GABRIEL: A DESIGN ENVIRONMENT FOR PROGRAMMABLE DSPs

E. A. Lee, E. Goei, H. Heine, W. Ho, S. Bhattacharyya, J. Bier, E. Guntvedt

U. C. Berkeley
Berkeley, CA 94720
eal@janus.Berkeley.EDU

ABSTRACT

Gabriel is a retargetable software system for the development of assembly code and microcode for single or multiple programmable DSPs. It is intended to ease code development even for processors that are not easy targets for conventional compilers. Code generation for the Motorola DSP56001 is emphasized. A Thor-based simulator supplies a variety of target multi-DSP architectures based on the DSP56001. The top-level algorithm description is a large grain data flow graph, and a graphical interface using OCT and VEM provides a natural representation of the high level structure of the algorithm.

1. INTRODUCTION

At the highest level, Gabriel applications are described as block diagrams (equivalent to large-grain data flow graphs [Bab84]). Feedback is efficiently supported, as are changes in sample rates. Blocks can have arbitrary granularity, and may come from a standard library or be generated by the user. A static (compile time) schedule is devised, and code generated to implement the functionality represented by the blocks. Each block is a parametrized code generator written in Lisp; this approach appears to be flexible enough to support target processors, such as the fastest DSP chips or VLSI core processors, for which traditional high level language compilers are not practical. To permit static scheduling, blocks must fit the *synchronous data flow* model [Lee87a][Lee87b]. This implies some significant limitations on the algorithms that can be represented, although we are working to overcome these limitations.

Synchronous data flow is a special case of data flow where the number of tokens produced or consumed by an actor when it fires is specified ahead of time. The number cannot be data dependent. This implies that SDF graphs cannot have data-dependent firing of actors, a significant limitation for many applications. Although we have developed more general scheduling strategies, they are not yet implemented in our software, so they will not be described here.

Gabriel is the second generation of DSP design environments at Berkeley, the first generation being *Blosim* [Mes84a][Mes84b], a simulation program also based on block diagrams. One consequence of this heritage is that some of the Blosim terminology survives in the new system. Specifically, function blocks in the block diagram, also called *actors* in the data flow literature, are called *stars* in Gabriel. The block diagram can be hierarchical, with a cluster of stars being called a *galaxy*. A galaxy can be manipulated as if it were a star, and can itself contain galaxies. An entire application is called the *universe*. Blosim, however, is a DSP *simulation* environment, whereas Gabriel is a real-time *implementation* environment. Some of the generality of Blosim had to be sacrificed in order to get efficient implementations.

A number of other block-diagram systems for DSP have appeared in recent years. Three commercially available programs are *BOSS* [Sha87] from the University of Kansas, *DSPlay* from Burr-Brown, and the Signal Processing Worksystem from Comdisco. BOSS and the Comdisco system are both currently strictly simulation systems, not aimed at real-time implementations. BOSS is particularly distinguished for its project management facilities. DSPlay is a PC based system for both simulation and code generation (for the AT&T DSP32). In the opinion of the authors, all three systems suffer from not using synchronous data flow techniques.

Two important non-commercial block-diagram systems aimed at real-time implementations on TMS320 systems are a prototype program from Lincoln Labs [Zis86]

[1] We gratefully acknowledge the support of Cygnet Technologies, a division of Everex Systems, Inc., Digital Equipment Corporation, Sony Corporation, and Hayes Inc., plus matching funds from the State of California Micro Program.

and Gospl [Cov87]. By not using synchronous data flow techniques, however, both systems are constrained to single sample-rate applications. Gospl does not attempt to parallelize the code. Techniques for *fine-grain* parallel real-time implementations have been investigated at Georgia Institute of Technology [Sch85][Sch86].

Block diagram languages for simulation actually have a long and distinguished history; this testifies to their attractiveness to the DSP community. A number of systems were proposed or built in the 1960s [Der69][Gol69][Kar65][Kel61] and 1970s [Cry74][Kor77][Hen75]. Since all of these systems were aimed at simulation, not implementation, they did not exploit one of the principle advantages of block diagram descriptions, which is the ability to automatically parallelize the implementation using data flow techniques.

