A Digital Signal Processor and Programming System for Parallel Signal Processing

David E. Van den Bout

Center for Communications and Signal Processing
Department of Electrical Computer Engineering
North Carolina State University

June 1987

CCSP-TR-87/6
# Table of Contents

Chapter 1. Introduction ................................................................. 1

Chapter 2. Background ................................................................. 8

Chapter 3. My Thesis ................................................................. 46

Chapter 4. Algorithm Analysis ....................................................... 51

Chapter 5. Interprocessor Communications ...................................... 69

Chapter 6. The DSP Architecture .................................................. 90

Chapter 7. The PaLS Automated Programming System ....................... 127

Chapter 8. Conclusions and Areas for Further Work ....................... 220

References ................................................................................. 238

Appendix A. The Axe Hardware Simulation Language ...................... 248

Appendix B. Prediction of Simulated Annealing Curves for Graph Partitioning .... 278

Appendix C. The Approach to Thermal Equilibrium ......................... 299
This thesis describes an \textit{integrated assault} upon the problem of designing high throughput, low cost digital signal processing systems. The dual prongs of this assault consist of:

- the design of a digital signal processor (DSP) which efficiently executes signal processing algorithms in either a uniprocessor or multiprocessor configuration,
- the PaLS programming system which accepts an arbitrary algorithm, partitions it across a group of DSPs, synthesizes an optimal communication link topology for the DSPs, and schedules the partitioned algorithm upon the DSPs.

The results of applying a new \textit{quasi-dynamic} analysis technique to a set of high-level signal processing algorithms were used to determine the uniprocessor features of the DSP design. For multiprocessing applications, the DSP contains an \textit{interprocessor communications port} (IPC) which supports simple, flexible, dataflow communications while allowing the total communication bandwidth to be \textit{incrementally allocated} to achieve the best link utilization. The net result is a DSP with a simple architecture that is easy to program for both uniprocessor and multiprocessor modes of operation (as was verified using the newly developed hardware simulation language, Axe).

The PaLS programming system simplifies the task of parallelizing an algorithm for execution upon a multiprocessor built with the DSP. PaLS differs from many other automatic programming systems in that it can map algorithms onto a group of processors while \textit{simultaneously deriving the interconnection topology for the processors}. This is possible because PaLS incorporates a model of computation which accounts for the costs of partitioning, communication link assignment, and scheduling. PaLS uses the method of simulated annealing coupled with the new technique of \textit{annealing curve prediction} to adaptively control the optimization of the algorithm mapping.
CHAPTER 1

Introduction

The uses of signal processing are ubiquitous within our society, from the synthetic voice which reminds us to buckle our seatbelt to the enhanced pictures beamed to Earth from the Voyager spacecraft circling Jupiter. Before World War II, all signal processing was done in the analog domain (i.e. continuous-time, continuous-signal). With the advent of integrated circuits in the 1960's, many applications moved from the analog to the digital domain (i.e. discrete-time, quantized-signal). Digital techniques permit greater precision, repeatability, signal-to-noise ratio, and flexibility than their analog counterparts [Alle85]. Programmable digital signal processors (DSPs) became widely available in the 1970's as large scale integration (LSI) was introduced. Even greater flexibility is available to the user of these devices since their operation can be tailored merely by changing the signal processing program.

As usually happens, such benefits are offset by disadvantages. In the case of digital signal processing, the bandwidth of the signals which can be processed is much smaller than for analog techniques. Programmable DSPs can usually be applied to voice-frequency signals ($\approx 10^4$ Hz) [Kawa80], while special-purpose digital systems can handle higher frequencies ($\approx 10^6$ Hz) [Lind85]. Analog systems have much higher throughput, and can process signals of ultra-high frequency ($\approx 10^{12}$ Hz).

The operating physics of analog devices makes them faster for many common signal processing tasks. For example, determining the response $y(t)$ of a linear system to a given input $x(t)$ is expressed mathematically as:

$$y(t) = \int_{-\infty}^{\infty} h(t - \alpha)x(\alpha)d\alpha$$

(1.1)
where \( h(t) \) expresses the impulse response of the given analog system. This *convolution* operation is a natural response of many types of analog systems. The digital form of Eqn. (1.1) is:

\[
y_k = \sum_{l=0}^{N-1} h_{k-l} x_l
\]

(1.2)

where \( y_k, h_k, \) and \( x_k \) are the discrete, digital counterparts of the previously described analog signals. As can be seen, determining \( y_k \) for some value of \( k \) requires \( N \) multiplications and \( N - 1 \) additions. This digital form of convolution is not a natural operation of digital systems, but must be explicitly provided for with hardware for performing multiplication and addition. Special-purpose digital systems can use many adders and multipliers in parallel to compute Eqn. (1.2), but to even approach the speed of analog convolution requires a large amount of hardware. Programmable DSPs, which typically have only one multiplier, are much slower than such hardware intensive systems.

The flexibility and programmability of digital systems also exacerbates their throughput problem. New algorithms are being proposed which could not readily be performed with analog components but are easily programmed on digital hardware. These algorithms (such as linear system solution and eigenvalue decomposition) require even greater computational resources.

In order to expand the range of applicability of programmable DSPs, the speed at which they execute signal processing algorithms must be increased. An increase in DSP speed can be achieved by:

- increasing the speed of the circuit technology,
- creating signal processing algorithms with reduced computational requirements,
- enhancing the DSP architecture,
- using multiple DSPs to perform computations in parallel.
The speed and density of silicon technology has steadily increased over time, and new technologies such as gallium-arsenide are becoming commercially feasible [Milu86]. However, quantum limits will eventually be reached beyond which further improvements are impossible [Mead80]. In any event, improvements in material technology can usually be applied independently of the other speed-up methods mentioned. Therefore, the effects of improved technology will not be considered here.

Algorithms with reduced computational requirements (such as the Fast Fourier Transform (FFT) and Winograd FFT [Wino78]), can make digital signal processing practical for certain applications. However, these procedures are not easily discovered, nor do they possess the usual simplicity of the majority of general-purpose signal processing algorithms. Therefore, this speed-up technique will not be expanded upon further.

Enhancing the architecture of a general-purpose computer in order to increase its performance for specific algorithms is a common technique. The execution speed of an algorithm can be increased as much as 1000% [Abd-74], although gains of 10%-50% are probably more common. In the same manner, the architecture of a programmable DSP can be altered to efficiently support common digital signal processing operations. The architectural enhancements will usually consume some additional circuit area and power, and since these two quantities are not in unlimited supply, a careful cost/benefit analysis is needed. Still, this is a likely area in which to improve the performance of DSPs.

Executing signal processing software with multiple processors offers the best hope for massive performance increases. The throughput of the total system can be increased merely by adding more processing elements while avoiding the fundamental physical limits encountered when trying to improve the circuit technology. The need to develop algorithms with reduced computational needs is replaced with the (slightly easier) job of creating algorithms consisting of many concurrent tasks. Ideally, these changes make it possible to increase the
total system speed indefinitely, rather than just the 10%-50% performance improvement achieved from enhancing the architecture of a uniprocessor.

The appraisal of parallel processing given above is overly optimistic from a practical point of view. Typical parallel processing systems have saturated their performance at 2-3 times the speed of an individual processing component [Fish83]. Such systems are usually victims of communications bottlenecks. In effect, a significant portion of the system processing time is spent passing data between the individual computing elements. In the worst case, the time spent servicing this communication overhead cannot be used to perform useful computations, thereby reducing the throughput of the system. Therefore, designers have spent considerable effort in developing parallel systems which minimize this overhead.

Probably the simplest multiprocessor network consists of a set of processing elements connected via a single global bus (Figure 1.1). The bus is used to transfer data and intermediate results between processing elements. Because the bus can handle only one data transfer at a time, algorithm execution slows when multiple processors simultaneously attempt to transfer data. Thus, while a single global bus is simple and easy to implement, bus contention can decrease the total system throughput.

At the opposite extreme, a bus can connect each processing element (Figure 1.2). With such a full interconnect, each processing element may transfer data without fear of conflicting with another element. Unfortunately, the number of individual buses required for $N$
processors is $\frac{1}{4}(N^2 - N)$, which becomes unreasonably large (and costly) as $N$ increases.

In order to reduce bus contention without building an extraordinary number of communication links, mesh-connected (Figure 1.3) and hyper-cube (Figure 1.4) multiprocessor networks have been built [Patt84]. These networks provide multiple buses (easing contention problems) but do not connect each pair of processing elements (lowering the implementation cost). A disadvantage of such networks is that data transferred from one processor to another may have to pass over several network links, thus increasing the transfer time (in the cases shown, the worst case delays are $O(\sqrt{N})$ and $O(\log_2 N)$, respectively).

Some of the possible tradeoffs between implementation cost and communication delays have been illustrated with the previously described systems. Because different signal processing algorithms have different rates and patterns of data flow, no static multiprocessing topology is universally optimal in terms of total system throughput. Such universality might be achievable with a dynamically reconfigurable multiprocessor network, but the cost of such
generality is prohibitive (so far). Therefore, an algorithm must be mapped to a fixed system architecture such that the communication overhead is minimized while the total computational load is evenly spread amongst the processing elements. The mapping process typically involves partitioning the algorithm into tasks for each system element to work on, and then scheduling the tasks such that the combined actions of the individual processors occur in a sequence which produces the correct results. Algorithm mapping is usually done manually.
by the programmer and requires much time, effort, and knowledge of the underlying hardware architecture. Some headway has been made in automating this process for simple and/or regular multiprocessor architectures and algorithms such as those found in the realm of digital signal processing [Lin85].

To recap the high points of the previous discussion:

- Digital signal processing can be more widely applied by increasing the throughput of programmable DSPs.

- The throughput of a signal processing system can be increased by 1) optimizing the architecture of a DSP for commonly found signal processing operations, and 2) creating networks of these DSP elements to perform the algorithm in parallel.

- Circuitry constraints restrict the possible optimizations of a DSP. Circuitry considerations also limit the number of buses in a multiprocessor system, leading to communication bottlenecks which decrease the system performance.

- Even given a good DSP architecture and network, mapping algorithms to a multiprocessor is difficult.

The next section provides a survey of how previous commercial vendors and academics have approached the problem of increasing DSP speed.
CHAPTER 2

Background

1. Introduction

This chapter will survey several attempts by others to provide high-speed DSPs and digital signal processing systems, along with software systems for mapping algorithms unto such architectures. The various systems which will be examined are listed in Table 2.1.

2. Single-Chip DSPs

Single-chip digital signal processors usually operate *stand-alone* with no interaction with a host computer. A typical system consists of:

- a DSP chip,

| Table 2.1: Digital Signal Processing Systems |
|-----------------|-----------------|-----------------|
| **Class**       | **Example**     | **Developer**   |
| Single-Chip DSPs| RSP             | International Business Machines |
|                 | TMS320C25      | Texas Instruments |
|                 | ADSP-2100      | Analog Devices   |
|                 | DSP56000       | Motorola         |
|                 | DSP32          | Western Electric |
|                 | μPD77230       | NEC              |
| Systolic Arrays | PSC            | Carnegie-Mellon University |
|                 | WSAP           | University of Southern California |
| Array Processors| MSSP           | Fairchild        |
|                 | MPP            | Goodyear Aerospace |
| Dataflow Systems| μPD7281        | NEC              |
|                 | DFSP           | Helsinki University of Technology |
| MIMD Systems    | BRACELET       | North Carolina State University |
|                 | MUSEC          | AT&T             |
| Silicon Systems | Macorells      | University of California |
|                 | FIRST          | University of Edinburgh |
• some random access memory (<64 Kbytes),
• an analog → digital converter (for acquiring input data from the environment),
• and a digital → analog converter (for sending results to the environment).

Even these small systems possess sufficient resources to perform such tasks as digital filtering, adaptive equalization, echo cancellation, speech synthesis, etc. The architectural features which provide the necessary speed for such applications will be seen as each DSP is examined.

2.1. Research Signal Processor (RSP)

The IBM Research Signal Processor architecture is shown in Figure 2.1 [Mint83]. Five functional subunits are discernible:

• Instruction fetch and sequence control unit,
• Data store address generation unit,
• Data store access unit,
• Arithmetic and logic unit (ALU),
• Input/output and control unit.

The instruction sequencer fetches instructions from external memory. This instruction transfer occurs over a separate bus than does data transfer, thus preventing contention between instruction and data accesses. This feature, along with a 4-stage instruction execution pipeline, increases the total speed of the RSP by permitting several operations to occur in parallel. While it might be thought that a 4-stage pipeline would need to be continually flushed due to program breaks, the predictable, repetitive nature of most signal processing algorithms makes this a relatively infrequent operation.
Figure 2.1: The Research Signal Processor
The data store address generation and access units work in tandem to transfer data between the RSP and external memory. The address generator performs direct, offset, indexed, and masked addressing, permitting data memory to be accessed in a very flexible manner. In particular, masked addressing allows the memory to be addressed with modulo-2^n addresses. This supports the first-in/first-out (FIFO) data structures common in signal processing algorithms. The same registers which are used to form data addresses can also serve as loop counters, thus providing hardware support for iterative operations. Unfortunately, there is only one bus to the data memory. This bus can become a bottleneck during such operations as digital convolution, since two data values (h_{k-1} and x_i) must be fetched and multiplied.

The arithmetic logic unit inputs 16-bit fixed-point operands, but permits accumulation of sums up to 32-bits wide. This is necessary to preserve precision during calculations. The ALU provides facilities for adding, subtracting, shifting, and logical functions. Interestingly, no hardware multiplier is provided. This function is performed in software using a specialized shift instruction. The large size of the multiplier circuit and the presence of reduced computation algorithms (e.g. the Winograd FFT) were responsible for this design feature. As was stated previously, such algorithms do not always exist for particular applications, so IBM has since developed DSPs with full hardware multipliers.

The I/O and control unit operates in conjunction with an external SPIO to provide communication with a host or with other RSPs. Communication can occur via a dataflow protocol or through the use of a global memory. The dataflow protocol allows multiple RSPs to be built into parallel and cascade systems for executing signal processing algorithms with minimal amounts of external hardware and software overhead [Mint81]. Alternatively, communications through a global memory permit the implementation of systems which undergo mode changes (i.e. infrequent but radical changes in the processing profile). The RSP is one of
the first signal processing chips which operates in a stand-alone mode and also can be used to build systems containing many RSPs.

Software support for the RSP includes a high-level assembler, linker/loader, simulator, debugger, and interface to the high-level language PL/I. Nothing has been reported, however, of automated tools for creating multiprocessor networks of RSPs.

2.2. The TMS320C25

The Texas Instruments TMS320C25 architecture (Figure 2.2) shares some similarities with the RSP [Fran86]. Both are built around a Harvard architecture with separate instruction and data buses. Both work with 16-bit fixed-point integer values and accumulate 32-bit results. Like the RSP, the TMS320C25 can also be built into multiprocessing systems through the use of arbitrated global memory. Direct links between TMS320C25s are also possible using their built-in serial I/O links, but the speed and flexibility of these links are limited.

The TMS320C25 has several important differences from the RSP. The most significant is the inclusion of a full hardware multiplier for generating 32-bit fixed-point products within a single instruction cycle. The inclusion of a multiplier eases the coding of fast, general-purpose signal processing algorithms. The single cycle speed of the multiplier would be wasted, however, if a multiplicand and multiplier could not be supplied at the same rate. To this end, the TMS320C25 incorporates two RAMs (as well as a program ROM which can store constant coefficients) which can simultaneously provide operands to the multiplier over the program and data buses. Since the program bus cannot simultaneously transfer data and instructions, facilities are provided to allow repetition of a single instruction. This looping ability relieves the need to receive a new instruction during every cycle, thus freeing the program bus to be used as another data bus. An instruction can be repeated up to 256 times in the TMS320C25 by loading the 8-bit loop counter with the appropriate value.
Figure 2.2: The Architecture of the TMS320C25
In order to easily access operands as well as provide instructions with sufficient power to take advantage of the single instruction looping capability, the TMS320C25 contains a set of eight auxiliary registers which can be used as pointers to memory. These registers can be incremented or decremented (either by 1 or by the value stored in auxiliary register AR0) in parallel with the operations of the data ALU and multiplier, thus eliminating memory addressing delays and permitting fast processing of arrays of data. *Bit-reversed* addressing is also provided to facilitate the programming of FFT routines. However, the TMS320C25 does not possess an equivalent of the RSP masked addressing mode. Instead, the TMS320C25 supports specialized instructions which simultaneously operate upon and move data in internal memory. These operations are useful for implementing general-purpose FIFOs since there is no restriction that the FIFO be of length $2^n$ as in the RSP. The disadvantage of this technique is that, while the TMS320C25 readily handles short delay line FIFOs (such as are used in *biquad filter sections*), each element of a long delay line must be individually moved. If no calculations are performed on a significant portion of the FIFO elements, these TMS320C25 instructions are much less efficient than masked addressing.

The majority of programs for the TMS320C25 are coded *in-line* rather than using looping constructs and conditional branching. This is done to remove the overhead involved in doing the loop counter updates (for multiple instruction loops, only) and checking array boundaries. This coding technique creates lengthy programs which execute in minimum time since every operation performs a useful calculation. However, the length of the program can exceed the addressable program space of the TMS320C25 for some applications (e.g. 1024-point FFT). In these cases, instructions composing the outer program loops (which are not as time-critical) must be recoded as software controlled loops.

The software support for the TMS320C25 consists of a standard assembler, linker/loader, simulator, and debugger. Some macro packages also exist, especially for expanding iterative
constructs into in-line code statements. There is no software provided to assist in programming multiprocessors built from several TMS320C25s.

2.3. The ADSP-2100

The architecture of the ADSP-2100 programmable signal processor is shown in Figure 2.3 [Roes86]. Once again, a Harvard architecture is employed with separate instruction and data buses of widths 24 bits and 16 bits, respectively. A 14-bit address bus is paired with each data bus, allowing the ADSP-2100 to access up to 32K bytes of external memory.

Data computations are performed upon 16-bit fixed-point data using three separate arithmetic units: a general purpose 16-bit ALU, a shifter, and a 16x16 bit multiplier coupled with a

![Figure 2.3: The Architecture of the ADSP-2100](image)
40-bit accumulator (capable of performing $2^{40} - 2^{32} - 2^{8} = 256$ multiplies and accumulations before overflow occurs). These units operate in parallel and are connected by a local bus (the R bus) to avoid congestion of the main bus while passing intermediate results.

Delays in delivering operands to the arithmetic units are avoided by using two independent data address generators which support indexed, bit-reversed, and modulo addressing modes.

The absence of RAM within the ADSP-2100 requires the use of external memory for storing data. Two operands are fetched by using both the data and instruction buses and their associated address buses. Naturally, dual data accesses cannot be performed if the instruction bus is being used to fetch the next instruction. In order to reduce the congestion on the instruction bus, a cache containing the last 16 executed instructions is built into the ADSP-2100. Small program loops can be held within this cache, thus eliminating the need to fetch instructions from external memory. Of course, dual data fetches cannot be performed the first time through a program loop as the required instructions are not yet in the cache. Program execution speed increases on subsequent repetitions.

