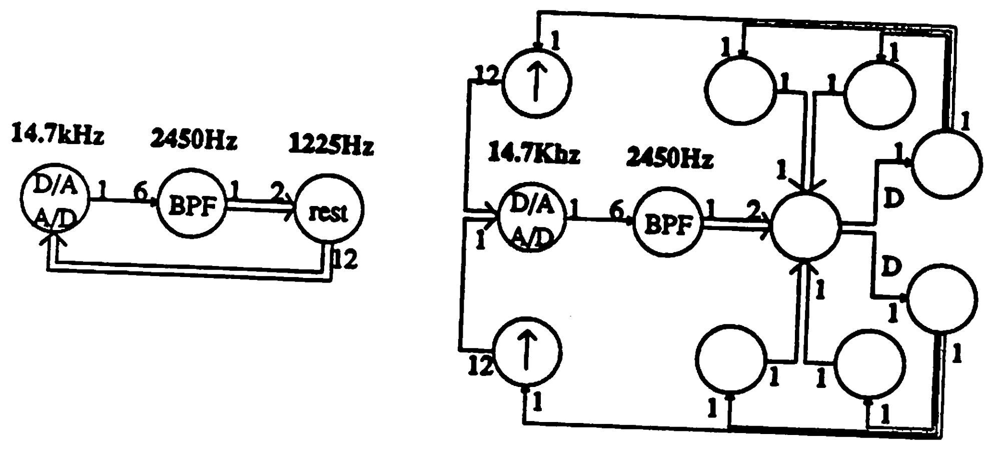


Figure 13. Partioning of Gabriel receiver universe into simplified graph. The graph is repartitioned to break the deadlock.

```
(|56data_source1| |56data_source2| |56fork1| |56fork2| !af9 !af10 |56data_source3| !af11 !af12 |56zero_hld1|
|56quantizer2| !af3 |56sub2| car_rec1%56gain1 |56zero_hld2| !af7 |56quantizer1| !af8 |56adda21|
car_rec1%56cx_mult1 car_rec1%56black_hole1 |56sub1| !af6 |56gain2| !af2 |56adda21| car_rec1%56dsplay_bq1
|56gain2| !af2 |56adda21| car_rec1%!af1 car_rec1%56sin1 car_rec1%56cos1 !af4 !af5 |56gain2| !af2 |56adda21|
|56gain3| |56cx_mult4| |56gain2| !af2 |56adda21| |56gain2| !af2 |56adda21| |56gain2| !af2 |56adda21| |56fir1|
|56fir2| |56gain2| !af2 |56adda21| |56cx_mult2| |56data_source1| |56gain2| !af2 |56data_source2| |56adda21|
|56gain2| !af2 |56adda21| |56gain2| !af2 |56adda21| |56gain2| !af2 |56adda21| |56gain2| !af2 |56fir1| |56fir2|
|56cx_mult2| |56cx_lms1| |56cx_mult3| |56fork3| |56fork4| |56cx_add1| |56gain4| |56gain5| |56gain6|
|56cx_mult5| |56black_hole2| |56fir3| |56gain7| |56add1| |56quantizer3| |56baud1|)
```

```
(|56data_source1| |56data_source2| |56fork1| |56fork2| !af9 !af10 |56data_source3| !af11 !af12 |56zero_hld1|
|56quantizer2| !af3 |56sub2| car_rec1%56gain1 |56zero_hld2| !af7 |56quantizer1| !af8 |56adda21|
car_rec1%56cx_mult1 car_rec1%56black_hole1 |56sub1| !af6 |56gain2| !af2 |56adda21| car_rec1%56dsplay_bq1
|56gain2| !af2 |56adda21| car_rec1%!af1 car_rec1%56sin1 car_rec1%56cos1 !af4 !af5 |56gain2| !af2 |56adda21|
|56gain3| |56cx_mult4| (3 |56gain2| !af2 |56adda21|) |56fir1| |56fir2| |56gain2| !af2 |56adda21| |56cx_mult2|
|56data_source1| |56gain2| !af2 |56data_source2| (4 |56adda21| |56gain2| !af2) |56fir1| |56fir2| |56cx_mult2|
|56cx_lms1| |56cx_mult3| |56fork3| |56fork4| |56cx_add1| |56gain4| |56gain5| |56gain6| |56cx_mult5|
|56black_hole2| |56fir3| |56gain7| |56add1| |56quantizer3| |56baud1|)
```

```
(|56fork2| !af12 |56zero_hld2| |56fork1| !af11 |56zero_hld1| (2 (6 |56adda21| |56gain2| !af2) |56fir1|
|56data_source2| |56data_source1| |56fir2| |56cx_mult2|) |56quantizer1| !af8 |56data_source3| !af9
|56quantizer2| !af3 car_rec1%56gain1 !af10 |56sub2| !af7 |56sub1| !af6 car_rec1%56cx_mult1
car_rec1%56black_hole1 car_rec1%56dsplay_bq1 car_rec1%!af1 car_rec1%56sin1 !af5 |56gain3| car_rec1%56cos1 !af4
|56cx_mult4| |56cx_lms1| |56cx_mult3| |56fork4| |56fork3| |56cx_add1| |56gain6| |56cx_mult5| |56fir3|
|56gain7| |56add1| |56quantizer3| |56baud1| |56black_hole2| |56gain5| |56gain4|)
```