2. THE USER MODEL

Gabriel runs as two concurrent Unix processes, a VEM process handling the graphical interface, and a Lisp process handling the code generation. It is possible to bypass the graphical interface and interact directly with the Lisp process. This is useful if a graphics workstation is not available.

Graphical representation of computer programs is a controversial topic. For general-purpose programming, many experimental systems have been disappointing. An interesting experiment is *Pict* [Gli84], a visual programming system in which entire programs are built graphically. However, many such systems go to the extreme of attempting to express everything graphically, and the result is severe limitations in the complexity of programs, and cryptic and arbitrary symbols representing concepts that have no natural visualization. Gabriel uses graphics to represent the high-level structure of the program, leaving many details to the textual definition of the stars. The graphical representation of a data flow organization is appealing, and complexity is easily handled using hierarchy. While we make no claim that a graphical representation of this type is suitable for all applications, we are convinced it is suitable for signal processing.

The Gabriel graphical interface is built using VEM, a graphical editor designed at UC Berkeley for CAD. A sample screen is shown in figure 1. The interconnection of icons represents the system topology and is stored by OCT, a design manager associated with VEM. Using VEM and X windows commands, the user can pan, zoom, resize, and move windows. Icons can be moved, copied, or dragged (dragging preserves connections, moving does not). Commands are all menu driven and have single-key accelerators.

Stars are collected into libraries organized around Unix directories and represented in palettes in the graphical interface. A separate library is used for each target processor. Gabriel currently has libraries for non-real time operation on the local workstation and for real-time operation on the Motorola DSP56000. A small library has also been built for the AT&T DSP32, demonstrating retargetability. The user can easily create star libraries and peruse them.

2.1. Creating Stars

Even the best designed star library will not satisfy the needs of most users. The Gabriel system attempts to make the creation of new stars as easy as possible. If a C compiler is available for the target processor (as it is for the DSP56001), then the functionality of a new star can be defined in C. Otherwise it must be written in the assembly language or microcode of the target. The most flexible mechanism available for doing this in Gabriel is to define a code generator in Lisp. A less flexible mechanism is also provided that requires no Lisp coding, but we will not describe it here.

The definition of a star is best illustrated with a trivial example, the adder represented by the plus sign icon in figure 1:

```
(defstar 56add
     (descriptor "DSP56000 - 2-Input Adder")
     (param saturation "yes")
     (input in1 (type dsp56000))
     (input in2 (type dsp56000))
     (output sum (type dsp56000))
     (function 56add)
)
```

Gabriel keywords are shown in **bold**. The name of the star is "56add". The **descriptor** provides a short summary of the functionality of the star. The **param** entry defines a parameter named "saturation" with default value "yes". Two inputs named "in1" and "in2" and one output named "sum" are defined. The **type** designator prevents the user from accidentally connecting this star to an incompatible star, such as a code generator for the AT&T DSP32. Finally a function named "56add" is associated with the star. The function is a Lisp routine defined as follows:

```
(def_cg_function 56add ()
     (emit_code "   move  x:" in1 ",x0")
     (emit_code "   move  x:" in2 ",a")
     (emit_code "   add   x0,a")
     (if (equal saturation "no")
     then
          (emit_code "   move  a1,x:" sum)
     else
          (emit_code "   move  a,x:" sum)
     )
)
```

This function begins by issuing two move instructions to get the inputs from memory and load them into registers. The inputs are referenced symbolically (by name), and Gabriel will supply the memory locations using an efficient and flexible *static buffering* scheme [Lee87b]. Then an add instruction is issued. Finally an instruction is issued to move the sum to the output memory location, where Gabriel again will supply the memory locations. There are two possibilities for the last instruction, depending on the value of the "saturation" parameter. That parameter specifies how to handle overflow conditions in the adder. If the parameter has value "yes" (the default), then an overflow will result in the DSP56001 writing to memory the largest number consistent with the sign of the result that can be represented in the 24 bit format. The code segment generated is:

```
;code for star3 -- ako of 56add
    move  x:0,x0
    move  x:1,a
    add   x0,a
    move  a,x:2
```

Otherwise, the overflow will be ignored and wraparound permitted.