Nested, multi-instruction loops are supported in the hardware of the ADSP-2100. Two four-level stacks are provided for storing loop counter values and the ending addresses of loops. Upon entering a loop, the current program counter value is pushed onto the normal stack, and the initial count and address of the end of the loop are pushed onto their respective stacks. A comparator circuit detects when the end address of the loop is reached and decrements the top entry on the loop counter stack. If the top counter becomes zero, the counter and end address stacks are popped and the loop is exited. Otherwise, control returns to the beginning of the loop by loading the program counter from the top of the program counter stack.
Multiprocessor systems composed of several ADSP-2100s can be built by using the bus request and grant signals to manage access to a global memory. The ADSP-2100 also provides a background register for each internal register to facilitate rapid storing and restoring of context. These features combine to support the creation of multiprocessors which are very responsive in the performance of mode changes.

Due to the complexity and flexibility of its operations, the ADSP-2100 is programmed using an algebraic code instead of instruction mnemonics. No support is available for programming multiprocessing systems built with the ADSP-2100.

2.4. The DSP56000

The DSP56000 (Figure 2.4) [Klok86] shares several features with the ADSP-2100 previously presented:

- Instructions are 24-bits wide.
- Two independent address generators are provided for performing dual data fetches. Indexed, bit-reversed, and modulo addressing are all supported.
- Up to 15 nested, multi-instruction loops are supported in hardware using loop counter and end address stacks.

The DSP56000 also has significant differences from the ADSP-2100. First of all, the DSP56000 performs all data computations on 24-bit fixed point numbers rather than 16-bit data. This increases the dynamic range from 96 dB to 144 dB.

The single-cycle multiplier uses two 24-bit data values to generate 48-bit results. Accumulation of these results is done using 56-bit accumulators. As is in the ADSP-2100, the eight extra accumulator bits permit up to 256 multiply/accumulations to occur before overflow is possible.

Two internal data memories are included in the DSP56000, along with separate address and data buses for each. This facilitates the movement of data to the arithmetic unit without
Figure 2.4: The Architecture of the DSP56000
requiring the use of the instruction bus (as was done in the TMS320C25 and ADSP-2100). Instructions or data may also be accessed from external memory using the single set of external data and address buses.

There are several features which support multiprocessing in the DSP56000:

- Multiprocessors built around global memory are made possible by the inclusion of bus request and grant signals in the DSP56000 pinout. Such multiprocessors will require significant external hardware to manage the interface to the global memory.
- A DMA-like interface is provided which allows a host processor to control the operation of the DSP56000 as well as transfer data back and forth.
- A serial interface provides a slow, inexpensive link between multiple DSP56000s.

The DSP56000 is programmed using mnemonic assembly code. Standard development tools include an assembler, simulator, and a high-level language compiler. No support for programming multiprocessors is available.

2.5. The DSP32

The DSP32 (Figure 2.5) contains 5 primary subunits [Hays85]:

- a data arithmetic unit (DAU),
- a control arithmetic unit (CAU),
- several ROM and RAM memories,
- a serial I/O unit (SIO),
- a parallel I/O unit (PIO).

The DAU of the DSP32 represents its most dramatic departure from the previous examples. The DAU is capable of multiplying and adding 32-bit floating-point data values (24-bit mantissa, 8-bit exponent) rather than fixed-point integers. This eliminates the need for scaling data values and increases the speed of algorithm execution. Code development is also
Figure 2.5: The Architecture of the DSP32
Because the DAU multiplier and adder are pipelined for maximum speed, it is difficult to perform logic and control operations within this section. Instead, these functions are handled in the CAU. The CAU is a full 16-bit fixed-point computer which generates addresses for the memory and controls the flow of execution in a signal processing program. Any one of 21 CAU registers may be used to address the data memory to obtain operands for the DAU. The selected register is simultaneously modified by adding the contents of another register to it. This operational sequence in combination with the CAU capability to perform general computer type functions permits the synthesis of many flexible addressing modes.

The internal memory of the DSP32 contains enough ROM for a 512 instruction program, as well as two RAM banks each capable of storing 512 floating-point numbers. The presence of two RAM banks allows fast convolution as in the TMS320C25. However, in the DSP32 these RAM banks form an interleaved memory system, so successive accesses to the same bank leads to a decrease in performance due to increased memory waiting time.

The DSP32 I/O system contains a parallel port for use in passing data to and from a host. There is also a serial interface which is primarily intended for interfacing to codecs, but can provide a link in multiprocessor configurations (as in the TMS320C25).

The DSP32 is programmed with sets of equations expressed in a 'C'-like syntax, much as the ADSP-2100 is. Partitioning of programs to run on multiple DSP32s is done manually.

2.6. The μPD77230

Like the DSP32, the μPD77230 (Figure 2.6) operates upon 32-bit floating-point data rather than fixed-point values [Eich86]. Other than this, however, the two processors are quite different.
Figure 2.6: The Architecture of the μPD77230
Two internal data RAMs are built into the μPD77230. These RAMs are non-interleaved, so their data may be accessed without excessive delays. Each RAM is also provided with an address generator which supports indexed and modulo addressing. (Bit-reversed addressing for FFT computations is performed using the main ALU.) Numerous internal buses exist to carry the data from the RAMs to where it is used. This prevents congestion of the main 32-bit data bus within the μPD77230.

The floating-point multiplier accepts two 32-bit values (24-bit mantissas and 8-bit exponents) and generates a 55-bit result (47-bit mantissa and 8-bit exponent). The associated floating-point ALU operates upon 47-bit fixed-point or 55-bit floating-point data and stores the result into a set of working registers. The ALU also contains a shifter which can be used to shift fixed-point data or can be driven by the exponent ALU so as to align floating-point values for addition or subtraction.

The μPD77230 instruction sequencer provides a single loop counter for hardware control of repetitive instructions. A bit in every instruction determines whether this counter will be decremented or not. When the counter reaches zero, the following instruction is skipped and the loop is terminated. Thus, while the μPD77230 has hardware support for multiple instruction loops, some software overhead is still necessary.

The μPD77230 can be used to create multiprocessors through the use of either its serial or parallel I/O ports. Low-speed, low-cost multiprocessors are obtained by cascading several μPD77230s via their serial ports. The parallel host interface may be used for higher speed systems.

The μPD77230 is programmed using groups of mnemonic operations to compose an entire 32-bit instruction. An assembler and simulator are available, but no support exists for creating and programming multiprocessor configurations.
3. Systolic Arrays

Systolic arrays [Kung79] attain high processing speeds by pipelining computations within a regularly connected network of relatively simple processors. All communications between processing elements are local and occur simultaneously as determined by a global clock which controls the "beating" of the total system. The pipelining of systolic systems provides high speed while the short, regular interconnections simplify implementation. However, most proposals for systolic arrays entail systems dedicated to certain vector functions (e.g. multiplying matrices). The expense of designing a system for such a narrow application has hindered their implementation.

A natural solution to the cost problem is to develop programmable systolic array elements. The two examples described below represent slightly different attempts at this.

3.1. The Programmable Systolic Chip (PSC)

The Programmable Systolic Chip architecture (Figure 2.7) consists of 5 subunits connected over a set of three buses:

- microinstruction storage and a sequencer,
- an ALU,
- a multiplier/accumulator (MAC),
- a set of three ports,
- and a register file.

User-defined horizontal microinstructions are stored in the microinstruction RAM. Horizontal instructions allow the units of the PSC to be used in parallel to obtain minimal execution times for algorithms. The microsequencer is also able to perform multi-point branches which permits complex control flows without a large amount of program overhead.
The actual data manipulations occur in the ALU and MAC. Each of these units operates on 9-bit operands: 8-bits of data and a 1-bit tag which indicates some property of the data. (The data word size limitation was brought about by technology limits at the time the PSC was built.) The ALU and MAC contain logic and storage to simplify the programming of multi-word operations (such as 32-bit arithmetic). The ability to operate these two units in parallel as well as the natural pipelining of systolic systems allows high throughput systems to be built with the PSC.

The register file stores operands for the ALU and MAC. Alternatively, operands may be fetched from another PSC through one of the three interprocessor ports. These three two-way ports allow linear, hexagonal, and tree-like systolic structures to be implemented.

The PSC is programmed with a form of microassembly language. Formulation of algorithms for systolic arrays will be discussed in a later section.

3.2. Wavefront Systolic Array Processor (WSAP)

The Wavefront Systolic Array Processor (Figure 2.8) contains the same major components as the PSC described previously [Kung84]. The datapath width has been increased to 32-bits to speed the execution of signal processing calculations. An additional fourth port has also been added to facilitate the implementation of mesh-connected two-dimensional arrays.
The most significant alteration, however, concerns the philosophy of interprocessor communication. All data transfers in standard systolic arrays occur simultaneously as directed by the global clock. *Wavefront systolic arrays,* however, pass data between elements as it becomes available in a dataflow type of protocol. This has several advantages:

- The systolic elements need no longer run in *lock-step.* Therefore, some sections of the array may be performing a different algorithm than other sections and true multiprocessing can be achieved.

- Array elements with indeterminate instruction cycle times are easily accommodated.

- Wavefront systolic arrays are easier to program since the input of data and generation
of results can be done without concern for the global timing scheme.

3.3. Systolic Array Programming

Generating a systolic array which performs a given application requires the user to specify the following:

• an interconnection pattern for the array elements;

• a timing format which insures that the correct data reaches the appropriate processors at the time it is needed;

• the computational steps performed by the individual array elements.

An algorithmic procedure has been developed for transforming the calculation intensive inner loops of an algorithm into a systolic array [Mold83]. The method applies transformations to the data dependencies of a selected algorithm in order to derive a space and time ordering.

In more detail, suppose the following nested loops were to be implemented as a systolic array:

\[
\text{DO 10 } l^1 = b^1 \text{ to } u^1 \\
\text{DO 10 } l^2 = b^2 \text{ to } u^2 \\
\text{...} \\
\text{DO 10 } l^n = b^n \text{ to } u^n \\
S_1(l) \\
S_2(l) \\
... \\
S_N(l) \\
10 \text{ CONTINUE}
\]

Here, \( \overline{l} = \{l^1, l^2, \ldots, l^n\} \) is the set of loop indices and \( \{S_1, S_2, \ldots, S_N\} \) are the statements executed within the innermost loop. The set of all valid possible combinations of the loop indices is:
When the loop is executed, the elements of $L^n$ are lexicographically ordered by the manner in which the source code is written. This induced ordering can be modified to suit the algorithm for systolic arrays while maintaining the data dependencies for the assignment statements $\{S_1, \ldots, S_N\}$.

Two indexed variables $X_{f(\bar{i}_1)}$ and $Y_{g(\bar{i}_2)}$ generated in statements $S(\bar{i}_1)$ and $S(\bar{i}_2)$, respectively, are data dependent if:

- $\bar{i}_1 < \bar{i}_2$
- $f(\bar{i}_1) = g(\bar{i}_2)$
- $X_{f(\bar{i}_1)}$ is an input variable to $S(\bar{i}_2)$

An algorithm has several data dependence vectors $\bar{d} = \bar{i}_2 - \bar{i}_1$, and the mapping procedure works best if these are constant (as occurs in many signal processing algorithms). The set of dependence vectors imposes an ordering $R$ on $L^n$. A transformation $T$ is found which performs the following mapping:

$$ T: <L^n, R> \rightarrow <L^n, R_T> $$

where the data dependencies of the transformed set are selected by the user. If the user specifies a set of dependencies $\Delta$ for an algorithm with an original set of dependence vectors $D$, then the transformation is found from the set of diophantine equations:

$$ TD = \Delta $$

This transformation is guaranteed to exist if the user satisfies some restrictions on his choice of dependencies.

The transformation $T$ can be partitioned into two related transformations:
The $\Pi$ transformation governs the timing relationships of the systolic array, while $S$ controls the spatial arrangement. It is desirable to minimize the dimensionality of $\Pi$ and maximize that of $S$ since this will lead to maximum concurrency by replication of systolic array elements.

The structure of the individual systolic elements can be determined by applying similar transformations to the bit-level data dependencies of the individual statements.

The technique outlined above can map nested loops with constant data dependency vectors into a systolic array. A variation of the method allows a similar mapping into an array processor which supports global data transport between elements (this is precluded in systolic arrays). These methods are useful when applied to the most computationally intensive portion of signal processing algorithms, but do not address the problems involved in mapping the remaining, less-regular portions of the program.

4. Array Processors

Array processors are typically composed of two major functional units:

- a one or two-dimensional array of simple processing elements which are usually connected in a nearest-neighbor configuration;
- a standard computer which controls the operation of the processing array based upon commands it receives from a host computer.

All of the processing elements in the array perform the same operation in lock-step upon different pieces of data. Array processors are thus classified as single-instruction, multiple-data (SIMD) computers. Some control logic is placed in each element to allow it to remain quiescent during an instruction based upon the value of its local data, but execution of independent processes is not possible within an array. This is sometimes beneficial since this
eliminates the need to synchronize inter-process communication - a major source of communication delays and performance degradation. Unlike systolic arrays, each processing element has access to global data output by the control processor without having to ripple the data through several communication links. Thus, array processors can attain the highest throughput of any type of multiprocessor provided the application can be tailored to fit the restrictive communication and flow-control capabilities of the array.

Two examples of array processors will be examined: a linear array for speech processing and a mesh array for image enhancement.

4.1. Multi-Serial Signal Processor (MSSP)

The Multi-Serial Signal Processor (Figure 2.9) is a one-dimensional array of simple processing elements intended for use in processing speech signals [Lyon84]. A central controller is responsible for interfacing to the host computer and transferring data and instructions to the linear array of processing elements. These array elements are connected in a nearest-neighbor configuration so that data can only move left or right by one processor during an

![MSSP Architecture](image)

Figure 2.9: MSSP Architecture
instruction cycle.

The Serial Signal Processing elements (SSPs) are each composed of 2 limiters, 2 adders, 2 multipliers, and an ALU which operate upon 32-bit signed data in a bit-serial manner. A 64-word RAM with the capability of delivering 8 words during an instruction is also provided. This combination of multiple arithmetic units and simultaneous access to multiple data words gives each processing element a high throughput rate.

The SSPs communicate with one another over 1-bit wide serial links. Each element is provided with 5 input and 8 output links. While two input and output links per element would have been sufficient for a linear array, the extra communication paths are envisioned as being useful in implementing a tree-structured or two-dimensional processing system. All I/O between elements occurs in unison, thus eliminating the need for synchronization overhead and increasing throughput.

The processing elements do not have facilities for masking their operations or for indexed addressing of RAM data. A special ALU instruction is provided which can select between two data values based upon their sign, thus providing support for limiters, rectifiers, and conditional assignment. As for indexed addressing, this function is left to the array controller which determines the memory addresses accessed by the array elements. The controller itself is sparse in terms of its capabilities. No provision is made for looping constructs, so all loops are unrolled into in-line code.

Programming the MSSP can be done with high-level languages which are translated to provide an intermediate sequential assignment code. This code is then processed by a compiler which schedules the operations such that the correct results are produced in near-optimal time while accounting for the pipeline delays of the SSP elements. The high-level mapping of the application to the array is left to the user.
4.2. The Massively Parallel Processor (MPP)

The Massively Parallel Processor is a two-dimensional array processor intended for processing image data from satellites [Batc80]. An array control unit is responsible for all non-parallel program operations and for managing an array consisting of 16,384 elements organized into 128 columns of 128 processing elements each.

Due to the fact that many image processing algorithms operate upon 1-bit wide data (pixels), the processing elements have simple 1-bit wide data paths with a small amount of memory (Figure 2.10). Single-bit wide data paths exist between each element and its four neighbors. This allows the elements to combine the data they contain with that of their surrounding neighbors to enhance the quality of the image data. A 1-bit adder and a 16-function logic unit are provided for this task. Each processor also has a mask register which can be loaded and whose value will determine whether the associated element will participate in the next instruction.

![Figure 2.10: MPP Processor Element](image-url)
Special care was taken in the MPP to balance the I/O and computational capabilities. Each processing element contains a special register which allows it to shift data to its right neighbor while receiving data from its left neighbor. This facilitates the loading of new image data while processed data is being output. The MPP is also provided with a corner turning memory which formats the serial data from the satellite telemetry into two dimensional frames for the MPP array. These features insure that the processing array spend the majority of its time performing calculations instead of doing I/O.

Due to the specialized application of the MPP and its simple processing elements, most coding is done by hand in order to obtain highly optimized code.

5. Dataflow Systems

Dataflow systems perform the computations specified by dataflow graphs. A dataflow graph specifies the data dependencies for a group of calculations (Figure 2.11). Operands (called tokens) traverse the arcs of the graph and arrive at the inputs to the computational

![Figure 2.11: A Dataflow Graph](image-url)
units. When a computational unit has at least one token upon every input, it fires, consuming the input tokens and producing an output token. This output token is passed along to any succeeding computational nodes in the graph, possibly causing some of them to fire. In some dataflow models, data tokens enter a queue at the computation unit inputs, while in others the computation nodes are not allowed to fire unless their output arcs are free of tokens. Thus, the computation is self-regulating so that data dependencies are not violated.

Typical data flow systems can be bisected into the following sections:

**Computation Section:**

The calculations specified by the dataflow graph are executed by a set of processing elements. These elements may be as simple as those in systolic arrays and perform simple dyadic operations (*small-grain dataflow*), or they may be fully functional computers executing complex operations with many input tokens (*large-grain dataflow*).

**Control Section:**

A central controller schedules dataflow graph computations which possess all the necessary input tokens unto available processing elements. The central controller also collects the outputs from the computations and updates the inputs of any connected nodes.

Ideally, dataflow systems dynamically extract parallelism from a program via the firing of computational nodes when all input operands are received. This frees the user from having to partition and schedule his application program amongst the processing elements. Also, the use of dataflow systems seems natural for digital signal processing since many DSP algorithms are specified in terms of *signal-flow graphs*. Unfortunately, the need for the central controller to schedule all computations usually limits the maximum throughput of such systems.
Two dataflow implementations will be examined: a single-chip useful for building dataflow systems and a full dataflow signal processor.

5.1. The \( \mu \text{PD7281} \)

The \( \mu \text{PD7281} \) is a single-chip dataflow element (Figure 2.12) intended to be used in image processing applications [NEC85]. Up to 14 of these chips may be arranged in a ring connected by 16-bit buses (Figure 2.13). The input controller within each \( \mu \text{PD7281} \) examines the tokens which circulate upon the interprocessor bus. If the address of a token matches the processor ID, the token is entered in the internal pipeline. Otherwise, it is passed along to the next processor through the output controller.

![Figure 2.12: A Block Diagram of the \( \mu \text{PD7281} \)](image)

![Figure 2.13: A Dataflow System Built from \( \mu \text{PD7281s} \)](image)
The link table stores the interconnection pattern for the part of the dataflow graph being executed by that particular processor. Using the link table, if the flow controller determines that the incoming token has enabled the firing of a computational node, then the function table is accessed to retrieve the opcode which must be executed. The activated opcode, the new token, and any tokens previously stored in the data memory are placed in the data queue to await manipulation by the processing unit. The sequence repeats using the output token from the processing unit to trigger any operations dependent upon the previously fired computational node.

If the output from the processing unit does not trigger a node stored in the current processor, the address of the appropriate processor is appended to the token. The altered token is cycled through the pipeline to the output queue and is placed on the system bus by the output controller.