Figure 14. Three schedules for the QAM receiver: the original scheduler output, it's iterated form, and the output of the multirate scheduler.

6. Conclusions

A set of Lisp functions were written as a preprocessor to the Gabriel in-line code generator to allow for efficient coding of multirate universes. Many DSP systems operate at several sample rates, where the sample rate ratios may be very high, as in speech coding, transform coding, modems, etc. Gabriel previously attempted to produce in-line code for these universes, and in many cases the generated code was impractically long. A modified form of in-line code was proposed where loops are put around the repetitive portions of the schedule. The existing Gabriel in-line code generator was used to produce this iterated in-line code with the use of loopstars using directive-type instructions and local buffers. Thus the multirate code generation preprocessor is seperate from the in-line code generator, and no modifications were made to the existing Gabriel code. A scheduler designed for producing natural-flow iterated schedules was developed to maximize the efficiency of iterated in-line code. As an example of the multirate code generating system, a voiceband data channel was developed and code was generated for this system which resulted in a dramatic improvement compared to in-line code. Code was generated by other students for a LPC-type speech coder universe which would not have been implementable using the in-line code generator. For prototyping many multirate systems, it is will probably be necessary to use the multirate code generator.

APPENDIX A
A Baud-rate Timing Recovery Scheme

The actual Gabriel universe implementing the QAM receiver is shown in figure 21. Its basic architecure is fairly standard and was described in Section 5. This section will describe the algorithm for estimating the phase error between the receiver baud clock and the incoming baud interval. This estimator was used in the timing recovery loop.

When using Gabriel in the simulator mode, it was noticed that an error in sampling phase manifiested itself as imbalanced ISI in the equivilent discrete-time channel. That is, the difference between the pre-cursor and post-cursor components of the ISI was seen to be approximately proportional to the timing phase error. A simple estimator of this differential ISI was implemented around the slicer, and the timing recovery loop based on this estimate converged for baseband and passband receivers in real-time. The estimator is described quantitatively for a simple receive pulse, and more graphically for the more general cases. However, the result is the same as a stochastic gradient timing recovery algorihtm [3], and this description provides a more geometric interpretation to the update equations. Timing phase error τ is defined for $-T_{baud}/2<\tau\leq T_{baud}/2$ as in figure 15. A demodulated, equalized, isolated pulse is shown sampled with both positive and negative timing phase error. Under these circumstances the samples represent the equivilent discrete-time channel, and thus the ISI. In this example, the equalized pulse meets the zero-forcing criterion. Note that for $\tau>0$ the pre-cursor ISI is negative while the post-cursor component is positive.