A number of comments are in order.

- Fixed point processors often have arithmetic properties (such as saturation arithmetic) that are difficult or impossible to represent in conventional high level languages such as C. The Gabriel mechanism, by contrast, has no such difficulty, as illustrated in the above example.
- Depending on the schedule generated by Gabriel, it is possible that the code immediately following the above code segment would move data from memory back into a register. For example, Gabriel may produce the following wasteful code segment at the boundary between two stars:

  ```
  move  a,x:2
  move  x:2,a
  ```

  Our plan is to eliminate such wasteful code segments with a post-optimizer, but this has not been done.
- No star can retain ownership of any registers. In other words, the star can make no assumption about the data in any register on entry. The entire context of the star must be stored in memory. Although this certainly leads to some inefficiency, maintaining modularity would be very difficult without this assumption.
- Because of the above inefficiencies, Gabriel produces more efficient code when the granularity of the stars is large than when it is small. However, Gabriel makes no attempt to exploit concurrency within a star, so large granularity means less concurrency to exploit in a multi-processor target.

In addition to the capabilities illustrated by the above example, Gabriel supports the following:

- A star may allocate memory in the target processor for storage of context information from one invocation to the next.
- A star may have a parameterized number of inputs and outputs.
- A star can have an initialization routine and termination routine. The initialization routine can manipulate or compute parameters (for instance digital filter coefficients) and can write assembly code that will appear before the main loop. This is useful for initializing memory locations and setting up I/O. The termination routine is invoked after the main loop has been written and is useful for defining subroutines.

  Large algorithms will be difficult to represent at the finest level of granularity, so the hierarchical capability of Gabriel should be used. A galaxy is a collection of stars that can be treated as a star. Inputs, outputs, and parameters of a galaxy are bound to inputs, outputs, and parameters of its component stars.

2.2. Multiple Sample Rates and Block Processing

Unlike most block diagram environments, Gabriel easily and efficiently supports multiple sample rates and block processing. This is accomplished using the principles of synchronous data flow [Lee87a][Lee87b]. The user model is simple — the number of samples that the star produces or consumes each time it is invoked is defined as a property of each input and output. For example, the **defstar** command for an FFT would be:

```
(defstar fft
        (descriptor "Computes the FFT of the input.")
        (param order 128)
        (input in (no_samples_used order))
        (output out (no_samples_made (* 2 order)))
        (memory twiddle-factors x (* 2 order))
        (init compute-twiddle-factors)
        (function fft))
```

The order of the FFT is a parameter. The star is not invoked until the number of samples given by the value *order* has accumulated on the input buffer. Gabriel knows to do this because of the **no_samples_used** property of the input. It gives the number of samples required for the star to fire. For most stars, this number is not specified and is assumed to be unity. In the run-time function, samples on the input are accessed using the format *name@number*, where the number gives the position of the sample relative to the most recent one. When the fft star fires, it will produce twice as many samples as it consumes, because this particular implementation outputs the real and imaginary parts sequentially using the same output port. Memory is allocated for the storage of the *twiddle-factors*, which are computed by Lisp code (at compile time) in the initialization routine. The run-time function is called *fft*.

Consider a universe consisting of a signal source connected to the above FFT star connected to a signal output star. Suppose the signal source produces one sample on each invocation. Gabriel will automatically determine that the signal source must be invoked 128 times before the the FFT can be invoked once. In the current release of Gabriel, the 128 invocations are accomplished by in-line code, i.e. 128 repetitions of the code for the signal source. This is obviously seriously wasteful of code space, and we are investigating methods for eliminating this waste.