The \( \mu \)PD7281 implements fine-grain dataflow and provides good throughput if its internal pipeline can be kept full (7 tokens are needed for this). Many image processing applications mesh naturally with its internal and interprocessor pipelining. Algorithms without such a regular communication structure would suffer from the rigid interconnection scheme.

Programming the \( \mu \)PD7281 is done via assembly language. The user is required to manually break his program up between the the separate processing elements. Thus, while each \( \mu \)PD7281 dynamically determines parallelism present in its own piece of code during the execution of the algorithm, the user is required to specify the global concurrency.

5.2. The Data Flow Signal Processor (DFSP)

The Data Flow Signal Processor (Figure 2.14) is a large-grain dataflow system [Hart86]. Unlike the \( \mu \)PD7281 which processes computational nodes restricted to two or fewer inputs, the DFSP implements complex dataflow actors which can accept any number of inputs. Such actors could implement an entire matrix multiplication or FFT as a single computational node.
Figure 2.14: The Data Flow Signal Processor

The operation of the DFSP is similar to that of the $\mu$PD7281. An activity store holds the description of a dataflow graph along with information regarding the presence and absence of tokens upon the graph arcs. The fetch unit is triggered when a dataflow actor has sufficient input tokens to fire, and sends a description of the actor to an inactive member of the bank of processing elements. The fetch unit also triggers the data transfer unit such that the operands for the actor are retrieved from data storage. These operands pass through the output DMA controller and arrive at the triggered processing element.

Once the processing element has completed its assigned task, it signals the update unit which determines the correct destination for the generated result tokens. The result transfer unit controls the storage of the results into the data storage block through the input DMA controller. Once all the result tokens are stored, the associated processing element is marked as inactive so that it can be re-used. The result transfer unit also checks to see if the new tokens
have triggered any new actors. If so, it signals the fetch unit and the sequence begins again.

During operation, large blocks of data are typically passed to the processing elements of the DFSP. This is due to the complex operations performed by the dataflow actors, and the fact that processing elements receive copies of entire arrays rather than memory pointers. This causes congestion in the input and output DMA controllers. The coarse granularity of the dataflow operations can also reduce the available concurrency within a program.

Finer grained dataflow actors would allow more parallelism, but would also cause a bottleneck within the control section of the DFSP due to the increased rate of initiation and termination of the processing elements. Thus, while the DFSP frees the user from specifically identifying the parallelism within an algorithm, the need for the central controller to dynamically extract the concurrency can lead to decreased performance.

6. Multiple-Instruction Multiple-Data (MIMD) Systems

MIMD systems usually consist of a small number of processors of moderate to high complexity. Each processor runs its own program (multiple-instructions) on locally maintained data (multiple-data) independently of the others until some resource (usually data) is needed from another processor. Then, the network interconnecting the processors is used to transfer the needed information. This requires some control overhead for identifying the requested information and synchronizing the transfer. This overhead reduces the throughput of MIMD systems and lessens their attractiveness for signal processing applications.

Not many MIMD signal processing systems have been built. This section details a hybrid SIMD/MIMD multiprocessor and an MIMD machine with a reconfigurable interconnection network.
6.1. The BRACELET

The BRACELET system (Figure 2.15) is composed of the following components [Mill86]:

- an array of 4 TMS32010 DSP chips connected via a network of 16-bit buses,
- a master controller which controls the array,
- a switch to channel I/O through the array,
- a global memory and bus to allow the TMS320s to access common data.

The TMS32010 DSP chips which make up the processor array can execute in synchrony or they can *diverge*, run separate programs, and *merge* back into the SIMD mode. Thus, the speed of array processors can be achieved while retaining the flexibility of MIMD machines.

The bus network connecting the four slave DSP chips is a combination of a ring and a binary tree (this is more apparent on BRACELET networks with 8 or more slave processors). The network was originally designed for the fast accumulation of inner products in adaptive filtering algorithms, but can be used for general interprocessor communication, also. The 16-bit width of these buses permits fast transferral of data between processors, but also increases the cost of implementation. For an *N* processor BRACELET, *N* log₂*N* unidirectional buses are needed. Since the TMS32010 provides no multiprocessing support, the networking circuitry must all be provided externally, thus increasing cost and complexity.

The master controller transfers pipelined data in and out of the BRACELET array via the switch. Data can be broadcast to all the array processors through the global bus from either the master processor or the global memory. This facilitates programming adaptive filters where all the processors must be apprised of the error value after each iteration.

The BRACELET is programmed through a combination of TMS320 assembly language and macro calls which support processor communication and other common functions. It is the responsibility of the user to map signal processing algorithms onto the BRACELET archi-
Figure 2.15: The BRACELET Architecture
6.2. **Multiplex Service Circuit (MUSEC)**

The MUSEC system (Figure 2.16) is mainly intended as a testbed for parallel execution of speech processing algorithms [Knud84]. The system is used as follows:

**Algorithm specification:**

The user formats his algorithm as a *high-level signal flow graph*. This entails partitioning the task into a small set of relatively complex program modules. For example, an LPC speech processing application might be divided into modules for A-D conversion, pre-emphasis, windowing, autocorrelation, matrix solution, and FIR filtering. These program components pass processed signal values to one another during the execution of the algorithm.

**Coding:**

The detailed algorithm for each program module is coded in assembly for the AT&T DSP chip included in each *Digital Signal Processing Module* (DPSM). The interconnection of the modules is also specified.

**Downloading:**

Each compiled program module is downloaded to a separate DSPM through the *Master Controller* and the associated *B-Bus*. The interconnection list is loaded into the *Switch Controller* (SC) which sets up the initial configuration of DSPMs.

**Algorithm execution:**

As the DSPMs execute their respective programs, interprocessor communication is accomplished through the *switch* over 1-bit serial lines. Each DSPM has 1 output line and 1 input line. The switch connects the the DSPMs together via a crossbar-like scheme which prevents the individual links from interfering with one another. The
Figure 2.16: The MUSEC Architecture
links may be dynamically re-routed by the Switch Controller in response to a request from a DSPM. This permits a module to interface to two or more other modules.

The above system is simple and supports a model of communications similar to that of the actual application (i.e. signal flow graphs). The high-level modules do not permit maximum concurrency, however, and the 1-bit data links can create a communications bottleneck. Dynamic switching requests can also overload the switch controller.

7. Silicon Systems

A relatively recent method of creating signal processing systems is through the use of silicon compilers. Broadly stated, these compilers take as input a description of the algorithm and output the detailed schematic of an integrated circuit which executes the program. Silicon systems designed by these compilers can attain very high processing speeds since they are constructed to run one specific algorithm. Such devices once built, however, cannot easily be used to perform other functions.

7.1. Macrocells

The Macrocell Compiler [Pope84] builds a signal processing system for a user specified algorithm as follows:

Manual partitioning:

The user manually divides the algorithm such that it can be run using several processing elements. A design file is created describing the particular program for each processing element as well as the interconnection network. The designer also specifies some hardware parameters for the processing elements, such as the word width of their accumulators and ALUs.

Global layout:

The first pass of the Macrocell Compiler extracts the parameters for the processors
from the design file. A basic processor layout is then modified using this data in order to create specialized processors for each section of the partitioned program. The basic processor is composed of the following macrocells:

- program counter
- program storage
- address generator
- finite state machine
- ALU with I/O ports
- data storage

The size and presence of the above macrocells in the final system design is determined by the extracted parameters.

**Detailed layout:**

Once the construction of each specialized processor is determined, the second pass of the compiler determines the detailed circuit layout for the entire system. The layout is determined by the size of the macrocells included in the processing elements as well as the processor interconnection network.

### 7.2. The FIRST Silicon Compiler

FIRST is a silicon compiler specialized for the creation of bit-serial signal processing systems [Deny84]. The bit-serial approach slows the performance of arithmetic functions. This disadvantage is offset by the fact that bit-serial operations can be pipelined. The circuit elements used for serial arithmetic are also easy to construct in a compacted form.

FIRST is used as follows:

*Algorithm entry:*

The user enters his algorithm in the form of a *functional flowgraph*. This flowgraph is
quite similar to a data flowgraph, except the designer is required to synchronize data transfers between processing nodes by judiciously inserting delay elements. The designer can only use processing elements with direct hardware equivalents when constructing his flowgraph.

Circuit layout creation:

The FIRST compiler reads the flowgraph and inserts the appropriate piece of hardware for each program element. A detailed circuit schematic can then be created.
1. The Problems of the Past

A significant percentage of signal processing systems are dedicated to performing one algorithm as fast and as cheaply as possible. The architectures examined in the previous section fail to provide such a service because they are lacking one or more of the following qualities:

**Speed:**

A system dedicated to a specific algorithm must be 1) built quickly, 2) programmed quickly, and 3) execute the program quickly.

**Cost:**

A signal processing system must have a low cost for dedicated applications. This depends on the availability of inexpensive components that are easily interconnected and programmed.

**Flexibility:**

Flexibility of the entire system is not so important for dedicated applications as is flexibility of the components used to build it. It must be possible to take the basic system component and use it to build substantially different systems without a large expenditure of time or money.

Single-chip DSPs certainly provide low-cost signal processing, but they do so only for low-frequency applications. In order to increase their range of applicability, several DSPs must be combined to solve the problem in parallel. Low cost multiprocessor networks are
obtainable by using the serial ports on each DSP, but the communication delays are large and the serial ports do not allow enough connectivity in the network. Higher throughput can be achieved by using a global memory multiprocessor architecture or by providing several fast, parallel buses between processing elements. This solution drives the implementation cost up due to the large amounts of external hardware needed to create the interfaces between the DSPs. Finally, mapping an algorithm to the completed network is a complex, manual procedure dependent upon the topology of both the network and the algorithm data flow.

Systolic arrays and array processors attain very high speeds for select problems such as those with nested iterative loops. Some automated programming methodologies also exist for such algorithms. Irregular problems, however, do not map as well to these systems of regularly interconnected programmable elements, thus increasing the programming time and effort. The mapping is also complicated when the size of the algorithm exceeds the fixed size of the processing array. Increasing the size of the array is costly and inefficient if small problems are being handled. In order to reduce the cost of array systems, the basic component must be built separately and interconnected as needed. However, the array processing elements of many architectures (such as the MPP) are too simple to be generally applicable to a wide range of signal processing algorithms. The programmable systolic elements (PSC and WSAP) are a better choice, but their I/O schemes are too restrictive for many problems.

Dataflow systems can execute any program expressed as a dataflow graph at very high speed (theoretically). No manual programmer effort is needed to map the algorithm to the architecture since dataflow computers dynamically extract the parallelism. Unfortunately, theoretical speeds are seldom achieved in practice. Typically, a single system controller bottlenecks the dynamic scheduling and the flow of tokens overloads the memory bandwidth. The complexity of the hardware for these systems also makes them very expensive. Attempts to remove the bottlenecks and reduce the costs of dataflow systems have adversely affected
their automatic concurrency detection (e.g. the μPD7281) and re-introduced the mapping problem.

MIMD systems lose much of their speed because of synchronization and communication delays between the processing elements. Due to the expense of the interprocessor network, the MIMD architectures produced so far have had fixed networks with low communication delays averaged over many applications. These fixed networks are not optimal for all algorithms, however, so some speed is given up for specific programs because it is too much effort and costs too much to alter the topology. The resulting mismatch between the algorithm and the architecture increases the mapping effort.

Dedicated signal processing systems created through the use of silicon compilers achieve high sampling rates, but a new chip must be designed for every algorithm. This is expensive for low-volume systems. There is also a relatively large time delay incurred by the need to fabricate the system. Of course, a programmable system could be designed using a silicon compiler, but this returns us to the question of how to construct a good programmable signal processing system in the first place.

2. A Proposed Solution

Previously built signal processing architectures have not provided adequate solutions to building high-speed, low-cost systems. My thesis is that such systems can be implemented if the following are provided:

- a DSP chip with an architecture that efficiently executes the operations commonly found in signal processing software;
- an interprocessor communication scheme which efficiently handles the overhead involved in data transfers and permits the resources devoted to communications to be incrementally allocated;
- an automatic programming system which examines a specific signal processing algorithm and determines:
+ the topology for a network of the DSP elements,
+ the partitioning of the algorithm for the network,
+ the scheduling of the algorithm components on the DSP elements,
such that the execution time of the algorithm is minimized.

Past attempts at computer design have taken the following directions:

- creation of a uniprocessor matched to a high-level language [Patt82],
- creation of a uniprocessor and associated optimizing compiler [Henn82],
- and creation of a multiprocessor architecture and associated parallel compiler [Fish84].

My thesis takes the unique approach of designing a programmable DSP element which is
easily built into a multiprocessor tailored for any specific digital signal processing algorithm.
Each multiprocessing system will be synthesized by an automated design program working
from the given signal processing algorithm. This is quite different from mapping an algo­
rithm onto an already constructed multiprocessor network.

Figure 3.1 depicts the development process for the articles described above. A signal
processing algorithm analysis is used to determine the uniprocessor qualities of the new DSP
chip architecture. This insures a good match between the processor and the application area.
Additional analysis will guide the choice of an interprocessor communication scheme. The
method of transferring data between processors will affect the architecture of the DSP chip
and the associated automated programming system. Once an initial configuration of the DSP
chip and programming system are available, they will be iteratively modified to remove
shortcomings.

The following chapters will discuss the decisions and choices made in performing the
algorithm analysis, designing the interprocessor communications scheme and DSP chip, and
creating the automated programming system.
Figure 3.1: Development Cycle for the DSP chip and Automated Programming System
1. Motivation

When designing a new computer, it is necessary to examine algorithms from the intended application area to find frequently used operations. Then the architecture of the proposed computer can be arranged to efficiently execute these operations. In this way, the computer architecture and application area are matched with the hope of improving some performance metric (e.g. throughput, memory usage, etc.).

This chapter describes the results of an analysis of a group of signal processing programs. The analysis was performed to determine what characteristics a digital signal processor should possess in order to efficiently execute such programs.

2. Plan Of Analysis

Previous efforts to analyze computer programs fall into two categories [Knut71]:

*Static method:*

The program text is scanned and statistics are kept concerning the relative frequency of the constructs and operations which are found. This method ignores the fact that some operations seldom occur in the program text but are executed many times while the program is running due to their inclusion within a loop or a much-used subroutine.

*Dynamic method:*

The program is executed and a record is kept of how many times each operation is performed. This alleviates the problem encountered with the static analysis in that
the significance of each operation is now based upon the actual number of times the operation is performed rather than how many times it occurs in the program text. A new problem arises, however, because the frequency of use for each operation may be dependent upon the data input to the program under analysis, and the statistics might change significantly from one data set to another. Such programs must be analyzed several times with different data sets and the results merged into a final set of statistics.

The program analysis technique reported here uses a new method which is a combination of the two discussed above. Since most signal processing algorithms use iterative loops which are nested within one another, a quasi-dynamic analysis method can be used. This entails automatically scanning the text of a program and adjusting a counter which records the current depth of loop nesting. Concurrently, as each program operation is encountered in the text, it is weighted by a factor dependent upon the current nesting depth and added into the statistical data. Obviously, those operations found within the innermost loop of a set of nested loops will weigh most heavily since they will be executed more often than those in the outer loops.

Quasi-dynamic analysis has the advantage of static analysis in that programs need only be scanned once. The benefits of dynamic analysis are also received since the importance of program operations is based upon the number of times they are executed. The main drawbacks of quasi-dynamic analysis are that the assignment of loop weighting factors can be rather arbitrary, and the effect of data-dependent flow control statements cannot be easily determined. However, dynamic analysis also suffers from these problems and attempts to reduce their significance by using multiple input data sets for each program being analyzed.
3. The Signal Processing Program Set

Once the method of gathering statistics is selected, a representative set of signal processing programs is needed for analysis. The primary criteria for selecting such a set of programs are:

- The programs must be widely used in order to prevent the skewing of the statistics by some special-purpose algorithm.

- The major portion of each program must perform signal processing related operations. Program operations related to input and output of data are not desired since they are system dependent.

- The programs should be coded as efficiently as possible so that unnecessary and redundant operations do not affect the statistics.

- The programs must be written at as high a level as possible so that the use of signal processing operations is not affected by the underlying architecture of the target machine.

Luckily, a library of widely used signal processing programs written in the C programming language is readily available on our VAX 780 system (Table 4.1). The majority of these programs satisfy the first criterion. An additional advantage to using these programs is the separation of data input and output operations from DSP operations via the use of subroutines. Thus, the second criterion is satisfied merely by excluding the data I/O subroutines from the program set. Unfortunately, all the DSP library programs do not possess the efficiency required to meet the third criterion, as will be seen later. Also, C does not support truly high-level signal processing constructs (APL would be a better choice for that), therefore the programs contain a bit too many low-level operations to truly satisfy the fourth criterion. Despite these flaws, the library was used for the analysis since it was available and the programs display many important characteristics of signal processing algorithms.
### Table 4.1: Signal Processing Programs Selected for Analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acfblk.c</td>
<td>Autocorrelation (block method)</td>
</tr>
<tr>
<td>acfrec.c</td>
<td>Autocorrelation (recursive method)</td>
</tr>
<tr>
<td>addmat.c</td>
<td>Matrix addition</td>
</tr>
<tr>
<td>autcor.c</td>
<td>Autocorrelation matrix computation</td>
</tr>
<tr>
<td>calcmean.c</td>
<td>Mean vector calculation</td>
</tr>
<tr>
<td>cepstrum</td>
<td>2-D spectrum calculation</td>
</tr>
<tr>
<td>conv3.c</td>
<td>2-D space domain convolution</td>
</tr>
<tr>
<td>copymat.c</td>
<td>Matrix copy</td>
</tr>
<tr>
<td>covmat.c</td>
<td>Expected covariance matrix calculation</td>
</tr>
<tr>
<td>durbin.c</td>
<td>Reflection coefficient calculation</td>
</tr>
<tr>
<td>eigenrtn.c</td>
<td>Eigenvalue/eigenvector calculation</td>
</tr>
<tr>
<td>fftdiv.c</td>
<td>Fourier transform division</td>
</tr>
<tr>
<td>fftlc.c</td>
<td>Linear combination of Fourier transforms</td>
</tr>
<tr>
<td>fftmult.c</td>
<td>Fourier transform multiplication</td>
</tr>
<tr>
<td>fft1d.c</td>
<td>1-D Fourier transform</td>
</tr>
<tr>
<td>fft2dm.c</td>
<td>2-D Fourier transform</td>
</tr>
<tr>
<td>hadamard.c</td>
<td>Hadamard transform (floating point)</td>
</tr>
<tr>
<td>ihadamard.c</td>
<td>Hadamard transform (fixed point)</td>
</tr>
<tr>
<td>initmat.c</td>
<td>Matrix initialization</td>
</tr>
<tr>
<td>invdiag.c</td>
<td>Diagonal matrix inversion</td>
</tr>
<tr>
<td>inversel.c</td>
<td>Matrix inversion</td>
</tr>
<tr>
<td>lpcgain.c</td>
<td>LPC gain calculation</td>
</tr>
<tr>
<td>lpcrx.c</td>
<td>LPC receiver simulation</td>
</tr>
<tr>
<td>lpctx.c</td>
<td>LPC transmitter simulation</td>
</tr>
<tr>
<td>multmat.c</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>normvi.c</td>
<td>Matrix infinity norm calculation</td>
</tr>
<tr>
<td>onepulse.c</td>
<td>Voiced speech generator</td>
</tr>
<tr>
<td>predict.c</td>
<td>Recursive filter simulation</td>
</tr>
<tr>
<td>pulser.c</td>
<td>Pulse train generator</td>
</tr>
<tr>
<td>quant.c</td>
<td>Quantizer simulation</td>
</tr>
<tr>
<td>rangaus.c</td>
<td>Gaussian signal generator</td>
</tr>
<tr>
<td>rfft.c</td>
<td>Fourier transform (real sequence)</td>
</tr>
<tr>
<td>scalmult.c</td>
<td>Scalar-matrix multiplication</td>
</tr>
<tr>
<td>spectrum.c</td>
<td>2-D spectrum calculation</td>
</tr>
<tr>
<td>sqrtdiag.c</td>
<td>Diagonal matrix square-root calculation</td>
</tr>
<tr>
<td>thres.c</td>
<td>Image threshold calculation</td>
</tr>
<tr>
<td>transmat.c</td>
<td>Matrix transposition</td>
</tr>
<tr>
<td>transmatq.c</td>
<td>Square matrix transposition</td>
</tr>
<tr>
<td>voisp.c</td>
<td>Voiced speech generator</td>
</tr>
<tr>
<td>vusdet1.c</td>
<td>Speech voiced/unvoiced/silence determination</td>
</tr>
</tbody>
</table>
4. Results of the Program Set Analysis

Statistics in the following areas were gathered on the signal processing library:

- data variable usage,
- arithmetic construct usage,
- flow control usage,
- subroutine usage.