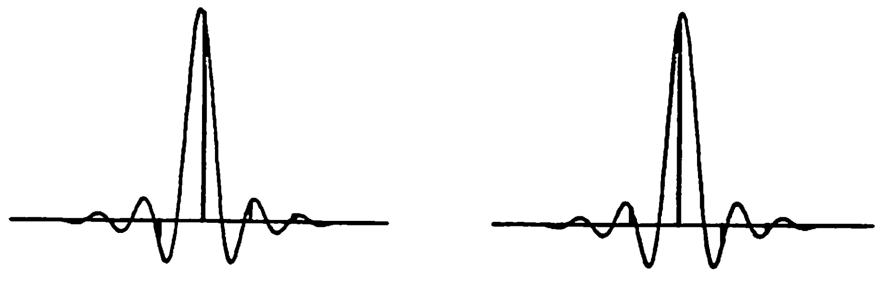


Figure 15. Positive and negative timing phase error for received isolated pulse with a zero-forcing equalizer. The equally spaced samples also represent the equivilent discrete-time channel and ISI. For $\tau>0$ pre-cursor ISI is negaitve and the post-cursor contribution is positive. The situation is reversed in the case of $\tau<0$.

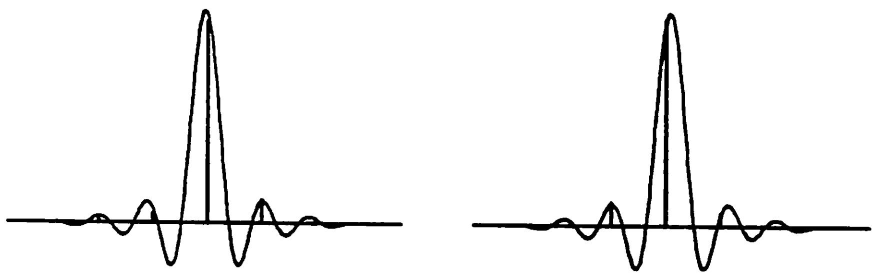


Figure 16. Positive and negative timing phase error for received isolated pulse with a non-ZF equalizer. The samples also represent the equivilent discrete-time channel and ISI. For $\tau>0$ pre-cursor ISI is less than the post-cursor contribution. The situation is opposite for $\tau<0$.

When $\tau<0$, the situation is opposite. An example of a non-zero-forcing equalized pulse, figure 16 shows the same sampling on a pulse with zero-crossing intervals less than the baud period. The relationship between differential ISI and τ are similar in both these examples.

To quantitatively show the relationship between ISI imbalance and τ, assume that the received QAM or PAM signal has been equalized and demodulated. The complex equivilent baseband received signal is

$$x(t)=\sum_{k=-\infty}^{\infty} A_k b(t-\tau-kT) \tag{2}$$

where A_k are the possibly complex symbols of the constellation, and $b(t)$ is the real equivilent baseband pulse which describes the transmit pulse through the channel. After equalization,

$$y(t)=\int_{-\infty}^{\infty} \sum_{k=-\infty}^{\infty} A_k b(t-\tau-kT-\gamma)\, h(\gamma)\, d\gamma \tag{3}$$

where $h(t)$ is the equalizer impulse response. Interchanging integration and summation and then sampling at $t=nT$ gives

$$y(n)=\sum_{k=-\infty}^{\infty} A_k \int_{-\infty}^{\infty} b(nT-\tau-\gamma-kT)\,h(\gamma)\,d\gamma \tag{4}$$

The integral repesents the equivilent discrete-time channel since

$$y(n)=\sum_{k=-\infty}^{\infty} A(k)\,\rho_\tau(n-k), \qquad \rho_\tau(n)=\int_{-\infty}^{\infty} b(nT-\tau-\gamma)\,h(\gamma)\,d\gamma \tag{5}$$

For $\tau=0$ and a zero-forcing equalizer, $b(t-nT)$ and $h(t)$ are orthogonal for all $n\neq 0$. Then $\rho_0(n)=\delta(n)$ and there is no isi. Consider the simple case where