3. REAL-TIME I/O

Suppose that the signal source in the above example is an A/D converter connected to the FFT star. It is assumed that the target architecture has an A/D converter connected to at least one of its processors. In Gabriel, an A/D star has one output and no inputs. In other words, in accordance with the data flow model, the A/D star is just a source of data that can be scheduled at any time. Suppose now that the FFT is used to implement an FIR filter in real time, and hence will be invoked repeatedly. Using the data-flow model the scheduler simply schedules the A/D star 128 times in a row, then schedules the FFT star once. With periodic repetitions of this schedule, the invocations of the A/D star are far from regular. If on each invocation of the A/D star it waits for input, then most of the time the processor will be idle, waiting for an input. Clearly this approach is not desirable.

We have implemented a simple and familiar solution, double buffering. The A/D star generates code for two routines, an interrupt service routine and a run-time routine. The interrupt service routine is invoked by the hardware independent of the Gabriel scheduler. It collects an input sample and stores it in a buffer. Each location in the buffer has a semaphore to indicate whether it is full or empty. The run-time routine collects data from the buffer, setting the semaphore to indicate that it is empty, and halting to wait for an interrupt only if the buffer is entirely empty. The interrupt service routine will be invoked at regular intervals, while the run-time routine can be invoked at arbitrary intervals. It is easy to determine the precise size of the buffer required by determining the total number of times the A/D star is invoked in one cycle of the periodic schedule.

4. TARGET ARCHITECTURES

Gabriel is intended to be retargetable both in the selection of component processors and in their multiprocessor interconnection. The system is new enough that we have only been able to test this on a limited number of configurations, but we expect it to be most useful for modest parallelism, on the order ten or fewer processors. Our lab is equipped with two target systems, a single-processor system and a four-processors system. The latter system is a prototype provided by Dolby Laboratories, of San Francisco. We have concentrated on the Motorola DSP56001 as the component DSP for several technical reasons [Lee89]. Perhaps most importantly, Motorola provides a simulator that can easily be linked to hardware simulation programs. We have linked this simulator to Thor [VLS86], a functional hardware simulator from Stanford, permitting us to construct a variety of multi-DSP systems in software and produce and test code for them. The salient features of each target architecture, such as the number of processors and the interprocessor communications mechanism, are specified as attributes of the target architecture. The intent is that Gabriel should make as few assumptions as possible.

5. WHY NOT USE C?

C compilers are available for many programmable DSPs, so why do we need Gabriel? Although the efficiency of the code generated by these compilers is often not adequate for cost-competitive, real-time applications, significant improvements are expected as optimizers are developed. Therefore, C compilers might be an attractive long term solution. Furthermore, critical sections of code can be written in assembly language, or better yet, implemented using efficient subroutines from a subroutine library. Also, many engineers have experience with C, so little additional time is required before code development can begin. Finally, writing applications in C ensures portability, so that little code conversion is required when new generations of DSPs are introduced.

These are compelling arguments, and for certain applications, we concur with the conclusion. Programmable DSPs, especially the recent generation of floating-point architectures, are used in non-real-time signal processing. Such applications often use elaborate data structures and have a high percentage of control code (vs. signal processing code). Control code tends to be large and typically involves much decision making. However, for many real-time signal processing applications, and for many high performance architectures, we do not believe that C compilers provide a complete solution. We expect them to be useful tools, but only if used as part of a more sophisticated development environment such as Gabriel. Our reasons follow.

C is not a particularly appealing language for describing signal processing algorithms, while block diagrams are. For instance, there is no clean way to express a delay (z^{-1}) in C. Furthermore, there are no "stream" data types, so signals are not naturally represented in the language. Finally, many of best features of C, such as the flexibility of its data structures, are irrelevant for many signal processing algorithms.

Subroutine libraries are in some ways similar to Gabriel's star libraries. They can be efficiently coded in assembly language, and can provide many of the basic signal processing functions, some of which are awkward to express in C. Both approaches involve some overhead. However, stars need not be passed their *parameters* at run time, only their data. Subroutines can *only* evaluate their parameters at run time. In fact, in Gabriel, the code generated can depend on the value of the parameters! Consequently, the code can be tuned not just to the function desired, but to its parameters. For example, a Gabriel FFT star may have the FFT order as a parameter without penalty because the twiddle factors can be computed at code generation time. If a C subroutine has the FFT order as a parameter, it has no choice but to compute the twiddle factors at run time.