The subcategories of interest within each of these areas are given in Table 4.2.

4.1. Data Variable Usage

4.1.1. Static Analysis: Data Declarations

The relative frequencies of the various types of data variable declarations within the program set are given in Table 4.3.

As can be seen, global variables are infrequently used, accounting for $\sim 1\%$ of the total number of data variable declarations. This is explained by noting that the programs in the set are intended to be used as subroutines, and the use of global data variables could cause unwanted side-effects. When global variables are used, they typically appear as floating-point arrays or as fixed or floating-point scalars. Complex global data structures are not used.

Function arguments comprise $\sim 38\%$ of all the data declarations. The bulk of the arguments are made up of integer scalars and floating-point arrays, although a significant percentage of function arguments are pointers to customized data structures ($\sim 5\%$).

The remaining $\sim 60\%$ of the data declarations are for local variables. The majority of the local variables are fixed and floating-point scalars which are used as temporary variables inside loops. A small percentage of local floating-point arrays is also declared. Complex structures are seldom used.
<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-Categories</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Variable Usage</td>
<td>global, local, and function argument declarations</td>
<td>int a, b, i, j, *c,*d; float e, f, g[10], h[10], *m; struct complex x, y;</td>
</tr>
<tr>
<td></td>
<td>scalar references</td>
<td>a = b + 5;</td>
</tr>
<tr>
<td></td>
<td>array references</td>
<td>g[i] = h[i] + 5.0;</td>
</tr>
<tr>
<td></td>
<td>pointer references</td>
<td>*c = *d + 5;</td>
</tr>
<tr>
<td></td>
<td>reference scope</td>
<td>global, local, function argument</td>
</tr>
<tr>
<td>Arithmetic Construct Usage</td>
<td>bit-logical</td>
<td>~, &amp;, ^,</td>
</tr>
<tr>
<td></td>
<td>logical</td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td></td>
<td>integer assignment</td>
<td>a = b + 5;</td>
</tr>
<tr>
<td></td>
<td>integer arithmetic</td>
<td>+, -, /, %, *, ++, --</td>
</tr>
<tr>
<td></td>
<td>integer relations</td>
<td>=, &lt;=, &gt;=, &lt;, &gt;</td>
</tr>
<tr>
<td></td>
<td>floating assignment</td>
<td>e = f + 5.0;</td>
</tr>
<tr>
<td></td>
<td>floating arithmetic</td>
<td>+, -, /, *</td>
</tr>
<tr>
<td></td>
<td>floating relations</td>
<td>=, &lt;=, &gt;=, &lt;, &gt;</td>
</tr>
<tr>
<td>Flow Control Usage</td>
<td>loop constructs</td>
<td>for, while, do</td>
</tr>
<tr>
<td></td>
<td>loop counter references</td>
<td>for(i=0; i&lt;10; i++)</td>
</tr>
<tr>
<td></td>
<td>loop nesting depth</td>
<td>g[i] = h[i] + 5.0;</td>
</tr>
<tr>
<td></td>
<td>conditionals</td>
<td>if, ?:, switch, case</td>
</tr>
<tr>
<td></td>
<td>flow breaks</td>
<td>break, goto, continue, return</td>
</tr>
<tr>
<td>Subroutine Usage</td>
<td>subroutine calls</td>
<td>e = square_root(f);</td>
</tr>
<tr>
<td></td>
<td>argument list length</td>
<td>m = mat_mult(g, h);</td>
</tr>
<tr>
<td></td>
<td>subroutine nesting depth</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.3: Variable Declaration Percentages for the Program Set

<table>
<thead>
<tr>
<th>Scope</th>
<th>Type</th>
<th>Scalars</th>
<th>Arrays</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Absolute</td>
<td>Pointer</td>
<td>Absolute</td>
<td>Pointer</td>
</tr>
<tr>
<td>Global</td>
<td>int</td>
<td>0.17%</td>
<td>......</td>
<td>0.17%</td>
<td>......</td>
</tr>
<tr>
<td>(1.01%)</td>
<td>float</td>
<td>......</td>
<td>......</td>
<td>0.67%</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>......</td>
<td>......</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>other</td>
<td>......</td>
<td>......</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>Function</td>
<td>int</td>
<td>14.36%</td>
<td>0.17%</td>
<td>0.33%</td>
<td>......</td>
</tr>
<tr>
<td>Arguments</td>
<td>float</td>
<td>2.50%</td>
<td>0.67%</td>
<td>9.35%</td>
<td>......</td>
</tr>
<tr>
<td>(38.72%)</td>
<td>double</td>
<td>0.33%</td>
<td>......</td>
<td>6.01%</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>struct</td>
<td>......</td>
<td>4.67%</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>other</td>
<td>......</td>
<td>0.33%</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>Local</td>
<td>int</td>
<td>35.39%</td>
<td>......</td>
<td>0.33%</td>
<td>......</td>
</tr>
<tr>
<td>(60.28%)</td>
<td>float</td>
<td>13.86%</td>
<td>......</td>
<td>5.18%</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>1.84%</td>
<td>......</td>
<td>0.67%</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>struct</td>
<td>1.34%</td>
<td>1.34%</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>other</td>
<td>......</td>
<td>0.33%</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

The statistics given in Table 4.3 are purely static in nature and show only the frequency with which a particular data type is declared. No information regarding the frequency with which the variable is referenced during program execution is included. However, the following point is clear: the majority of data variables in the program set are simple integer and floating-point scalars and arrays. Some or all of the following factors may account for this:

- These data structures were the most natural for expressing the DSP algorithms.
- The programmer was very concerned with efficiency and knew that present-day computers handle simple data structures better than complex ones.
- The programmer did not understand the advantages gained by the use of more complex data structures.

The use of this analysis to drive the design of a DSP makes it desirable that the initial two items had the major impact upon selection of data structures used in the library, since this implies the presence of an intelligent programmer. If, however, the third item was the deciding factor, then the resulting computer may be optimized only for poor programmers!
We will assume that this is not the case.

4.1.2. Quasi-Dynamic Analysis: Data References

Table 4.4 details the total number of references made to various data variables in the program set. Each category is divided into subcases where loop nesting is not considered (equivalent to doing a static analysis) and only the innermost loops are counted (equivalent to an infinite weighting of the operations within the innermost loops).

The majority of variable references are to scalars rather than to arrays (between 4 and 5 scalar references to every array reference). This seems contradictory when the main purpose of most DSP algorithms is considered, which is to process arrays of data. However, each array access usually requires one or more references to various scalar indices. Then once the array element is fetched, it is combined with others and held in a temporary scalar variable. Finally, a small number of array references are actually recorded as references via pointers. Thus, the abundance of scalar variable references can be directly attributed to the number of array accesses in signal processing programs.

Table 4.4: Variable References for the Program Set

<table>
<thead>
<tr>
<th>Type or Scope</th>
<th>Total</th>
<th>Inner Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>4762</td>
<td>763</td>
</tr>
<tr>
<td>Array</td>
<td>861</td>
<td>182</td>
</tr>
<tr>
<td>Pointers</td>
<td>243</td>
<td>16</td>
</tr>
<tr>
<td>Global</td>
<td>59</td>
<td>8</td>
</tr>
<tr>
<td>Argument</td>
<td>1608</td>
<td>271</td>
</tr>
<tr>
<td>Local</td>
<td>3956</td>
<td>666</td>
</tr>
<tr>
<td>Loop Counter</td>
<td>1526</td>
<td>275</td>
</tr>
<tr>
<td>Current Loop Counter</td>
<td>1305</td>
<td>211</td>
</tr>
<tr>
<td>Previous Loop Counter</td>
<td>200</td>
<td>52</td>
</tr>
<tr>
<td>Remote Loop Counter</td>
<td>21</td>
<td>12</td>
</tr>
</tbody>
</table>
Because most of the signal processing programs use pointers to access global structures, direct references to global data are infrequent. Therefore, references to global variables are negligible compared to function argument and local variable references. Arguments are also referenced less than half as often as local variables. This occurs since local variables are used as loop counters, array indices, and temporary variables (generating many references), while arguments are typically used just to pass set-up values to the subroutine (generating few references).

Approximately a third of the total scalar references are made to loop counters. In turn, references to the counter controlling the innermost loop are about four times more frequent than references to the counters for the surrounding, outer loops.

The following conclusions may be drawn from the quasi-dynamic data variable analysis:

- References to local data are most common during program execution, followed by accesses to function arguments and global data (which is almost negligible).
- Loop control counters are a very tiny sub-sub-category of the scalar variable family, yet they receive a large percentage of the scalar references. Of all the loop control variables, those for the two innermost nested loops are most frequently accessed.

4.2. Arithmetic Constructs

The type and frequency of occurrence of arithmetic constructs for the signal processing library are shown in Table 4.5. Significant subcategories of the arithmetic constructs are logical operators, arithmetic operators, and relational operators.

4.2.1. Logical Operators

Logical operations such as AND, OR, and NOT are used only 11 times in the entire program set. This displays the simplicity of flow control within the program set, and will be discussed in more depth later.
Table 4.5: Arithmetic Operations for the Program Set

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>Inner Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Bit Logical</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>Integer Assignment</td>
<td>936</td>
<td>76</td>
</tr>
<tr>
<td>Integer Arithmetic</td>
<td>996</td>
<td>203</td>
</tr>
<tr>
<td>Integer Addition</td>
<td>323</td>
<td>78</td>
</tr>
<tr>
<td>Integer Subtraction</td>
<td>119</td>
<td>15</td>
</tr>
<tr>
<td>Integer Multiplication</td>
<td>229</td>
<td>74</td>
</tr>
<tr>
<td>Integer Division</td>
<td>67</td>
<td>6</td>
</tr>
<tr>
<td>Increment</td>
<td>229</td>
<td>24</td>
</tr>
<tr>
<td>Decrement</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Floating Assignment</td>
<td>615</td>
<td>113</td>
</tr>
<tr>
<td>Floating Arithmetic</td>
<td>607</td>
<td>173</td>
</tr>
<tr>
<td>Floating Addition</td>
<td>144</td>
<td>52</td>
</tr>
<tr>
<td>Floating Subtraction</td>
<td>115</td>
<td>34</td>
</tr>
<tr>
<td>Floating Multiplication</td>
<td>265</td>
<td>77</td>
</tr>
<tr>
<td>Floating Division</td>
<td>71</td>
<td>4</td>
</tr>
<tr>
<td>Integer Relations</td>
<td>365</td>
<td>41</td>
</tr>
<tr>
<td>Integer &lt;</td>
<td>244</td>
<td>30</td>
</tr>
<tr>
<td>Integer &lt;=</td>
<td>45</td>
<td>7</td>
</tr>
<tr>
<td>Floating Relations</td>
<td>37</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit-by-bit logical operations occur infrequently compared to the number of arithmetic operations. This would seem paradoxical to anyone familiar with the programming of DSP chips, such as the TMS32020, whose programs contain many instances of multiple-bit shifts and other bit-logical operations. However, these DSPs require the use of such operations in order to maintain arithmetic precision and/or work around an imaginary radix point. The algorithms analyzed here are written in a high level language which provides floating-point operations, thus eliminating the need for the majority of the bit-wise logical operations.

The following points can be made:

- The simple flow control of most signal processing programs requires little use of complicated logical operations.
- Floating point operations eliminate the need for most bit-logical operations.
4.2.2. **Arithmetic Operators**

Integer arithmetic is mainly used within the DSP program set for manipulating array indices and loop control variables. Floating-point arithmetic is applied to the actual data values which are the desired output from the programs. Despite its relegation to "housekeeping" chores, integer arithmetic occurs more frequently than real arithmetic.

In both integer and real arithmetic, additions and multiplies occur far more frequently than divisions. This occurs because most algorithms avoid division due to its lack of speed, and the fact that division is not often used for data indexing (even with modulo addressing). Multiplication and addition, however, are used in accumulation which is a common operation in signal processing algorithms. Data indexing is also easily visualized through the use of multiplication and addition of array dimensions and offsets. However, in an efficient program the data indexing multiplications can usually be eliminated via the use of an iterative loop with an updated array pointer. The abundance of integer multiplications (especially within inner loops) signals that some of the programs in the DSP set are not well written. While re-vamping the programs would probably have eliminated these redundant operations, it is believed that the qualitative aspects of the statistics would remain unchanged.

From the data presented, it can be seen that:

- Data values are most often added, subtracted, incremented, and multiplied. Division is less common.
- Operations performed on data indices and loop control variables are as important as those which involve data values.

4.2.3. **Relational Operators**

The relational operators usually occur within statements which alter the flow of control in a program. The statistics indicate that comparisons between integers occur far more frequently than comparisons of floating-point data values. The integers involved in comparisons
are usually data indices and loop counters, while the floating-point numbers typically represent signal values. Thus, program flow is usually independent of the data input to the algorithm. Instead, the execution is directed by the values of loop counters and data indices which vary in a deterministic fashion.

4.3. Flow Control Usage

The frequencies of use of various constructs which control the flow of execution in a program are given in Table 4.6. Three main families of constructs are available for controlling iterative looping, conditional execution, and breaking the program flow.

4.3.1. Loop Constructs

Iterative execution of statements is achieved with the for, while, and do constructs. The for loop occurs more than twenty times as often as the seldom used while and do loop constructs. The majority of the for loops are used to process arrays starting from a low array address and proceeding to the upper index limit. This is apparent by observing that the operator for incrementing occurs far more than the decrement operator (see Table 4.5), while

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>Inner Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Constructs</td>
<td>289</td>
<td>0</td>
</tr>
<tr>
<td>For</td>
<td>275</td>
<td>0</td>
</tr>
<tr>
<td>While</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Do</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Coincident Loop</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>Terminations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditionals</td>
<td>130</td>
<td>4</td>
</tr>
<tr>
<td>If</td>
<td>113</td>
<td>4</td>
</tr>
<tr>
<td>?:</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Flow Breaks</td>
<td>76</td>
<td>4</td>
</tr>
<tr>
<td>Return</td>
<td>61</td>
<td>4</td>
</tr>
<tr>
<td>Break</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
the most frequently used comparison operators are $<$ and $\le$ (which can compare an array index against its upper limit).

Of the 289 loops contained in the program library, 66 occurrences were found where nested loops ended simultaneously.

Figure 4.1 displays a histogram of the maximum depth of loop nesting found in the program set. Nesting to a depth greater than six levels does not occur, and the average loop nesting depth is 2.5.

The analysis of looping constructs allows us to conclude:

- For loops which increment their loop counters are the prevalent mode of iteration.
- The nesting of iterative loops rarely exceeds 3
4.3.2. Conditionals

The conditional constructs include the if, ?, and switch statements. Within the program set, the if is used for the bulk of conditional statement execution, but the majority of these do not reside within the computationally intensive inner loops.

The infrequent use of logical operators (&&, ||, and !) in the DSP program set indicates that the tests performed by the conditional execution statements are very simple (i.e. if \(a < b\) is much more likely than if \((a < b \&\& c >= d || x != y)\)).

Therefore, we may conclude:

• The if statement is the predominant conditional construct.

• Conditional execution is infrequently performed within inner program loops.

• Conditional execution is usually controlled by very simple logical test statements.

4.3.3. Flow Breaks

The return, break, and continue constructs break the flow of control found in subroutines and loops and cause a change of context. Such flow breaks occur infrequently in the signal processing program set - mostly when a subroutine completes its execution and returns to the calling routine.

4.4. Subroutine Usage

The statistics on subroutine usage in the program set are shown in Table 4.7. Of primary concern are the number and nesting of subroutine calls as well as the size of their argument lists.
Table 4.7: Subroutine Usage in the Program Set

<table>
<thead>
<tr>
<th>Type</th>
<th>Total</th>
<th>Inner Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Subroutine Calls</td>
<td>294</td>
<td>9</td>
</tr>
<tr>
<td>DSP Subroutine Calls</td>
<td>87</td>
<td>4</td>
</tr>
<tr>
<td>Total Passed Arguments</td>
<td>967</td>
<td>33</td>
</tr>
<tr>
<td>DSP Passed Arguments</td>
<td>369</td>
<td>24</td>
</tr>
</tbody>
</table>

4.4.1. Subroutine Calls

The DSP program set contains a significant number of subroutine calls, but less than a third of these pass control to a subroutine which performs a signal processing task. The remainder of the subroutine calls are made to scientific routines (e.g. sine, cosine, square root, etc.) or programs concerned with data input and output. However, very few of these subroutine calls are made in the time-critical inner loops.

The maximum subroutine nesting for each program in the library was used to make the histogram in Figure 4.2. No subroutines were nested more than three deep, and the majority of the signal processing programs called subroutines which executed and returned without further nesting.

Figure 4.3 is a histogram of the maximum combined nesting caused by loops and subroutine calls. The majority of the signal processing programs exhibit a maximum total nesting of only three levels or less.

The analysis of subroutine calls for the library programs shows:

- While subroutines are frequently used, they are not often called within the most computationally intensive program segments (i.e. the innermost loops).
- Subroutine nesting and total nesting (including the effect of loops) do not reach excessive levels within signal processing programs.
Average Subroutine Nesting = 1.2
Standard Deviation = 0.9

Figure 4.2: The Relative Frequencies of Subroutine Nesting

Average Subroutine + Loop Nesting = 3.1
Standard Deviation = 2.24

Figure 4.3: The Relative Frequencies of Looping+Subroutine Nesting
4.4.2. Subroutine Arguments

Figure 4.4 displays the relative frequencies of the sizes of subroutine argument lists found in the signal processing program set. The average subroutine receives approximately three arguments, but a significant portion of the subroutine calls require that 5 or more arguments be passed. However, since the histogram is compiled based upon all subroutine calls, it may be biased by calls to input/output subroutines.