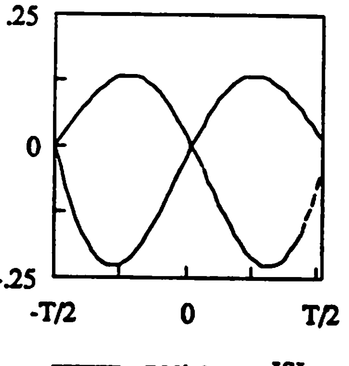
$$b(t)=h(t)=\frac{\sin(2\pi t/T)}{2\pi t/T} \tag{6}$$

are 0% excess bandwidth Nyquist pulses. In this simple case the convolution of two sinc functions is itself a sinc function, and sampling the sinc function at $t=nT-\tau$ gives the equivilent channel

$$\rho_\tau(n)=\rho(nT-\tau)=b(t)*h(t)\Big]_{t=nT-\tau}=\frac{\sin(2\pi t/T)}{2\pi t/T}\Bigg]_{t=nT-\tau} \tag{7}$$

For this simple case, $\rho_\tau(-1)$ (pre-cursor ISI) and $\rho_\tau(1)$ (post-cursor ISI) is plotted for $T_{baud}/2<\tau<T_{baud}/2$ in figure 17. Pre-cursor ISI is positive and post-cursor ISI is negative for $-T_{baud}/2<\tau<0$, and visa versa for $\tau<T_{baud}/2$. Their difference, the "differential ISI" is also plotted against τ. This differential ISI isn't a monotonic function of τ, but indicates the required direction for phase correction. Since timing phase is corrected for in a digital PLL, only the sign of τ is important anyway.

In the general case where the equalization is not zero-forcing, the differential ISI is still a similar function of τ. In figure 18 and figure 19, pre-cursor and post-cursor



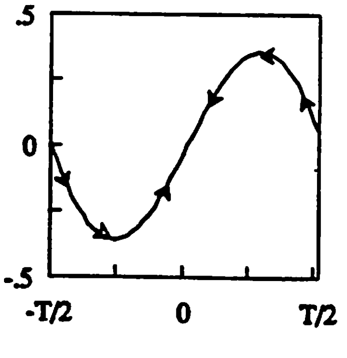


Figure 17. Pre-cursor and post-cursor ISI as a function of timing phase error τ. The differential ISI as a function of τ. The arrows show the direction of the phase adjustment as a function of τ.

ISI is shown for pulses with zero-crossings at intervals slightly larger and smaller than T_{baud}. The differential ISI versus τ for both these cases have the same shape as the zero-forcing case. Looking at this timing recovery scheme graphically, equalizing the pre-cursor and post-cursor ISI is equivilent to centering isolated pulse as in figure 15 and figure 16 and obtaining τ=0.

The simple estimator of differential ISI in the receiver uses 3 input samples to the slicer, and 1 sliced symbol. The algorithm for three transmitted symbols -1-i, -1+i, and 1+i are received. The received version of the symbol -1+i is affected by the other two transmitted symbols according to the ISI. If post-cursor ISI is larger than pre-cursor ISI, then the received version of -1+i would be "drawn" towards 1+i. In other words, the received data symbol will contain a larger component of the future symbol if $\rho_\tau(1)>\rho_\tau(-1)$. In general, an error vector from the slicer output to the slicer input is

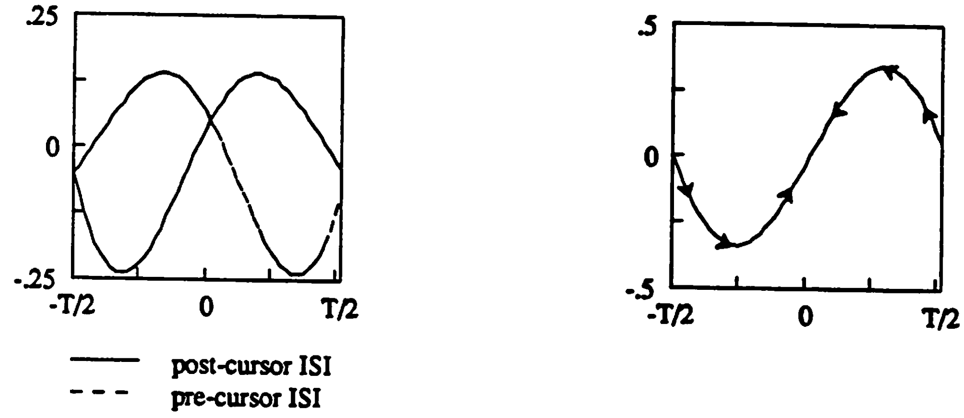


Figure 18. Pre-cursor and post-cursor ISI as a function of timing phase error τ. Differential ISI as a function of τ. The equalized receive pulse has zero-crossings at intervals > T_{baud}.