There is a strong trend towards increased functionality in DSP architectures, a trend that makes writing efficient C compilers easier. However, the DSPs with the fastest performance (such as the AT&T DSP16, the Hitachi DSPi, and numerous proprietary microcoded DSP cores) do not have C compilers, and do not have the functionality that makes it easier to write C compilers. Gabriel provides a high level interface that can be used with very high performance devices of limited functionality. The star library is customized for the processor, while the C language is fixed.

6. PARALLEL GABRIEL

The scheduling algorithm used in the current version of Gabriel is given in [Lee87a]. It assumes that the run-time of each star is known reasonably accurately. The less accurate the run-times, the less optimal the schedule. The scheduler also assumes that communicating between any two processors is equally easy, and that the time required is not much greater than what is required to communicate between two stars within one processor. This model is reasonable for shared memory machines with small numbers of DSPs. Fortunately, as long as communication between any pair of processors is *possible*, Gabriel produces code that is robust in the sense that it will work, although possibly not at the predicted speed.

The Gabriel parallel scheduler is invoked before code generation to determine which processor each invocation of each star should be mapped to. The parallel schedule is displayed as shown in figure 1.

The interprocessor communication mechanism is defined as an attribute of the target architecture. For retargetability, it is not built into Gabriel. However, in order to produce robust code, Gabriel assumes handshaking is done when two processor communicate. Note that is a shared memory architecture there is no need for an indivisible test-and-set operation, often required when semaphores are used in shared memory. The reason such a mechanism is often required is that if a processor tests a semaphore and sets it in separate cycles, then it is possible for another processor to set the semaphore between the test and set operations of the first processor. However, because of the data flow structure, the use of the semaphores is sufficiently disciplined that this is not a problem. Specifically, only one processor will ever write to each buffer, and only one processor will ever read from each buffer. The writing processor will not write unless it reads an "empty" semaphore, and the reading processor will not read unless it detects a "full" semaphore. Each processor busy-waits for the semaphore to reach the proper state (it is up to the scheduler to ensure that time is not wasted busy-waiting).

The parallel code generation mechanism has been tested with a four-processor parallel architecture donated by Dolby Labs of San Francisco. It has four DSP56001s, each with private memory, plus a shared memory accessed using bus arbitration. The code generated by Gabriel takes a minimum of about 12 instruction cycles to send or receive data from the shared memory, including the overhead of processing semaphores. While this is lean compared to many interprocessor communication mechanisms on parallel machines, it is high enough to preclude effective exploitation of fine-grain parallelism. We think it can (and should) be improved through architectural modifications to the DSP. Modest reductions can also be obtained with different organizations of the multiprocessor architecture. For example, using our Thor-based simulation we have tested an architecture that uses a multi-ported shared memory, and hence does not require interaction with a bus arbitrator. The reduction in interprocessor communication time is about 25%, but clearly the hardware cost would be significantly increased.

7. CONCLUSION

The synchronous data flow techniques used in Gabriel provide a high level application development environment that can target architectures for which conventional compilers are not suitable. However, there are still significant inefficiencies and limitations. Many of the inefficiencies can be removed, in principle, using a post-optimizer. More fundamental work is required, however, in order to broaden the domain of algorithms and multiprocessor target architectures that can be supported.

8. ACKNOWLEDGEMENTS

Many people have worked on this project. The most important contributors are Wai-Hung Ho [Ho88], who developed the basic code generation mechanism, Holly Heine and Edwin Goei, who developed the graphical interface, Shuvra Bhattacharyya, who developed the interactive parallel schedule display, Jeff Bier, who developed the multi-DSP hardware simulator based on Thor, Erik Guntvedt and Eric Cox, who developed most of the real-time I/O capability, and Andria Wong, who demonstrated retargetability to the AT&T DSP32. Finally, Dave Messerschmitt provided much of the inspiration through his Blosim program.

References.