By eliminating calls to I/O subroutines, the average argument list for a pure signal processing subroutine is found to contain more than four parameters (see Figure 4.5). Percentage-wise, the need to pass 5 or more subroutine arguments is even greater than in the previous case. A manual examination of the signal processing library revealed that many of the subroutines use extra function arguments to set up internal parameters which make the subroutine more general. Thus, the size of the average argument list could be decreased by using less general algorithms.

![Histogram](image)

**Figure 4.4:** A Histogram of Subroutine Argument List Sizes
The histograms of subroutine argument list sizes and a manual examination of the library reveals the following:

- Large argument lists are infrequently used in signal processing programs.
- Most signal processing subroutines can be limited to accepting four or fewer arguments.

5. Utility of the Analysis

The program analysis presented in this chapter can be used to determine many of the characteristics of a DSP element which performs efficiently in a uniprocessor configuration. DSP support for multiprocessing networks are examined in the next chapter.
CHAPTER 5

Interprocessor Communications

1. Objectives

The preceding algorithm analysis can be used to guide the definition of a DSP architecture which operates efficiently as a uniprocessor. However, the analysis does not provide any guidelines for selecting a good scheme for achieving multiprocessing capabilities with the same DSP. The performance of a multiprocessor system is heavily influenced by the method used for communicating between processing elements and how well it matches the flow of data within the algorithms which are executed. Typically, a multiprocessor architecture is envisioned for a specific algorithm, so the processing element and communications architecture can be matched to the algorithm and data flow, respectively. [Lind85] In our case, however, we desire the creation of a DSP architecture which efficiently executes widely varying types of signal processing algorithms in both uniprocessor and multiprocessor configurations. No specific target algorithm exists from which to extract the dataflow, so we must rely on general guidelines in the development of a suitable communications technique. A suitable set of qualities which must be possessed by the interprocessor communications interface of the DSP is given below:

**Applicability:**

The communications between DSP elements must reflect the underlying communications structure of signal processing applications.

**Low overhead:**

The synchronization and flow of data between parallel processes should be accomplished with a minimal transfer of information.
Distributed control:

The flow of data between processing elements should be locally controlled to avoid overloading a central coordinator. However, the ability to provide some form of centralized synchronization is also desirable when array processing applications are performed.

Flexibility:

The communication scheme must be suitable for constructing multiprocessor systems with significantly different architectures.

Resource allocation:

The interprocessor communications facilities consume chip area, power, and I/O while providing a communication channel with a fixed bandwidth (i.e. a maximum rate of information transfer). Waste of the communication resources can be avoided if the bandwidth is allocatable between several communication links such that heavily used links receive proportionately more bandwidth than do those over which data transfers are less frequent.

Efficient implementation:

The circuitry for the interprocessor communications interface must be capable of being efficiently implemented with present-day VLSI technology.

The remainder of this chapter will present an interprocessor communications method and its implementation which satisfy all of the above criteria.

2. Interprocessor Communications Based Upon a Restricted Dataflow Model

As was previously seen, standard dataflow systems have a bank of processing elements connected on a token bus over which operations and operands circulate. Any processing element may execute any operation which has sufficient operands to fire. The results of the
operation are returned to the token bus where they cause another operation to execute on some other dynamically assigned processor. The flow of tokens can overload the bus or a central controller if one is employed to moderate the flow.

A method of performing interprocessor data transfer based upon a restricted dataflow model eliminates these problems of standard dataflow systems. Restricted data flow statically assigns operations to specific processing elements and connects these elements together in a configuration which reflects the structure of the dataflow graph (Figure 5.1). Each processing element executes one of its assigned operations whenever all of the operands have been received from other processors via the dedicated links. The results of the operation are

\[ ((i1 \times i2) + (i3 + 1)) - (i3 + 1) \]

![Diagram of operation assignment](attachment:image.png)

**Figure 5.1:** Static Operation Assignment in Restricted Dataflow Systems
passed over dedicated output channels to processors waiting to use them in subsequent operations. The presence and absence of data dynamically controls the synchronization of the processing elements, just as in standard dataflow systems. However, in restricted dataflow systems no dynamic assignment of operations to processing elements occurs, thus removing the need for a global token bus or a central controller.

The compliance of the restricted dataflow model with the list of desired qualities for the interprocessor communications method will now be examined.

2.1. Applicability

Dataflow graphs have been viewed as excellent representations of signal processing algorithms. [Wu83, Brow85] The dataflow approach has also been successfully used with matrix computations which are often found in signal processing applications [O'Le85]. Therefore, standard dataflow computers should be capable of being effectively programmed for signal processing chores (examples were seen in Chapter 2). But do restricted dataflow systems have the same qualities?

The analysis performed in the previous chapter offers an affirmative answer to this question. It was noted that the main flow of control within signal processing algorithms is deterministic since the main control construct is the for loop with constant bounds. The deterministic nature of the algorithms gives us the ability to statically assign dataflow computation nodes to specific processors, thus eliminating the need to do fully dynamic scheduling. Therefore, restricted dataflow systems should be applicable to signal processing programs. Further confirmation is given by noting that both the Wavefront Systolic Array Processor and the MUSEC system shown in Chapter 2 employ methods of interprocessor communication very similar to restricted dataflow.
2.2. Low Overhead

The restricted dataflow model also offers the promise of low communications overhead. Since dedicated links exist between the processors which must communicate, there is no need to append *addressing* information to each datum transferred. This removes a large chunk of the overhead found in standard dataflow systems and other MIMD computers.

It might be argued that data addresses are needed even with dedicated links, such as when a processor sends two distinct pieces of data to another processor. The second processor needs to distinguish between the data elements so that they may be used in the appropriate computations. However, the deterministic nature of most signal processing algorithms usually makes it possible to examine the program and statically determine the order of data transmissions. Then, the sending and receiving processors can be programmed for the determined ordering and no addressing information needs to be sent. Of course, the use of conditional statements could remove enough determinism within the algorithm to prevent use of the above method. In such cases, separate dedicated links could be used for each datum or addressing tags could be added to distinguish the data elements.

With the addition of bidirectional communication links terminated by queues, the restricted dataflow model allows a simple, low overhead method of process synchronization. Data to be transmitted is stored into a transmitter queue in the originating processor until it can be sent over the appropriate dedicated link, after which it is loaded into a receiver queue in the sink processor. The transmission process can continue as long as data exists in the transmitter queue and there is sufficient free space in the receiver queue. If the receiver queue fills, the receiver sends a signal back through the bidirectional link informing the transmitter of the situation. Data transmission is then halted until the receiving processor unloads some data and the receiver signals that the queue is no longer full. If the transmitter queue is filled before this occurs, the process creating the data is halted until the link becomes
unblocked. Thus, the queues and the bidirectional links eliminate the synchronization software overhead and the need for separate hardware back-links.

2.3. Distributed Control

The restricted dataflow model with its dedicated links and automatic synchronization of data transfers allows control of algorithm execution to be distributed throughout the multiprocessing system. The elimination of the central controller removes the possibility of a control bottleneck arising within the system as was possible with standard dataflow systems.

Central control is sometimes preferable in systems such as array processors. In array processors, a data value is often broadcast to many processing elements simultaneously from a single source. Broadcast transmissions can be accommodated with the same synchronization method used to control simple point-to-point data transfers. The only additional requirement is that the signal the transmitter receives indicating a full receiver queue must be formed from the logical OR of the analogous signals from each individual receiver. Thus, data will not be broadcast until each attached receiver has space available in its queue.

Instructions and subroutine entrypoint addresses can be sent over a communication link as easily as data values. Such transmissions enable one processor to send control information to a group of processing elements. These processors would then execute the received instructions. In combination with multi-point transmission, this allows a form of centralized control where each processing element operates in loose lockstep with all the others.

2.4. Flexibility

The use of a restricted dataflow model for signal processing systems allows many different architectures to be built with a single component (Figure 5.2). The ability to distribute and/or centralize the control of the system permits the construction of pipelines, arrays, MIMD, and hybrid systems. The system architecture which is selected depends upon the
particular algorithm to be executed.

2.5. Resource Allocation

A glance at Figure 5.2 indicates that each DSP element must possess hardware to support several dataflow links. Assuming each link transmits and receives 16 bit data words in a parallel format, then 4 links requires 64 I/O terminals per processor. This is a relatively large amount of I/O which might be better utilized for other functions (e.g. use the I/O pins to build another bus to external data memory and fetch twice as much data per time period). This is especially true if only a fraction of the available I/O bandwidth is utilized (which occurs if the processors do not talk to each other frequently).

An alternate solution to the problem is to transfer data over the links in a serial format. Now, four links require only four I/O pins and a 16 bit data word can be transferred in 16 time units. Unfortunately, situations will always be found where serial links introduce too

![System Architectures Using Restricted Dataflow Model](image-url)

Figure 5.2: System Architectures Using Restricted Dataflow Model
much delay and degrade the performance of the entire system.

A better solution is to look upon the pins dedicated to the interprocessor communications interface as an allocatable resource. As an example, suppose 16 pins are dedicated to a reconfigurable dataflow transmitter port in a system which uses 16 bit data words (Figure 5.3). A little-used data link between two processing elements might require the bandwidth obtainable from just one I/O terminal (i.e. a communication delay of 16 time units is acceptable for a link with light traffic). Of the 15 remaining pins, 8 could be used to form a much faster link capable of transferring data with a delay of only two cycles. Such a fast link would find use between two processors which need frequent communication to accomplish a shared task in parallel. The remaining 7 I/O terminals can be assigned in a similar manner. Analogous choices can be made when configuring the pins of the receiver port.

Architectures with as many as 32 separate and independent dataflow links can be implemented with two 16 bit wide reconfigurable ports. Such ports give a system designer a range of options for connecting DSP elements into a multiprocessing system and decrease the pro-
bability that I/O resources will be wasted. If low communication delays are needed, a system configured as a pipeline with a single 16 bit input and output link per processing element would be best (provided the signal processing algorithm can be mapped onto such a structure). If an algorithm is best mapped to a mesh-connected architecture, each processing element could be configured with 4 input ports and 4 output ports, each 4 bits wide.

2.6. Efficient Implementation

The reconfigurable dataflow ports described above solve many problems related to process communication and synchronization while allowing the I/O resources to be incrementally allocated. While dataflow links have been used before (the WSAP possesses 4 such ports), our proposed communication scheme offers the improvements of bidirectional synchronization signaling, multi-point data transfer, and reconfigurability which have not previously been seen. These improvements are not obscure or without merit, so the only plausible reason for their failure to appear before this is the lack of a practical design. Our design for the dataflow ports (hereafter referred to as interprocessor communication ports or IPCs) is very efficient to implement given present-day VLSI technology.

As a starting point, consider the case of building 16 slow, 1 bit links. This is easy: simply build separate transmitter and receiver banks each composed of 16 loadable shift registers which are capable of holding 16 bits (the basic data word length). Once a physical link is established between a particular transmitter and receiver register, data can be transferred as follows (Figure 5.4):

- load the transmitter register with a 16 bit data value,
- simultaneously shift the contents of both the transmitter and receiver registers 16 times,
- unload the data from the receiver register.

This data transfer requires 18 time units to complete: one to load the transmitter, 16 to serially
Figure 5.4: Data Transfer Over a 1 Bit Link
transmit the data, and one to unload the receiver.

At the opposite end of the spectrum, the parallel transfer of a 16 bit data word in a single time unit is difficult using the architecture described above. Each bit in the data word must exit from the transmitter on a separate pin and enter the receiver in the same manner in order to meet the timing constraint. This requires the use of all 16 transmitter and 16 receiver pins. Since each pin is accessible only from its associated shift register, all the transmitter and receiver shift registers must be involved in a 16 bit parallel data transfer. This requires that each of the transmitter shift registers be loaded with a value whose least significant bit reflects the next successive bit of the data to be transmitted. Then, each transmitter register shifts once, loading each receiver register with one of the 16 data bits. Finally, all the receiver registers have to cooperate in order to reassemble the data word.

It turns out that the given IPC architecture can handle the parallel data transfer if each transmitting and receiving shift register is modified to include a combinatorial right-shifting network within its load/unload circuitry. With this addition, a fully parallel data transfer takes place as follows (Figure 5.5):

- The data word enters the first transmitter shift register and is stored. Simultaneously, the combinatorial shifting network shifts the data value right by one bit position and passes the result to the next transmitter register.

- The next transmitter register loads the shifted data, shifts it another bit position to the right, and passes it down to the next transmitter register.

- The above loading process is repeated by each transmitter register. By the time the data reaches the final register, 15 right shifts have occurred. The net result of this loading and shifting is that the least significant bit of each transmitter shift register contains a different bit from the original data word. The combinatorial nature of the right shifter network allows the entire loading process to be completed within one cycle.

- The transmission of the data to the receiver is accomplished by allowing the transmitter shift registers to shift once. This places the original data bits into the most significant bits of all the receiver registers in just one time unit.

- The reassembly of the transmitted data begins with the top receiver register passing its received data downward.
Figure 5.5: Data Transfer Over a 16 Bit Link
• The second receiver register uses its combinatorial shifter to shift the data from the upper register by one bit position. The shifted value is then ORed with the contents of the second register and passed downward to the next receiver register.

• After passing through the remaining 14 receiver shift registers, the original data has been recovered from the dispersed bits and can be output. As was the case with the transmitter loading, this entire operation is accomplished in one time unit due to the combinatorial logic employed in the right shifting network.

The extreme cases of data transmission have been covered. Can the middle ground be handled using this IPC architecture? As an example, consider how a 4 bit link would be implemented (Figure 5.6):

• A 4 bit link requires the use of only four transmitter and receiver registers. As usual, the first transmitter register is loaded with the original data value. This value is also shifted right one bit and passed downward to the remaining transmitter registers.

• The next three transmitter registers do not load any new data, but they each activate their combinatorial right shifters to produce 3 more bit shifts in the data value.

• The fourth transmitter register following the initially loaded register is loaded with the data (which has now undergone 4 right shifts). The data is right shifted again and passed downward to the rest of the transmitter registers.

• The above process continues until four transmitter registers have been loaded with input data which has undergone successive four bit shifts. Thus, the lower four bits of each loaded register contain a unique four bit field of the original data value.

• The data transmission is accomplished in four time units as the loaded transmitter registers shift four times to load the uppermost bits of the connected receiver registers with a distinct field of the original data.

• The topmost active receiver register now passes its new value to the next lower register. The next register does not alter the value it contains, but merely gives the input data a right shift and passes it downward.

• The above process continues until the fourth successive receiver register is reached. This register accepts the shifted data and logically ORs it with its own data, thus concatenating the two four bit fields together. The result is shifted right one bit and passed to the next register.

• The above process is repeated until the original data word is reconstructed in the lowest of the four receiver registers. At this point, the remaining combinatorial right shifters are turned off and the original data value passes out of the receiver.
Figure 5.6: Data Transfer Over a 4 Bit Link
The preceding discussion has indicated how to decimate and reconstruct a data word such that it may be transmitted over from 1 to $2^k$ physical wires (where $k=4$ for our IPC architecture). This is only half the story, however. The synchronization of the receiver and transmitter must be provided such that data is sent only when it is capable of being received. Figure 5.7 details a possible link transmission format which meets this requirement. Each bit transmission time is divided into three phases:

**Period 1:**

During the first phase, each receiver register informs the transmitter register to which it is connected (using the bidirectional link) as to whether another data bit can be accepted.

**Period 2:**

The next phase is used by each transmitter register to signal the receiver register as to the availability of data at the transmitter.

**Period 3:**

The final phase is used to send a data bit if one is available for transmission and

![Dataflow Link Transmission Format](image)

*Figure 5.7: Dataflow Link Transmission Format*
storage space exists in the receiver.

The decision tree for the transmission protocol is as given in Figure 5.8. This protocol allows the transmitter and receiver registers to arbitrate the flow of data without the need for intervention. The processors associated with each IPC are free to perform other duties and need only load and unload the IPC at their convenience. Also, the parallel operation of each transmitter and receiver register is possible, since each register is provided with an appropriate state machine and decoding logic to control the separate operation of the shift register and the combinatorial right shifter. This allows, for example, a transmitter shift register to send out data bits while simultaneously using its combinatorial shifter to aid in the loading of other transmitter registers. A similar situation exists for the receiver registers. Thus, a processing element connected to an IPC possesses a set of reconfigurable communication links adaptable to the needs of a particular algorithm, yet little software overhead is required in the processor.

![Dataflow Protocol Decision Tree](image)

*Figure 5.8: Dataflow Protocol Decision Tree*
to control up to 32 links.

The three phase transmission protocol described above uses two phases to transfer status information and one for the exchange of actual data. In order to reduce the transmission overhead, a two phase format is possible (Figure 5.9). The first phase is still used by the receiver to relay its status to the transmitter, but the remaining phase is used to send the transmitter status and data. During the first transmission period, the transmitter uses the second phase to send a start bit signaling the receiver that data will follow. Upon receipt of the start bit, the receiver assumes that data is present during the second phase of each succeeding transmission period, thus eliminating the need for further status information from the transmitter. The two phase transmission format reduces the time required to transfer data over 1,2,4 and 8 bit wide links (it decreases performance for 16 bit wide links by 33%) while

![Figure 5.9: Two Phase Dataflow Link Transmission Format](image-url)
requiring only a small increase in the complexity of the IPC.

The overall structure of the IPC transmitter and receiver are shown in Figure 5.10 and Figure 5.11. They appear to be very similar and they are, as the following discussion will show.

The interface to the IPC transmitter consists of:

- A 16 bit data port for loading data;
- A 5 bit address used to select one of the 31 dataflow transmitter links;
- Clocks which synchronize the operation of the transmitter;
- A status signal which prevents overwriting the transmitter register contents;
- A set of 16 dataflow links for transmitting data.

The registers used to build each IPC transmitter contain:

- A 16 bit loadable shift register which stores data as it is being transmitted;
- A combinatorial shifter which shifts databus values right 1 bit position;
- A counter which records the number of bits remaining to be transmitted;
- A decoder which controls the loading of the associated shift register;
- A transmitter which encodes data for transmission and checks synchronization.

The interface to the IPC receiver is similar to that of the transmitter:

- A 16 bit data bus used to unload data from the receiver;
- A 5 bit address used to select one of the 31 dataflow receiver links;
- Clocks which synchronize the operation of the receiver;
- A status signal which prevents the reading of non-existent data tokens;
- A set of 16 terminals for receiving dataflow interprocessor data.
Figure 5.10: IPC Transmitter Structure
The components from which each receiver register is made are nearly the same as those of the transmitter register, but they interact in a different manner to accomplish reception:

- A 16 bit readable shift register which stores data as it is being received;
• A combinatorial shifter which shifts databus values right 1 bit position;
• A counter which records the number of bits remaining to be received;
• A decoder which controls the unloading of the associated shift register;
• A receiver which decodes received data and maintains synchronization.

The structures composing the IPC receiver and transmitter are very efficiently implemented using modern VLSI device technology. Therefore, the integration of the IPC with the DSP portion of the processing element is feasible.
1. Design Goals

Chapter 4 discussed the characteristics of signal processing algorithms while Chapter 5 detailed a communication technique suitable for the distributed execution of such algorithms. A DSP whose architecture is based upon these results will now be described.