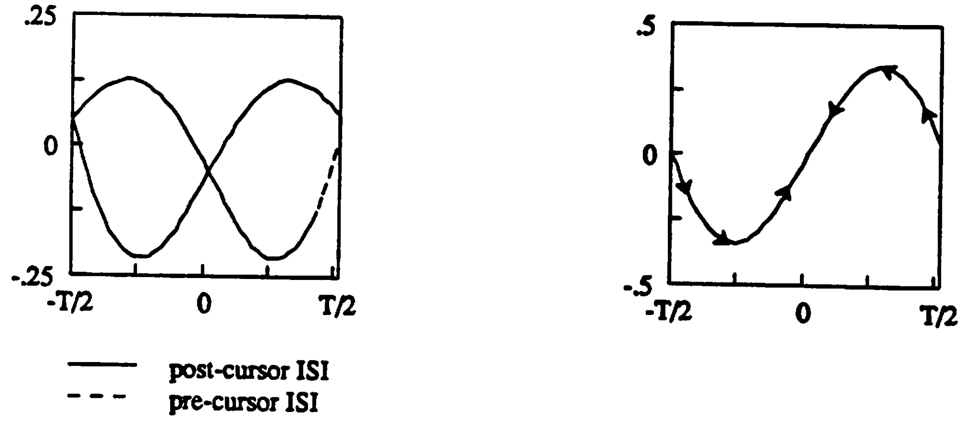


Figure 19. Pre-cursor and post-cursor ISI as a function of timing phase error τ. Differential ISI as a function of τ. The equalized receive pulse has zero-crossings at intervals < T_{baud}.

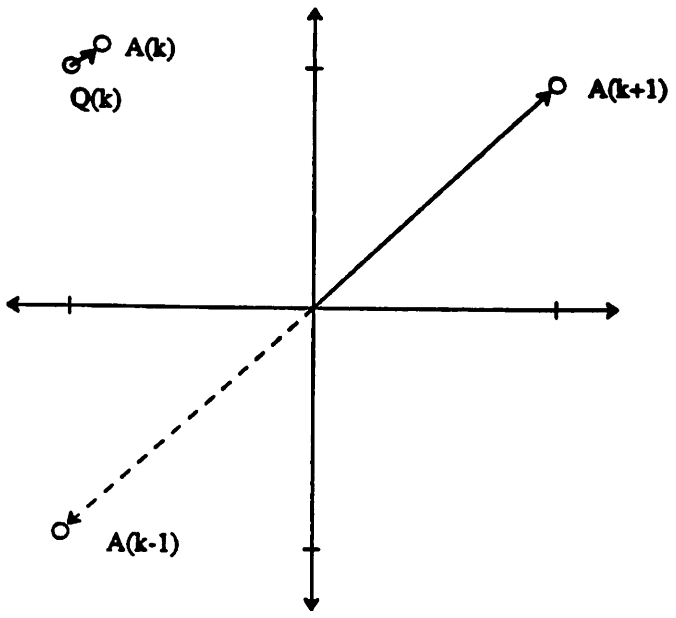


Figure 20. Three received symbols arriving in a clockwise order. The difference vector from the slicer output -1+i to the slicer input shows a component in the direction of the future sample.

computed. The projections of this error vector onto the previous and future transmitted symbol vectors gives an estimate of the pre-cursor and post-cursor ISI components. The difference of the components is then used as an estimator of the differential ISI

$$\Delta ISI_{est} = (A_k - Q_k)\cdot A_{k-1} - (A_k - Q_k)\cdot A_{k+1} = (A_k - Q_k)\cdot(A_{k-1} - A_{k+1}) \qquad (8)$$

where the A_k's are the slicer inputs and Q_k is the slicer output. Note that if the previous and the future sample are the same symbol, the timing phase estimate will be small. This corresponds to the case when the transmitted sequence doesn't contain any timing information, as 101010101... . On average, for a white symbol stream timing information will be available at 1/2 the symbol rate, since

$$Pr[Q_{k-1} \neq Q_{k+1}] = .5 \qquad (9)$$

Assuming $\Delta ISI_{est} = k\tau$, then the timing phase update equation can directly use the above inner product as

$$\Delta\tau = \mathrm{Re}\left\{ (A_k - Q_k)^* (A_{k-1} - A_{k+1}) \right\} \qquad (10)$$