[Bab84] R. G. Babb, "Parallel Processing with Large Grain Data Flow Techniques" *Computer*, **17(7)** July, 1984.

[Cov87] C. D. Covington, G. E. Carter, and D. W. Summers, "Graphic Oriented Signal Processing Language - GOSPL", ICASSP, Dallas, 1987.

[Cry74] T. Crystal, and L. Kulsrud, "Circus", CRD Working Paper, Institute for Defense Analysis, Princeton, NJ, Dec., 1974.

[Der69] M. Dertouzous, M. Kaliske, and K. Polzen, "On line simulation of block-diagram systems", *IEEE Trans. on Computers*, **C-18(4)** April, 1969,

[Gli84] E. P. Glinert and S. L. Tanimoto, "Pict: An Interactive Graphical Programming Environment", *Computer*, **17(11)**, November, 1984.

[Gol69] B. Gold and C. Rader, *Digital Processing of Signals*, McGraw-Hill, 1969.

[Hai85] D. J. Hait, "The BLOSIM Simulation Program", Master's Report, U. C. Berkeley, Nov. 11, 1985.

[Hen75] W. Henke, MITSYN - An Interactive Dialogue Language for Time Signal Processing, MIT Research Laboratory of Electronics memo. no. RLE-TM-1 Cambridge, MA, Feb. 1975.

[Ho88] W. H. Ho, E. A. Lee, and D. G. Messerschmitt, "High Level Data Flow Programming for Digital Signal Processing", VLSI DSP Workshop, IEEE ASSP Society, Monterey, November 1988.

[Kar65] B. Karafin, "The new block diagram compiler for simulation of sampled-data systems", in *AFIPS Conference Proceedings*, 27 pp. 55-61, Spartan Books, 1965.

[Kel61] Kelly, Lochbaum, and Vyssotsky, "A Block Diagram Compiler", *Bell Sys. Tech. J.* **40(3)** May, 1961.

[Kor77] G. Korn, "High-speed block-diagram languages for microprocessors and minicomputers in instrumentation, control, and simulation", *Computers in Electrical Engineering* **4** pp. 143-159, 1977.

[Lee87a] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers*, January 1987, C-36(2).

[Lee87b] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow" *IEEE Proceedings*, September, 1987.

[Lee89] E. A. Lee, "Programmable DSP Architectures, Part II", *ASSP Magazine*, January, 1989.

[Mes84a] D. G. Messerschmitt, "A Tool for Structured Functional Simulation" *IEEE Journal on Selected Areas in Communications*, SAC-2(1), January, 1984.

[Mes84b] D. G. Messerschmitt, "Structured Interconnection of Signal Processing Programs" *Proceedings of Globecom 84*, Atlanta, GA, Dec., 1984.