A primary objective in the design of any digital signal processing system is to obtain high throughput. In our case, the DSP architecture must allow the construction of high throughput systems containing one or more DSPs. This is possible if the architecture provides adequate external and internal parallelism.

External parallelism is exhibited when several distinct DSPs cooperate in the execution of a common task. The issues involved in obtaining external parallelism have already been addressed in the previous chapter on the restricted dataflow model and its implementation, the IPC. The IPC permits multiple processors to communicate with a minimum of software overhead, resulting in more processor cycles being used for performing useful computations.

Internal parallelism involves the simultaneous processing of several tasks within the same DSP. The amount of internal parallelism within a DSP is directly affected by:

- the number of independent computational units contained in the DSP,
- the number of internal buses which can be employed to transfer data between the computational units.

It can be seen that internal and external parallelism are similarly limited by the number of computational units or distinct DSPs, and the number of internal buses or amount of external
I/O bandwidth, respectively.

An additional factor moderating the internal parallelism is the control bandwidth. This is the amount of information available during each instruction cycle for controlling the computational units and internal buses. Obviously, large numbers of internal computational units and buses are of little use if the control bandwidth is insufficient to manage all of their simultaneous operations. It is necessary to obtain a balance between the number of computational units, internal bus bandwidth, and control bandwidth in the design of a DSP architecture.

The second goal, that the DSP be implementable using current technology, restricts the options available for reaching the first goal of high throughput. Technology concerns limit the number of I/O pins, chip area, and power. A limited number of I/O pins restricts the communication between DSPs and reduces the maximum feasible external parallelism. As was seen previously, the IPC addresses this difficulty by allowing the I/O pins to be allocated for greatest effect. Chip area and power limit the internal parallelism by preventing the construction of large numbers of computational units and buses. Control bandwidth can be restricted by either I/O (if the control information is brought into the DSP from an external source) or chip area (if the control information is stored internally). The previous analysis of signal processing algorithms can be used to direct some of the tradeoffs between throughput and implementability when determining the internal DSP architecture.

The ability to easily construct multiprocessor systems with small amounts of hardware and software is the third goal driving the DSP architecture. Achieving this objective is assured by the nature of the IPC interface, which can pass data between processors without the aid of external logic. The IPC also removes much of the low-level software burden involved in communicating within a multiprocessor.

A final goal is to reduce the effort needed to create software for the DSP. This can be achieved in part by designing the DSP architecture such that it supports a logical, regular
2. A High-Level View of the DSP Architecture

Figure 6.1 depicts the overall internal architecture of a DSP which satisfies the previously discussed design goals. The main components of the architecture are:

IPC:

The interprocessor communications port handles the transmission and reception of data between processing elements in a multiprocessor system.
Data memory:

The data memory component contains a combination of RAM and ROM for use in holding data values and constants needed in signal processing calculations. This section also controls the access to external memory resources.

Data manipulator:

This component performs the majority of the calculations on data used in a signal processing program. As such, it contains a full hardware multiplier to facilitate these operations.

Address generator:

The address generator is concerned with supplying the addresses of locations in the IPC and data memory which are used by the data manipulator. General purpose computations are also supported.

Instruction storage and control:

This component stores the instructions which implement a particular signal processing algorithm and also manages the sequence in which they are executed.

The above components are interconnected by a set of buses and control lines described below:

Data buses:

Two buses are provided for shipping data between the IPC, memory, data manipulator and address generator. Each bus is broken into four adjacent sections connected via gateways. Closing these gateways allows each component to use a section of a data bus without interfering with the actions of the others. Opening the gateways permits the transfer of data between components.

Address buses:

Two buses are used to send addresses from the address generator to the IPC and/or
data memory.

Program address bus:

An additional bus originates from the address generator and is used to inform the controller of the address of the next instruction.

Status:

Status lines from the datapath provide information to the controller which affects the sequencing of instructions.

Control lines:

The signals which control the datapath are derived from the decoding of the instructions and sent to the datapath over these lines.

For practical reasons mentioned above, the number of data buses was limited to two. The need for sufficient internal parallelism precipitated the addition of the gateways on these data buses. The gateways do not offer a perfect solution, however, since non-adjacent components which share data can prevent an intervening unit from using a particular data bus. For example, the transfer of data from the IPC to the address generator over data bus A prohibits the data manipulator from using the same bus to fetch data stored in the memory unit. Therefore, the linear arrangement of the datapath components can greatly affect the amount of internal parallelism delivered by the dual data buses. The speed at which the data manipulator can perform calculations is a primary determinant of the DSP throughput. Therefore, the other components were arranged such that the data manipulator could easily access and process data. The following arguments show how the component ordering of Figure 6.1 was determined:

Data memory placement:

The memory component holds arrays of data used by the data manipulator during the execution of a signal processing algorithm. Therefore, these two units were
adjacently placed.

**IPC placement:**

The IPC is used to transfer data values between DSPs in a multiprocessor system. Therefore, it is often a source or sink for data handled by the data manipulator and should be placed close by. The IPC is not accessed as frequently as the memory, however, and so it is placed adjacent to the memory and farther from the data manipulator. An added advantage of this arrangement is the possibility of allowing the IPC to perform block transmissions and receptions by independently accessing the data memory during idle cycles.

**Address generator placement:**

The address generator creates addresses for the data memory and IPC ports and can be placed next to these units. However, the address generator also provides support for general purpose computations and can be used to aid the data manipulator. (For example, the data manipulator could handle the mantissas of data values while the address generator could perform computations with the exponents.) This promise of synergism prompted the adjacent placement of the address generator and data manipulator.

In addition to the arrangement of the components, a suitable timing scheme can increase the utility of the data buses. Such a 4-phase timing technique is illustrated in Figure 6.2 and is predicated upon the assumption that transfer times over a precharged bus are significantly shorter than the cycle times of the computational components (particularly the multiplier in the data manipulator). Therefore, a single bus can be used to perform two data transfers during one 4-phase instruction cycle. If each datapath component is constructed so that input and output occur only during φ2 and φ4, then intercomponent communications will also be easily accomplished.
reduce the total number of program steps required to express a given algorithm. Finally, many signal processing algorithms can be implemented with compact programs which can easily fit into a modestly sized internal ROM, therefore the increased area due to the wide instruction is not a major consideration.

No facilities are provided for fetching instructions from an external source. In order to support externally stored programs, the external data bus must be used or a separate program instruction bus (with associated program address bus) must be built. Using the external data bus forces each instruction to be the same width as a data word or requires incremental instruction fetching via multiple memory fetches. A 16-bit data bus would force the use of a heavily encoded instruction or several memory accesses, both of which create more delays. A 32-bit data bus eases these problems, but even so instructions cannot be read into the DSP simultaneously with data for the arithmetic units. The delays caused by this bus contention can be eliminated with an independent program bus, but the I/O resources required for this are substantial. Therefore, it greatly simplifies things if the program is stored internally.
Instructions are read from the ROM using a simple decoder which accepts the program address and several interrupts as inputs. If the interrupts are quiescent (or their masks are enabled), then the program address is used to access the ROM for the next instruction. Otherwise, each interrupt causes an instruction to be fetched from a dedicated ROM location. The instruction can cause a jump to an interrupt service routine for complex operations. Many interrupts, however, require only simple servicing (such as reading a value from an analog-to-digital converter). For these cases, the horizontal instruction format provides enough parallelism to perform the entire interrupt service routine in a single step. This leads to very fast interrupt servicing with low overhead.

Instructions are retrieved from the ROM and stored in an instruction register. They are then applied to the datapath at the start of the instruction cycle (see Figure 6.4). It is possible for an instruction to be aborted because it requires the use of a resource which is presently unavailable. There are two conditions which can cause an instruction abort:

**IPC fault:**

An IPC fault occurs if an attempt is made to:

- read data from an IPC receiver which is currently empty,

---

![Figure 6.4: Instruction Controller Timing](image)

---
• load data into an IPC transmitter which is currently full.

Memory fault:

A memory fault occurs when an attempt is made to:

• use a data value before the associated memory fetch is complete,
• store a data value into memory before a previously initiated store operation is complete.

If either of the above faults occurs, the instruction register is reloaded with its current contents. Thus, an instruction will repeat until the fault is cleared. An abort signal is also supplied to the datapath so that the state of the DSP (e.g. the program counter) does not change until the instruction can successfully complete. Both the IPC and data memory components possess sufficient independence to clear their respective faults without the aid of the main controller, so eventually the instruction flow will continue.

4. The DSP Data Manipulator The main computational chores of a signal processing algorithm are processed by the data manipulator (Figure 6.5) through the combined actions of three major subunits:

• a windowed register set,
• an ALU containing a multiplier, adder, accumulator, various shifters and support for exponents and division,
• and a stack for the storage of accumulator values.

4.1. The Data Manipulator Register Set

The register set is used to store frequently referenced data values and temporary results, thus reducing the accesses to the data memory. Subroutine calls and interrupts, however, can necessitate the storing of these registers into memory to prevent their alteration by a
subroutine or interrupt service routine. This saving and restoring of registers creates overhead which reduces the throughput and slows the interrupt response of a DSP. A significant reduction in this overhead can be achieved by organizing the register set hardware as a series of smaller register windows [Kate83]. With such an arrangement, a subroutine call causes instructions to access registers in a window other than the one used by the calling routine. Returning from the subroutine reactivates the original window whose register values are unchanged. A similar process occurs when an interrupt is serviced. Note that explicit software for saving and restoring the register values has been replaced with a hardware signal which switches the active window. In effect, the register set has been turned into a stack composed of windows.

By overlapping the register windows, the effort of passing subroutine parameters and returning computed results can also be reduced. The registers which are common to both
windows can be used for these dual purposes while the remaining registers in each window can still be used independently.

Figure 6.6 shows details of the register set. A total of 20 registers organized into four windows are provided. Each window contains eight registers with an overlap of four registers between each adjacent window. A set of four register windows was chosen since the program analysis found no signal processing programs containing more than four nested subroutines. If interrupts are active, then the maximum number of nested subroutines might have to be reduced to three (to prevent an overflow of the register set). However, many interrupt service routines in signal processing systems simply handle data transfer with an external device and do not need access to an independent register window. Thus, four windows will usually suffice for handling four nested subroutines while still responding to interrupts. The overlap between adjacent windows consists of four registers which can be used to pass parameters or results between a calling routine and a subroutine. The overlap was
determined by noting from the program analysis that a majority of the subroutines received four or fewer parameters.

Because the data manipulator ALU typically requires two operands upon which to operate during each instruction cycle, the register set is dual-ported to prevent the ALU from standing idle while waiting for data. The NMOS design of the register bit cells (CMOS is also possible) and their interface to the data buses is detailed in Figure 6.7. Note that when a register is accessed, different buses are used for reading and writing data. This allows more flexible data transfer capabilities, such as the extreme case wherein the contents of two registers are simultaneously exchanged.

The timing used by the register bit cells confers several important capabilities. Data can be read from and written to the cell during clock phases \( \phi_2 \) and \( \phi_4 \), as is required by the global bus timing seen previously. The occurrence of the refresh signal during phases \( \phi_1 \) and \( \phi_3 \) uncouples the registers and allows them to be simultaneously read and written. This timing sequence also provides a means of prohibiting state changes in the DSP when instructions are aborted. If a register is written during \( \phi_2 \), this operation can be negated by the disabling of

Figure 6.7: Register Set Construction and Organization
the refresh during \( \phi 3 \) by the abort signal. Alternately, if a register is to be written during \( \phi 4 \), this can be easily prevented by masking the write signals with the abort signal. In either case, state changes can be prevented as long as the instruction abort signal is valid before the start of phase \( \phi 3 \) and lasts through the duration of \( \phi 4 \). The abort signal is ignored during phases \( \phi 1 \) and \( \phi 2 \), thus insuring that each register cell is refreshed at least once during an instruction cycle.

4.2. The Data Manipulator ALU

The data manipulator ALU circuitry is shown in Figure 6.8. The ALU contains the following units:

- a multiplier/accumulator,
- an exponent controller,
- and a quotient controller.

The multiplier/accumulator is built from the following set of serially connected components:

- two input latches which receive the operands from the data buses during phase \( \phi 2 \);
- a multiplier which either computes the 32-bit product of the 16-bit operands, or merely concatenates the operands into a 32-bit number;
- a right shifter which can shift the output of the multiplier by up to 31 bit positions;
- an adder which can add or subtract the 32-bit right shifter output from the 32-bit accumulator value;
- another right shifter which can shift the adder output either zero or one bit position;
- a 32-bit accumulator which stores the output from the one-bit right shifter.

The combination of the multiplier and adder makes it easy to perform sums of products, but with an accumulator which is only 32-bits wide, the possibility of overflow is must be
Figure 6.8: Closeup of the DSP Data Manipulator ALU
considered. Instead of adding more bits onto the accumulator to prevent overflow, a method of scaling the sum was chosen. Upon overflow (as indicated by a signal from the adder), the output of the adder is right-shifted by one bit position and the sign is restored (effectively dividing the next accumulator value by two). Simultaneously, the exponent controller increments the S register. The S register controls the amount by which the multiplier product is right shifted. Incrementing S scales the multiplier products by a factor of two, also. As the summing continues, any further overflows will increase the scaling. This insures that no accumulator overflows occur at the cost of a loss in precision in the final answer. The amount by which the final answer has been scaled can be recalled from the S register. It is also possible to perform static scaling or multi-cycle floating point operations by loading the S register with the appropriate value.

Hardware support is also provided for division, although this operation is not nearly as pervasive as multiplication in signal processing algorithms. The quotient controller works in conjunction with the multiplier/accumulator and exponent controller to perform a non-restoring division in 16 instruction cycles [Wase82]. The quotient is stored in register Q while the remainder is found in the accumulator.

The total set of ALU operations is given in Table 6.1. The set is sparse: no facilities are provided for logical, bit-logical, or comparison operations. However, the program analysis showed that these operations were infrequently performed on data values and did not need support in the data manipulation component.

The timing of the data ALU operations is given in Figure 6.9. The ALU begins a new operation at the start of phase φ3 (after the operands have arrived) and delivers a new value to the accumulator by the end of phase φ2 of the next instruction cycle. Therefore, the programmer must insert a single instruction delay when storing the accumulator value into the register set or memory since this can only occur during phase φ2.
Table 6.1: Possible Data Manipulation Component Operations

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no operation</td>
</tr>
<tr>
<td>1</td>
<td>ACC → ACC + B:A</td>
</tr>
<tr>
<td>2</td>
<td>ACC → ACC \times B</td>
</tr>
<tr>
<td>3</td>
<td>ACC → A × B</td>
</tr>
<tr>
<td>4</td>
<td>ACC → ACC + A × B</td>
</tr>
<tr>
<td>5</td>
<td>ACC → ACC - A × B</td>
</tr>
<tr>
<td>6</td>
<td>Q → 0, S → 16 (division initialization)</td>
</tr>
<tr>
<td>7</td>
<td>ACC → ACC ÷ B (division step)</td>
</tr>
<tr>
<td>8</td>
<td>Load S</td>
</tr>
<tr>
<td>9</td>
<td>Store S</td>
</tr>
<tr>
<td>10</td>
<td>Store Q</td>
</tr>
</tbody>
</table>

4.3. The Data Manipulator Accumulator Stack

The accumulator stack is composed of two independent stacks, each of which stores half of a double precision accumulator value (Figure 6.10). Four entries are provided in each stack, thus permitting subroutine nesting to a depth of five (including the original accumulator in the ALU). Upon subroutine entry, the value in the data ALU accumulator is pushed
onto the stack. The top of each accumulator stack is mapped into the register set (see Table 6.2), so the subroutine can access the old accumulator value as a parameter while it performs calculations. Upon returning from the subroutine, the accumulator stack is not automatically popped, thus permitting the subroutine to return a double-precision result in the accumulator.

5. The DSP Address Generator

The separate addressing component of the datapath reflects the importance of address generation in signal processing programs. The significance of addressing was seen during the program analysis where it comprised approximately half of the total computational load. The address generator (Figure 6.11) is made up of the following components:
Table 6.2: Data Register Mapping

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>data register 0</td>
</tr>
<tr>
<td>1</td>
<td>data register 1</td>
</tr>
<tr>
<td>2</td>
<td>data register 2</td>
</tr>
<tr>
<td>3</td>
<td>data register 3</td>
</tr>
<tr>
<td>4</td>
<td>data register 4</td>
</tr>
<tr>
<td>5</td>
<td>data register 5</td>
</tr>
<tr>
<td>6</td>
<td>data register 6</td>
</tr>
<tr>
<td>7</td>
<td>data register 7</td>
</tr>
<tr>
<td>8</td>
<td>lower accumulator stack</td>
</tr>
<tr>
<td>9</td>
<td>upper accumulator stack</td>
</tr>
<tr>
<td>10</td>
<td>lower accumulator</td>
</tr>
<tr>
<td>11</td>
<td>upper accumulator</td>
</tr>
<tr>
<td>12</td>
<td>not used</td>
</tr>
<tr>
<td>13</td>
<td>not used</td>
</tr>
<tr>
<td>14</td>
<td>open left bus gateway</td>
</tr>
<tr>
<td>15</td>
<td>open right bus gateway</td>
</tr>
</tbody>
</table>

- a windowed register set identical to the data manipulator register set with respect to timing and construction,
- two ALUs for generating new addresses and performing general purpose computations,
- two temporary accumulators for storing the results of the ALU computations,
- a stack containing the loop counters, and
- a program counter stack.

5.1. The Address Generator ALUs

The dual ALUs provide addresses such that two fetches from data memory may occur in parallel, thus allowing the data manipulator to receive an uninterrupted flow of operands. The basic ALU operation has the following format:

\[ \text{reg1} \leftarrow \text{reg1 op reg2} \]

The timing for such an operation is shown in Figure 6.12. The operands are received by the
Figure 6.11: The DSP Address Generator
ALUs from the data buses during phase $\phi_2$. During the same phase, the IPC and data memory modules receive the operand values over their associated address buses. (Obviously, the address generator supports post-modified addressing only.) During the succeeding three phases, the ALUs perform computations and load the results into the temporary accumulators.

The temporary accumulators serve two functions:

- register aliasing,
- register updating.

The address ALUs receive new operands during $\phi_2$ of every instruction cycle, but the updated values for the address registers affected by the previous instruction are still stored in the temporary accumulators at this time. If the same registers are accessed on consecutive instructions, erroneous results will be obtained. For this reason, each temporary accumulator is capable of comparing the register address of its contents with the register addresses
currently being accessed. If a match occurs, the temporary accumulator disables the register access and places the value it contains onto the appropriate bus. This aliasing of a temporary accumulator as a register insures that the correct operand value will be used [Kate83].

After aliasing is completed, each temporary accumulator unloads its value into the appropriate register during phase $\phi_4$ (now that the data buses are free). Note that the source of the operands for the addressing ALUs does not necessarily lie in the address generator register set, since the data buses could be driven by any of the other three major datapath components. For cases like this, the temporary accumulators do not attempt to write the result back to the original source of the operand due to the complexity of such an operation. Instead, each accumulator updates a default address register in the current window (ALU1 uses address register 6 while ALU2 uses register 7).