This result is equivilent to Qureshi [4], but provides a graphical interpretation rather than a derivation based on an approximated discrete-time derivative. The form of the estimator shows it can be easily implemented around the slicer.

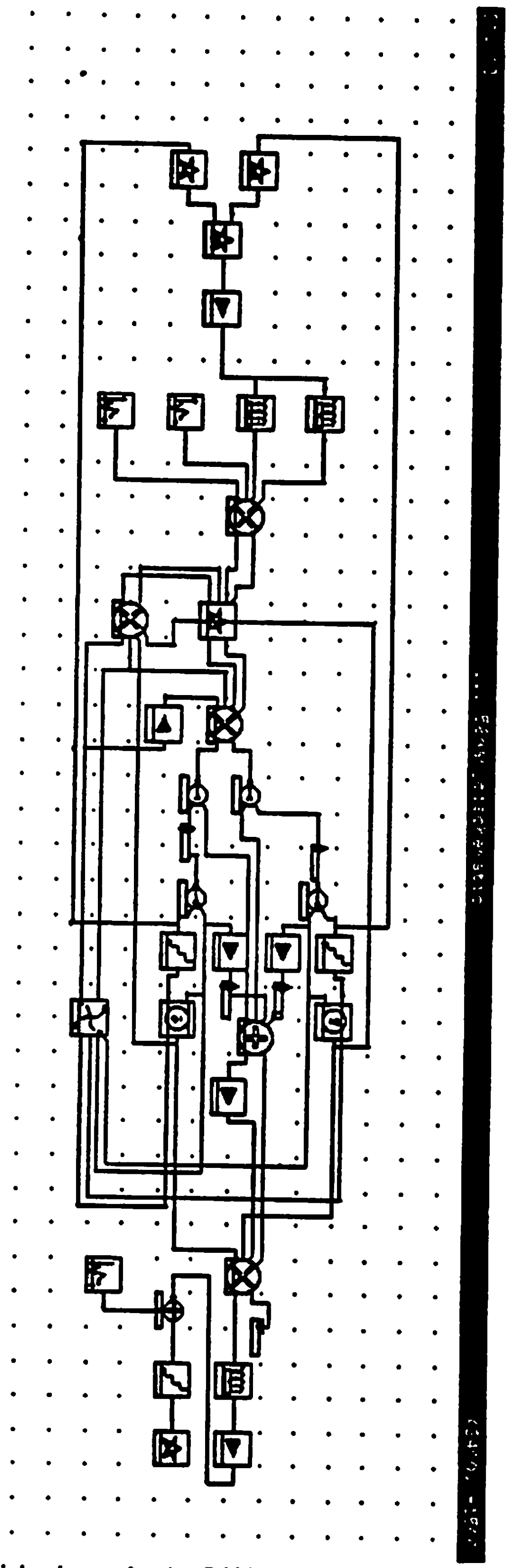


Figure 21. The Gabriel universe for the QAM receiver.

References

[1] E.A. Lee and D.G. Messerschmitt, "Syncronous Data Flow", *Proceedings of the IEEE,* September, 1987.

[2] E.A. Lee, W. Ho, E. Goei, J. Bier, and S. Bhattacharyya, "Gabriel: A Design Environment for DSP", *IEEE Trans. on ASSP,* November, 1989.

[3] E.A. Lee and D.G. Messerschmitt, *Digital Communication,* Kluwer Academic Publishers, Boston, MA (1988).

[4] S.U.H. Qureshi, "Timing Recovery for Equalized Partial-Response Systems", *IEEE Trans. on Communications,* December, 1976.