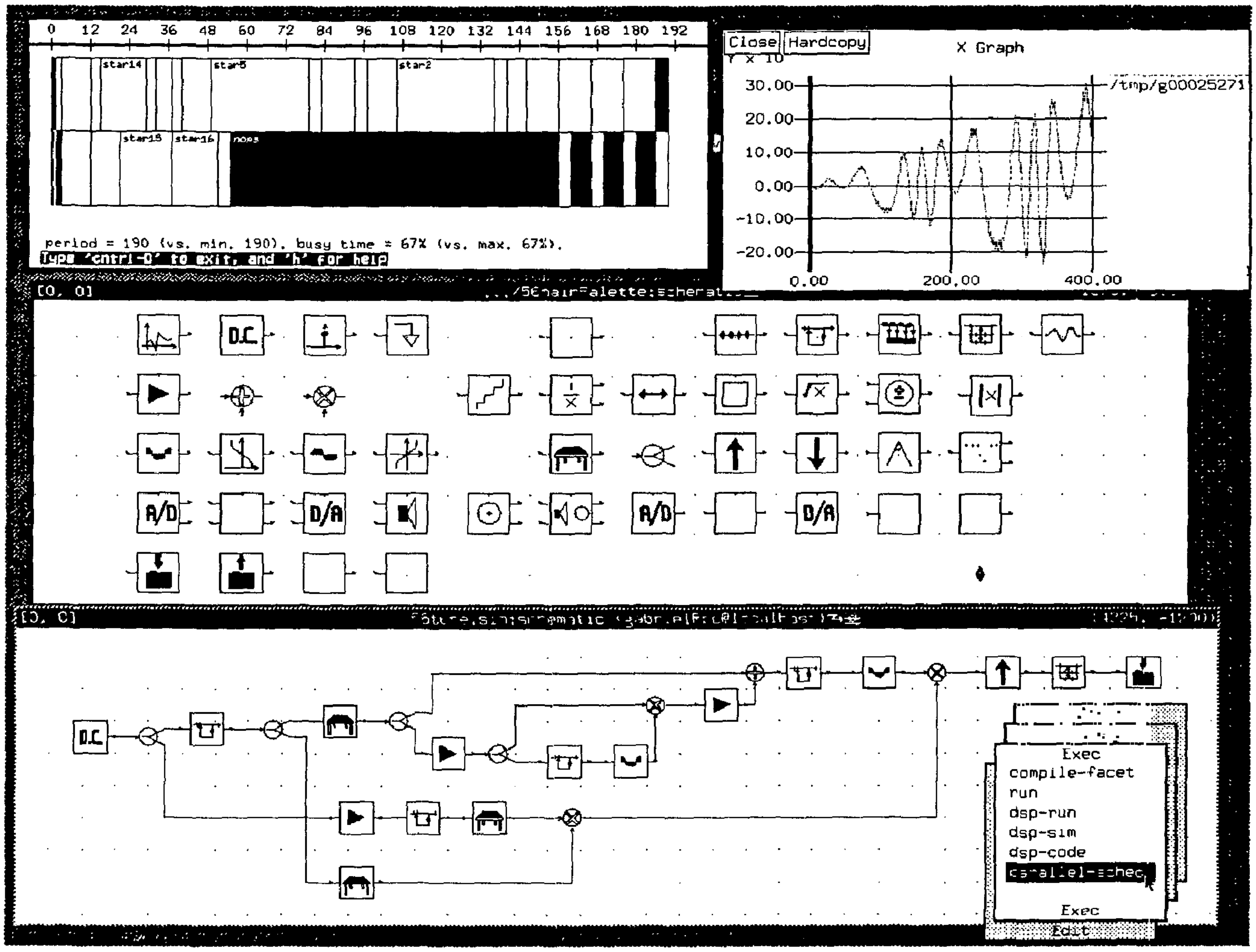
[Sha87] K. S. Shanmugan, G. J. Minden, E. Komp, T. C. Manning, and E. R. Wiswell, "Block-Oriented System Simulator (BOSS)", Telecommunications Laboratory, University of Kansas, Internal Memorandum, 1987.

[Sch85] D. A. Schwartz, "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs", Georgia Institute of Technology Technical Report DSPL-85-2, (PhD Dissertation) July 1985.

[Sch86] D. A. Schwartz, T. P. Barnwell, III, "Cyclo-Static Solutions: Optimal Multiprocessor Realizations of Recursive Algorithms", in *VLSI Signal Processing*, IEEE Press, 1986.

[VLS86] VLSI/CAD Group, *Thor Tutorial*, Stanford University, Stanford, CA, 1986.

[Zis86] M. A. Zissman, G. C. O'Leary, and D. H. Johnson, "A Block Diagram Compiler for a Digital Signal Processing MIMD Computer" DSP Workshop Presentation, Chatham, MA, October 1986.



```
period = 190 (vs. min. 190). busy time = 67% (vs. max. 67%).
Type 'cntrl-D' to exit, and 'h' for help
```

Figure 1. A Gabriel screen showing an application in the bottom window, a palette of functional blocks in the middle window, a two processor schedule at the upper left, and a signal display at the upper right. The application at the bottom (an FM-based music synthesis system) is built using blocks from the palette immediately above it. Code is generated and run in real time on a Motorola DSP56001 system. The menu at the bottom is the Gabriel command menu.