A more detailed view of an address generator ALU (Figure 6.13) shows that it contains:

- a general purpose ALU,
- a register for storing an offset which can be used to modify an address,
- a multiplexer which selects either the offset register or second data bus for use as an operand,
- a modulus register for storing the length of a FIFO,
- modulo logic for correcting the updated address such that it "wraps around" at the FIFO memory boundaries,
- a small ROM for storing 16 arbitrary constants.

The general purpose ALU supports operations useful for generating addresses as well as the standard functions (Table 6.3). The shifting, bit-logical, and comparison instructions which were excluded from the data manipulator ALU are housed in the address generator ALUs. These operations can still be applied to values used by the data manipulation section by opening the appropriate data bus gateways, however this prevents the generation of one
Figure 6.13: Closeup of a Single DSP Address Generator ALU
Table 6.3: Address Generator ALU Operation Codes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no operation</td>
</tr>
<tr>
<td>1</td>
<td>add</td>
</tr>
<tr>
<td>2</td>
<td>add with carry</td>
</tr>
<tr>
<td>3</td>
<td>subtract</td>
</tr>
<tr>
<td>4</td>
<td>subtract with carry</td>
</tr>
<tr>
<td>5</td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td>6</td>
<td>shift right</td>
</tr>
<tr>
<td>7</td>
<td>read ROM constant</td>
</tr>
<tr>
<td>8</td>
<td>rotate right</td>
</tr>
<tr>
<td>9</td>
<td>rotate left</td>
</tr>
<tr>
<td>10</td>
<td>increment</td>
</tr>
<tr>
<td>11</td>
<td>decrement</td>
</tr>
<tr>
<td>12</td>
<td>bit-logical AND</td>
</tr>
<tr>
<td>13</td>
<td>bit-logical OR</td>
</tr>
<tr>
<td>14</td>
<td>bit-logical XOR</td>
</tr>
<tr>
<td>15</td>
<td>bit-logical NOT</td>
</tr>
<tr>
<td>16</td>
<td>modulo addition</td>
</tr>
<tr>
<td>17</td>
<td>modulo subtraction</td>
</tr>
<tr>
<td>18</td>
<td>modulo increment</td>
</tr>
<tr>
<td>19</td>
<td>modulo decrement</td>
</tr>
<tr>
<td>20</td>
<td>modulo addition of offset</td>
</tr>
<tr>
<td>21</td>
<td>modulo subtraction of offset</td>
</tr>
<tr>
<td>22</td>
<td>addition of offset</td>
</tr>
<tr>
<td>23</td>
<td>subtraction of offset</td>
</tr>
<tr>
<td>24</td>
<td>load offset</td>
</tr>
<tr>
<td>25</td>
<td>load modulus</td>
</tr>
<tr>
<td>26</td>
<td>load flags</td>
</tr>
<tr>
<td>27</td>
<td>store offset</td>
</tr>
<tr>
<td>28</td>
<td>store modulus</td>
</tr>
<tr>
<td>29</td>
<td>store flags</td>
</tr>
<tr>
<td>30</td>
<td>comparison</td>
</tr>
<tr>
<td>31</td>
<td>conditional register transfer</td>
</tr>
</tbody>
</table>

or more of the data memory addresses. This loss of parallelism is acceptable since such operations are seldom carried out upon data values (as shown by the program analysis).
Each address generator ALU usually requires two operands per instruction cycle (for a total of four operands per cycle). Since only two data buses exist to carry the data, an offset register is included in each ALU. This permits two operands to be fetched and modified by arbitrary amounts stored in the offset registers during a single instruction cycle. Because many signal processing algorithms repeatedly modify addresses by some constant, the offset register does not need to be updated very often. Of course, it is still possible to update an address with some value other than the one stored in the offset register, but this requires the use of both data buses and typically leaves one address generator ALU idle.

The modulo logic is quite similar to that found in the DSP56000 and ADSP-2100. The address to be modified and the value in the modulus register can be used to determine the starting and ending addresses of a memory region. If the modified address falls outside these boundaries, then the modulo logic corrects the result. The modulo logic can support FIFOs of size 1 to 65,535 by loading the modulus register with the desired length. Loading the modulus with zero reverses the direction of carry propagation in the ALU so as to support a bit-reversed addressing mode suitable for addressing data during calculations of the Fast Fourier Transform.

A small ROM, whose contents are specified by the programmer, is incorporated in each ALU to store often used constants or pointers to larger tables of data. This eliminates the need to store immediate data in the instruction store itself, which would require an increase in the instruction word size or an additional ROM access.

5.2. The Address Generator Loop Stack

A stack of hardware loop counters (Figure 6.14) is built into the DSP due to the pervasive use of the for loop in signal processing programs. As a loop is entered by the program, the number of iterations is pushed onto the stack, forcing all the previous counter values down. The counter on the top of the stack possesses dedicated logic which
The innermost loop is terminated when the top counter reaches zero, causing the previous loop counters to pop up and resume their previous positions. The counter decrement and trigger on zero method was chosen although our program analysis indicated that the majority of loops counted upward and terminated when the index exceeded an upper bound. Being faithful to the analysis would have required extra hardware to store the upper bounds yet would not have yielded better performance since there is no effective difference between incrementing and decrementing a counter.

The loop counter stack approach efficiently supports nested loops. Only four counters are provided on the stack since the program analysis indicates that loops rarely nest deeper than that. If loop nesting does exceed four, the outer loops must be handled by explicit software, but since they are executed relatively infrequently compared to inner loops the overhead does not cause much performance degradation.
The two counters on the top of the stack are mapped into the address generator register space (Table 6.4). Only these two counters were made randomly accessible since the program analysis showed that references to the remaining loop counters seldom occurred. The ability to read the counter values allows them to be used to modify addresses during iterative execution (such as when addressing arrays of data). The ability to overwrite the contents of a loop counter permits premature loop termination by writing in a zero, or arbitrary counter updates through the use of an address ALU instead of the built-in incrementer.

5.3. The Address Generator Program Counter Stack

The program counter stack closely resembles the loop counter stack (Figure 6.15). The address of the current instruction is always kept on the top of the stack and is incremented as instructions are executed. When a subroutine is called, the current program counter is pushed down and the subroutine entrypoint is placed on the top of the stack. Upon termina-

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>address register 0</td>
</tr>
<tr>
<td>1</td>
<td>address register 1</td>
</tr>
<tr>
<td>2</td>
<td>address register 2</td>
</tr>
<tr>
<td>3</td>
<td>address register 3</td>
</tr>
<tr>
<td>4</td>
<td>address register 4</td>
</tr>
<tr>
<td>5</td>
<td>address register 5</td>
</tr>
<tr>
<td>6</td>
<td>address register 6</td>
</tr>
<tr>
<td>7</td>
<td>address register 7</td>
</tr>
<tr>
<td>8</td>
<td>innermost loop counter</td>
</tr>
<tr>
<td>9</td>
<td>innermost-1 loop counter</td>
</tr>
<tr>
<td>10</td>
<td>program counter</td>
</tr>
<tr>
<td>11</td>
<td>previous program counter</td>
</tr>
<tr>
<td>12</td>
<td>not used</td>
</tr>
<tr>
<td>13</td>
<td>not used</td>
</tr>
<tr>
<td>14</td>
<td>not used</td>
</tr>
<tr>
<td>15</td>
<td>open left bus gateway</td>
</tr>
</tbody>
</table>
tion of a subroutine, the old program counter pops up from below and program execution resumes at the instruction following the call to the subroutine. This is a standard stack mechanism for supporting nested subroutine calls.

The stack also supplies a method for performing loop iterations with low software overhead. Upon entry to a loop, the number of loop iterations is pushed onto the loop counter stack. The program counter contents are also pushed down into the PC stack, thus saving the loop entrypoint address. The program counter contents are not replaced, however, and execution continues with the first iteration of the loop. When the end of the loop is detected, the loop counter is decremented and if the result is not zero, the previously saved loop entrypoint is copied back into the current program counter. This returns the flow of control to the top of the loop.
Eventually, the loop counter is decremented to zero and the loop terminates. A special stack operation pops the PC stack but does not overwrite the program counter. This removes the old loop entrypoint address and permits program execution to continue with the instruction following the end of the loop.

Each DSP instruction includes a special tag indicating whether or not it occurs at the end of a loop. This tag bit controls the stack operations previously discussed. This method of loop termination was chosen over that used by the DSP56000 and ADSP-2100 (which store the ending address of the loop and continually compare it to the program counter contents) because it allows a loop to be terminated in multiple locations. Using the comparison technique, conditional branches in a loop can require the insertion of extra program jumps in order to hit the one address which signals the end-of-loop condition. These jumps create delays during the loop iterations, while the tag method avoids this penalty.

The program counter stack contains room for 6 addresses in addition to the program counter. This number was arrived at from the program analysis which showed that loop and subroutine nesting reached a combined depth of six or less the majority of the time. The PC stack, like the loop counter stack, has no overflow protection if nesting becomes too deep. This is not a problem since the nesting of a signal processing program can usually be statically determined (as we did in our program analysis) and explicit software can be added to manage overflows. The ability to address the second PC stack entry and the special stack operations aid in managing the stack during overflows.

The incrementer and looping hardware built into the program counter stack provide hardware support for the most common types of program flow control. In addition, the mapping of the program counter into the register set permits the programmer to synthesize many types of program flow control. For example, conditional branches can be handled through the use of the conditional register transfer operation of the address ALUs with the program
counter register as the destination of the result.

6. The DSP Data Memory

The data memory component of the datapath (Figure 6.16) contains:

- two independent banks of RAM/ROM,
- dual memory address registers with control logic, and
- dual memory data registers with control logic.

6.1. The RAM/ROM Banks

Each RAM/ROM bank is a single-ported memory containing:

Figure 6.16: The DSP Data Memory
- 256 single-precision RAM locations for data storage, and
- 256 single-precision ROM locations for coefficient storage.

Each bank accepts an address during phase $\phi_2$ and outputs data during phase $\phi_4$.

6.2. The Memory Address Register

The memory address register latches the contents of both internal address buses during phase $\phi_2$. Either address which accesses internal memory is sent to the appropriate internal RAM/ROM block (see the DSP memory map in Table 6.5). Otherwise, an external memory access is being performed and the new address is latched onto the external address bus during phase $\phi_3$. If both addresses are external, then the memory address control logic initially outputs the address carried on address bus A. Upon completion of this memory reference, the address latched from address bus B is output.

Table 6.5: The DSP Memory Map

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000-0x000F</td>
<td>1 bit wide IPC ports</td>
</tr>
<tr>
<td>0x0010-0x0017</td>
<td>2 bit wide IPC ports</td>
</tr>
<tr>
<td>0x0018-0x001b</td>
<td>4 bit wide IPC ports</td>
</tr>
<tr>
<td>0x001c-0x001d</td>
<td>8 bit wide IPC ports</td>
</tr>
<tr>
<td>0x001e</td>
<td>16 bit wide IPC port</td>
</tr>
<tr>
<td>0x001f</td>
<td>4 bit wide IPC global receiver port</td>
</tr>
<tr>
<td>0x0020-0x00ff</td>
<td>Unused</td>
</tr>
<tr>
<td>0x0100-0x011f</td>
<td>ROM A / ROM B</td>
</tr>
<tr>
<td>0x0200-0x02ff</td>
<td>RAM A / RAM B</td>
</tr>
<tr>
<td>0x0300-0x03ff</td>
<td>Unused</td>
</tr>
<tr>
<td>0x0400-0xffff</td>
<td>External Memory</td>
</tr>
</tbody>
</table>
6.3. The Memory Data Register

All memory transfers occur between the internal/external memory and the registers in the data manipulator. The memory data register and logic control this flow of data and halts instructions in the event of a memory fault. The possible memory faults are:

- attempting to write data into memory before a previous write operation is complete, or
- attempting to use data which has not yet been retrieved from memory.

When data is moved from the registers to memory, the memory data register checks to see if it currently contains data waiting to enter memory. If so, a fault signal is generated during phase $\phi_2$. Otherwise, the data is transferred to the memory data latch during phase $\phi_4$ (so as not to interfere with the use of the data bus for transferring operands to the data manipulator ALU). The data is then loaded into internal or external memory. If internal memory is accessed, then the transfer is complete by the end of $\phi_4$. Otherwise, the data is latched onto the external data bus during phase $\phi_4$ and remains there until the external memory signals that the data has been received.

When data is being read from memory into a data register, the memory data register behaves like the temporary accumulators of the address generator ALUs. When a data memory location is read, the memory data register stores the address of the destination register to which the data will be sent. In the case of an internal memory access, the data can be loaded into the destination register during phase $\phi_4$. External reads, however, may incur an arbitrarily long delay. Therefore, the memory data register monitors the use of data registers in the data manipulator to observe if the destination register is used as an operand. If so, then the memory data register logic signals the occurrence of a memory fault which causes the instruction to abort. The instruction continues to abort until the needed data is received from external memory. The memory data register then places it onto the appropriate data bus
for use as an operand during phase φ2. Finally, the data is placed into the register set during
the following phase φ4. Note that the instruction abort does not occur when the memory
read is initiated, but rather when the results are used. This permits the programmer to pre-
fetch external memory operands, effectively eliminating the external access delays.

7. The DSP Interprocessor Communications Port

The IPC was described in the previous chapter. The interface between the IPC and the
remainder of the datapath is identical to that of the data memory with respect to functionality
and timing (Figure 6.17). Data is accepted by the IPC during phase φ2 or output to one of the
data manipulator registers during φ4. An IPC fault will occur under the following conditions:

• an attempt is made to load an IPC transmitter which is not empty,
• or an attempt is made to read an IPC receiver which is not full.

The IPC transmitter and receiver interface to the external environment through 16 bit wide ports. An additional four bit wide port is included in the IPC receiver such that global instructions and data can be received from a central controller without reducing the I/O available for communications between DSPs in an array.

8. Performance Results for the DSP Architecture

Due to its dual address generators, single-cycle hardware multiplier, and dual internal RAMs, the DSP described here has a performance equivalent to those described elsewhere when executing simple FIR, IIR, and adaptive filters and FFTs [Klok86, Eich86, Roes86, Fran86]. It is believed that our DSP will offer an advantage when algorithms with more complex flow control are executed, although no actual applications have been coded as yet.

Of more interest is the operation of the DSP in a multiprocessing configuration. In order to observe this operation, the DSP architecture has been coded for simulation in the Axe hardware description language (see Appendix A). Axe allows very detailed simulation and collection of statistics for the DSP in both uniprocessor and multiprocessor systems.

The BRACELET architecture (refer to Chapter 2) was then simulated using the DSP architecture instead of the TMS320. Table 6.6 lists the maximum sampling frequencies for an LMS adaptive filter [McCo80] running on a BRACELET architecture composed of four DSPs with an additional DSP acting as the global controller. The features of the DSP design were altered in the following ways to note the effect upon the system execution rate:

Address generation capabilities:

One of the dual address generators was disabled to reduce the ability of the DSP to access memory. In addition, the modulo arithmetic feature of the address generators
Table 6.6: Performance of a 20-Tap LMS Filter upon a 4 DSP BRACELET

<table>
<thead>
<tr>
<th>Addressing</th>
<th>Memory</th>
<th>IPC</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>No</td>
<td>External</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No</td>
<td>External</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No</td>
<td>External</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>No</td>
<td>Internal</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Yes</td>
<td>Internal</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Yes</td>
<td>Internal</td>
</tr>
</tbody>
</table>

was turned off to determine the effect of pointer arithmetic overhead on the execution speed.

Memory organization:

The effect of memory access delays upon the BRACELET architecture was observed by:

- disabling the internal RAMs and forcing all memory operations to access external RAM with a single cycle delay;
- turning off the pipeline ability of the memory data section so that access delays become visible.

IPC facilities:

The LMS algorithm requires the global broadcast of the error term from the central controller to the four processors forming the ring structure. Removing the global channel from the IPC of each DSP forces all data to be transferred over the local links with a corresponding increase in delay.

The base DSP architecture used for the BRACELET simulation has one address generator with no modulo arithmetic capabilities, no internal memory or memory pipelining, and no global IPC link. A complete iteration of a 20-tap LMS adaptive filter executes in 103 instruction cycles on such a system (giving a maximum sampling frequency of 97 KHz if each
instruction requires 100 nanoseconds). By way of contrast, a similar system built from
TMS32010 signal processors is capable of achieving execution times of only 70 cycles [Barn].

The execution time decreases to 90 cycles when the memory pipelining is enabled. The
ability of the memory data section to handle memory accesses independently of the main DSP
control section has obvious advantages.

Enabling the global IPC receiver removes the delays of passing the error term over the
local IPC links between processors. This lowers the execution time to 81 cycles.

Directing all memory references to internal memory removes the access delays and
decreases the execution time to 66 cycles. Finally, we have matched the TMS32010!

Enabling both address generators permits dual accesses of internal memory to occur
simultaneously, lowering the execution time to 51 cycles.

Finally, addition of the modulo arithmetic capabilities to the addressing generators
removes the pointer update overhead from the DSP program. Now, the LMS algorithm exe-
cutes upon a 4 DSP BRACELET system in just 41 cycles (243 KHz).

The preceding simulation can also be used in an indirect way to judge the cost efficiency
of the DSP architecture. The detailed description for an individual DSP requires ~3000 lines
of Axe code. An equivalently sized program would be needed for describing the TMS320 at a
similar level of detail, since the two designs have roughly the same level of complexity. The
Axe description of a BRACELET system built from DSPs requires just 5 lines since no external
components are needed to provide interprocessor communications or synchronization. The
actual hardware for a working BRACELET, however, requires a full 10''x12'' circuit board
containing 105 chips. Therefore, the Axe description of the TMS320 BRACELET would
require at least 100 lines of code. By this measure, a BRACELET constructed using the DSP
will be roughly 20 times less complex than one built using the TMS320. This comparison
does not even count the interconnection cost, which can be estimated by the total number of
I/O pins possessed by all the chips in each system:

BRACELET using the TMS32010: 105 chips @ 16 pins/chip = 1680 pins
BRACELET using the DSP: 5 chips @ 88 pins/chip = 440 pins

This measure favors the DSP by a factor of four.

A comparison of just one algorithm executed upon just one other system is not conclusive evidence of the superiority of the DSP design. Much work remains to be done in benchmarking algorithms executed on other parallel system configurations against those built using the DSP.
1. The Need for a Programming System

At this point, a DSP architecture has been presented which is matched to the operations most often used in signal processing algorithms. Coupled to this architecture is a flexible interprocessor communications port which synchronizes data transfers between the DSPs in a multiprocessor system. Does this hardware alone solve the problem of creating high-speed signal processing systems through the use of parallelism?

Unfortunately, the answer is no. Each DSP, while optimized for performing signal processing chores and supporting flexible interprocessor communications, is usually but a component of a larger system which executes one or more complete algorithms. During the construction of a multi-DSP system, the designer must answer the following questions (among others):

- How many DSPs are needed to meet the system performance specifications?
- What network topology should be used to interconnect the DSPs?
- What sub-tasks of the total algorithm are assigned to each DSP?
- In what order are the sub-tasks executed by each DSP?

The answers to the above questions are interrelated and non-trivial, creating a large system design space which is difficult to explore. Typically, the system design decisions have been made based upon a combination of prior experience and guesswork. Unfortunately, experience can bias a design away from an original and advantageous configuration, while a single poor guess can distort an entire system due to interactions between decisions.
An obvious solution to the system design problem is to create an automated programming tool which can explore the design space and find some near-optimal solution based upon the system specifications. Such a programming tool would relieve the effort involved in exploring the design space and encourage the investigation of tradeoffs between system objectives. The tool would take as input a description of the signal processing algorithm and the system requirements on throughput, cost, etc. and generate as output a network interconnection topology and detailed programs for each individual DSP. Changes in the system configuration as a result of alterations in the algorithm or system objectives could be easily investigated. For example, given a specific signal processing algorithm, the programming tool could generate systems based on the following criteria:

- Maximizing throughput (effectively attaining the highest sampling rate);
- Minimizing system component count (use as few DSPs as possible);
- Minimizing data transfer delays (this helps throughput);
- Minimizing the width of interprocessor data links (a slow, narrow link costs less);
- Minimizing quantization error (transfers of single-precision data are fast but increase error);
- Minimizing memory needs (eliminates DSPs’ need for slow, expensive external RAM);
- Maximizing program regularity (allows DSPs’ to run efficient inner loops);
- Maximizing program determinism (avoid data-dependent data transfers);

While all of the above factors are important, the main emphasis in the remainder of this chapter will be on designing systems with maximum throughput, since this is a primary concern in signal processing algorithms.

In addition to facilitating system design, the automated programming tool can also have beneficial effects upon the architecture of the DSP. For instance, the programming system might identify a certain mode of interprocessor data transfer which would best be supported
in the DSP hardware for maximum benefit. The DSP design could be altered to include this feature and match it more closely to the programming tool as well as the application area. This technique complements the more conventional modern trend of other computer designers who transfer complexity from the hardware to the support software (for example, by building optimizing compilers for stripped-down architectures [Henn83]).

2. The Architecture of an Automated Programming Tool - PaLS

Figure 7.1 shows the initial architecture of the PaLS automated programming tool, which performs functions such as those described above. A signal processing algorithm processed by PaLS passes through 5 main phases:

Algorithm Input:

The programmer's algorithm is transformed into a convenient representation for further processing.

Partitioning:

The transformed algorithm is divided into a set of sub-tasks which are each assigned to be executed on a single DSP.

Link Assignment:

A network of links is constructed taking into account for the IPC hardware capabilities and the amount of communication needed between the DSPs as a result of the previous partitioning.

Scheduling:

The algorithm sub-tasks for each DSP are scheduled for execution such that the total time to complete the algorithm is minimized.

Code Generation:

The sub-tasks for each DSP are transformed into machine code taking into account for
Figure 7.1: PaLS Architecture
the previously determined schedule.

Each of these phases will now be examined in more detail.

2.1. Algorithm Input

Several formats are available for expressing signal processing algorithms:

- sequential languages;
- signal flow graphs;
- dataflow languages.

Sequential languages such as FORTRAN and C have been used for years to specify algorithms. Therefore, they enjoy the advantages of being very familiar to programmers who can also access large libraries of previously written and debugged subroutines. Unfortunately, these languages force the programmer to state an algorithm in a format suitable for serial execution, thus requiring the extraction of the inherent concurrency for efficient execution upon a multiprocessing system. The detection of concurrency in a sequential language program requires a substantial effort [Elli85, Fish84] due primarily to data dependencies hidden by memory aliasing.

In order to lessen the effort needed for concurrency extraction, the algorithm could be entered in the form of a signal flow graph (SFG). Such graphs are commonly used to express signal processing algorithms [Oppe75] and are quite similar to the dataflow graphs seen previously. An example signal flow graph for a simple second-order filter section (known as a biquad) is shown in Figure 7.2. The connection of the graph nodes (which represent the actual operations) by directed edges (which represent the flow of data values) explicitly depicts the data dependencies present in an algorithm. This makes the detection of operations which may occur simultaneously considerably easier than it was in the case of sequential language representations. The disadvantages to using SFGs are:
Figure 7.2: Biquad Filter Section Signal Flow Graph

- Graphical input devices are needed to enter the SFG into the programming system;
- Updating a large SFG graphically is difficult unless a hierarchical structure is employed.

The problems encountered using the sequential language or SFG representations can be solved by using dataflow programming languages such as VAL [Acke79] or LUCID [Wadg85]. They allow the expression of parallelism in an algorithm, and their model of computation (dataflow graphs) is close to that used for DSP algorithms (signal flow graphs). In addition, the program may be input in a text format which is easily modified and updated using standard text editors. However, these languages have a small user community with correspondingly small libraries of tested programs. Also, the syntax of dataflow languages is rather strange to those familiar with FORTRAN, especially the iterative constructs which would be the most useful for signal processing programs.

The method chosen for expressing signal processing algorithms to PaLS uses both graphical and text entry (Figure 7.3). Signal flow graphs can be entered using a general-purpose graphical editor. The output file from the graphical editor is then processed by a special-purpose program which extracts the signal flow graph description. Alternately, the algorithm may be expressed as an SFG using C language, compiled with a special library of subroutines, and then executed. The output from either of these methods is a list of:
The operations executed in an SFG are represented as nodes and characterized by the following parameters:

- A number identifying the particular SFG node;
- A number representing the time required to perform the node operation;
- A string which describes the node operation.

The flow of data between SFG operators is depicted by edges which possess the

Figure 7.3: Graphical and Textual Entry of Signal Processing Algorithms
following descriptors:

- The identifying number of the node which generates the data (source node);
- The identifying number of the node which accepts the data (sink node);
- The number of data items transferred;
- The direction of the data transfer.

2.2. Partitioning

Once a signal processing algorithm has entered the PaLS system, its SFG is passed to the partitioning phase. The partitioner divides the total algorithm into sub-tasks which are assigned to individual processing elements. Since the main objective of PaLS is to produce signal processing systems with maximum throughput, the partitioning phase attempts to satisfy two opposing goals:

*Minimization of communication:*

Even with IPC links set at their maximum width, transfer of data between DSPs will consume several clock cycles. Therefore, interprocessor communication must be kept to a minimum to reduce the idle time spent by processors waiting for data on which to operate. A trivial solution which minimizes the data transfers assigns the entire SFG to a single DSP and idles the rest (eliminating the need to transfer data). The acceptance of such a solution is prevented by the second goal of the partitioner.

*Balancing computations:*

In any multi-DSP system, throughput is usually determined by the slowest processing element (i.e. the one with the most to do). In order to evenly spread the effort and avoid a bottleneck in a system with \( N \) DSPs, each DSP should ideally receive the fraction \( \frac{1}{N} \) of the total computational load. Except in rare cases, achieving this load balance will require some communication between the individual processors.
Using the goal descriptions just given, one can roughly approximate the search for a good set of algorithm sub-tasks as the search for a partitioning of the underlying SFG such that:

- Each bin in the partition contains nearly equal numbers of nodes (load balancing);
- The number of edges (data transfers) crossing between bins is minimized.

The next section will develop a more precise mathematical framework for this approximation.

2.2.1. Mathematical Framework for the SFG Partitioning Problem

A signal flow graph can be described by the following sets: nodes (representing SFG operations),

\[ N = \{ n_1, n_2, \ldots, n_{|N|} \} \]

node weights (related to the time to perform the operation),

\[ W_N = \{ w_{n_1}, w_{n_2}, \ldots, w_{n_{|N|}} \} \]

edges (representing data transfers between SFG operations),

\[ E = \{ e_1, e_2, \ldots, e_{|E|} \} \]

and edge weights (related to the amount of data being transferred).

\[ W_E = \{ w_{e_1}, w_{e_2}, \ldots, w_{e_{|E|}} \} \]

In addition, two functions \( v_+ \) and \( v_- \) are available which determine the source and sink nodes of a directed edge:

\[ \text{if } e_i = n_j - n_k \text{ then } v_+(e_i) = n_j \text{ and } v_-(e_i) = n_k \]

In order to divide the SFG, a set of bins (representing DSP processing elements) is needed in which to place the nodes:

\[ B = \{ b_1, b_2, \ldots, b_{|B|} \} \]

In turn, each bin \( b_i \) is a set which contains as elements the nodes currently occupying \( b_i \). A function \( \rho \) is provided which determines the bin in which a node presently resides:
\[ \text{if } n_j \in b_i \then p(n_j) = b_i \]

The assignment of subsets of nodes from \( N \) to the bins of \( B \) such that some function is optimized while certain constraints are maintained is termed the \textit{graph partitioning problem} [Kern70]. Unfortunately, this problem is NP-complete so the effort needed to solve it grows exponentially with the size of the graph [Gare79].

Because it is so difficult to find the optimum partition, \textit{heuristic algorithms} are usually employed to find near-optimal solutions. A generic algorithm used to search for good partitions is shown in Figure 7.4. Starting from some initial partitioning of a graph, a search is made to find better solutions (if any) until a certain termination criterion is satisfied. In order to "flesh out" the generic algorithm, four items are needed:

- The \textit{state space} \( \Omega = \{p_1, p_2, \ldots, p_{|P|}\} \) of all possible graph partitionings;
- An \textit{objective function} \( H \) which assigns a figure of merit to each \( p_i \);
- A \textit{search function} \( S \) which generates a new state \( p_j \) given the present state \( p_i \);
- A \textit{termination criterion} which signals when the search should end.

Each of these items will now be examined.

\textbf{2.2.1.1. The Graph Partitioning State Space}

The state space \( \Omega \) for graph partitioning is a set of \( |N| \)-dimensional vectors, where \( |N| \) is the number of nodes in the SFG. Each vector \( p_i \) contains elements \( p_{ij} \) which indicate the bin occupied by node \( n_j \). Thus, each \( p_{ij} \) element may store any one of \( |B| \) possible values because any node may be assigned to any bin regardless of where the other nodes are placed. Therefore, there are \( |B|^{|N|} \) possible states within \( \Omega \).

\textbf{2.2.1.2. The Objective Function}

The objective function \( H \) assigns to each state \( p_i \in \Omega \) a value \( H(p_i) \) so that the algorithm can judge the merit of the given partition.
Figure 7.4: Generic Graph Partitioning Algorithm
The previously discussed tradeoff between load balancing and minimization of communication must be articulated by the $H$ function. An objective function which meets these requirements is:

$$H = H_{\text{cross}} + \alpha_{\text{bal}} H_{\text{bal}}$$

$H_{\text{cross}} = \text{Cost of edges crossing between bins}$

$H_{\text{bal}} = \text{Cost of bin imbalance}$

$\alpha_{\text{bal}} = \text{scale factor relating importance of imbalance to communications}$

The cost of crossing edges is merely the sum of the edge weights which have source and sink nodes in different bins:

$$H_{\text{cross}} = \sum_{e} w_e \left[ 1 - \delta(\rho(v_e^+(e)), \rho(v_e^-(e))) \right]$$

$$\delta(i,j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

The balance cost term is the sum of the deviation of each bin from the ideally balanced configuration:

$$H_{\text{bal}} = \sum_{b} \left[ \frac{1}{|B|} \sum_{N} w_{n} \left( \frac{1}{|B|} \sum_{N} w_{n} \delta(\rho(n_i), b_i) \right) - \sum_{N} w_{n} \delta(\rho(n_i), b_i) \right]$$

$$\frac{1}{|B|} \sum_{N} w_{n} = \text{Ideal total node weight in a bin}$$

$$\sum_{N} w_{n} \delta(\rho(n_i), b_i) = \text{Actual total node weight in bin } b,$$

A slightly different load balancing term can be used in place of the linear/absolute-value method just given:

$$H'_{\text{bal}} = \sum_{b} \left[ \left( \frac{1}{|B|} \sum_{N} w_{n} \delta(\rho(n_i), b_i) \right)^2 \right]$$

This load balancing term punishes deviations from ideal balance more harshly than the previous method.
2.2.1.3. Search Functions

Given a particular graph partition \( p_i \), there exists a set or \textit{neighborhood} of adjacent states \( \Lambda(p_i) \) reachable by perturbing \( p_i \). The search function \( S \) uses some criterion to select a new state to visit from the set \( \Lambda(p_i) \).

There are many ways in which to define the neighborhood set of a state, some of the simplest being:

\textbf{Single node/bin change:}

\[ \Lambda(p_i) = \{ p_j \mid p_{jk} \neq p_{ik} \text{ for } k = 1 \text{ or } 2 \ldots \text{ or } \mid N \mid \text{ and } p_{jl} = p_{il} \text{ for } l \neq k \} \]

\textbf{Node interchange:}

\[ \Lambda(p_i) = \{ p_j \mid p_{jk} = p_{jm} \text{ and } p_{jm} = p_{ik} \text{ for } k = 1 \text{ or } 2 \ldots \text{ or } \mid N \mid \text{ and } m \neq k \text{ and } p_{jl} = p_{il} \text{ for } l \neq k, m \} \]

The method of selecting the next state to visit from \( \Lambda(p_i) \) is a primary distinction between graph partitioning algorithms, as will be seen later. Several choices for the \( S \) function are:

- Random choice: \( S(p_j) = \text{Rand} \{ \Lambda(p_i) \} \);
- Any improvement in \( H \): \( S(p_j) = \text{Rand} \{ p_j \mid H(p_j) < H(p_i) \text{ and } p_j \in \Lambda(p_i) \} \);
- The most improvement in \( H \): \( S(p_j) = \text{Min}_{H(p_j)} \{ p_j \mid p_j \in \Lambda(p_i) \} \).

2.2.1.4. Stopping Criteria

As was the case with search functions, rules for terminating the partitioning algorithm are varied. Perhaps the simplest is to quit once a state \( p_i \) is reached whose neighbors all have higher objective function values (i.e. \( p_i \) is a local minimum). Improvements on this technique will be examined in subsequent sections.
2.2.2. Some Real Graph Partitioning Algorithms

Instantiating the generic subroutines found in Figure 7.4 with specific algorithms allows us to examine several real graph partitioning techniques.

2.2.2.1. Greedy Partitioning

Greedy partitioning, as its name implies, begins with an initial partitioning of a graph and traverses a set of successively adjacent partitions while trying to decrease $H$ as much as possible at each step. More formally, the algorithm proceeds from an initial partitioning $p_0$ to successive partitionings $p_1, p_2, \ldots, p_{\text{loc.opt}}$ such that:

- $p_i \in \Lambda(p_{i-1}) \; \forall \; i > 0$
- $H(p_i) < H(p_{i-1}) \; \forall \; i > 0$
- $H(p_i) \leq H(\lambda_j) \; \forall \; i > 0 \; \text{and} \; \forall \; \lambda_j \in \Lambda(p_{i-1})$

The adjacent states $\Lambda(p_i)$ are usually defined as those which can be reached using a single node movement to a different bin, or by interchanging two nodes between bins. The greedy partitioner stops its search when a state is reached whose neighbors all have higher $H$ values.

The greedy partitioner operates as follows:

**Step 1 - Initialize greedy search:**

Place the starting partition into $p_0$ and set $k \leftarrow 0$.

**Step 2 - Initialize search of adjacent states:**

Set $H_{\text{min}} \leftarrow H(p_k)$.

**Step 3 - Search for the best immediate move:**

For each $\lambda_j \in \Lambda(p_k)$, if $H(\lambda_j) < H_{\text{min}}$ then set $H_{\text{min}} \leftarrow H(\lambda_j)$ and $\lambda_{\text{min}} \leftarrow \lambda_j$.

**Step 4 - Check for termination:**

If $H_{\text{min}} = H(p_k)$, then at least a local minimum has been found, so stop searching.

Otherwise, set $p_{k+1} \leftarrow \lambda_{\text{min}}$ and $k \leftarrow k + 1$ and return to step 2.
The advantages of the greedy partitioner are its simplicity and the speed with which it converges to a solution. Unfortunately, the solution is usually of low quality. This can be seen by examining Figure 7.5, which is a graph of $H(p)$ versus $p$ and assumes $\Lambda(p_i) = \{p_i-1, p_i+1\}$ (so we can use a linear abscissa). Obviously, in Figure 7.5 (a) the greedy technique will find the true global optimum since the state space is convex with respect to $H$. This is a rarity when dealing with real-life graph partitioning problems, in which case Figure 7.5 (b) is more indicative. Here, the greedy algorithm gets "stuck" in a local optima unless luck intervenes and the starting partition resides in the potential well containing the global optimum.

2.2.2.2. The Kernighan-Lin Algorithm

The greedy algorithm fails to find the globally optimal partition because it is unable to work its way out of local minima. The Kernighan-Lin algorithm [Kern70] solves this problem by employing hill climbing, i.e. it visits states which temporarily increase $H$. This is done by

![Figure 7.5: The Failure of Greedy Partitioning](image)
looking for a sequence of node movements which brings about a total decrease in $H$ although some of the individual moves may increase $H$. The kernel of the algorithm operates as follows:

Step 1 - Initialize:

Set $M = \{n_1, n_2, \ldots, n_{|N|}\}$, $k = 1$, and let $p_0$ be the starting graph partition.

Step 2 - Find the best move from the current partition:

Determine the node $n_i \in M$ and associated destination bin $b_j \neq \rho(n_i)$ such that the move $\rho(n_i) - b_j$ causes the largest decrease $\Delta_k$ from $H(p_{k-1})$. It is possible that $\Delta_k \geq 0$ (when hill climbing occurs).

Step 3 - Make the move found in step 2:

Perturb partition $p_{k-1}$ by moving $n_i$ to bin $b_j$ to create $p_k$, and set $M = M - \{n_i\}$ and $k = k + 1$.

Step 4 - Generate new partitions until every node has been moved:

Return to step 2 until $M = \emptyset$.

Step 5 - Select the best of the generated partitions:

Determine the $K$ that minimizes $\sum_{k=1}^{K} \Delta_k$.

Step 6 - Check for termination:

If $K = 0$, then the initial partition $p_0$ is taken as the optimum and the algorithm terminates. Otherwise, the modified partition $p_K$ becomes the new optimum with an objective value of $H(p_K) = H(p_0) + \sum_{k=1}^{K} \Delta_k$. Return to step 1 using $p_K$ as the initial partition.

An example of how the Kernighan-Lin algorithm partitions a graph is shown in Figure 7.6. The graph has seven nodes of weight 1 connected by nine edges, three of which have weights of 2. The $H$ function employed in the example is given by Eqn. (7.1),
Eqn. (7.2), and Eqn. (7.3) with $\alpha_{bal} = 0.7$.

The Kernighan-Lin algorithm has been used previously to perform partitioning of algorithms [Berm85]. The ability to climb out of local optima gives it improved performance over the greedy partitioner (although there is no guarantee the global optimum will be found). Even so, the Kernighan-Lin method retains a vestige of the greedy algorithm within its inner-loop in that each move added to the sequence is the best available at the time. Trapping in local optima is only forestalled by the requirement that $|N|$ moves must be made at each iteration. If trapping does occur, improved solutions can still be found by starting the Kernighan-Lin algorithm with different initial partitions and selecting the best of the final answers.

2.2.2.3. Partitioning by Simulated Annealing

Simulated annealing [Kirk83] is a new technique for solving combinatorial problems (like graph partitioning) by viewing them as physical systems seeking a low energy state. The annealing of metals or the growing of crystals starts with a molten, disordered collection of particles at a high temperature which, through thermal agitation combined with a slow cooling process, arrives at a highly ordered solid state with low internal energy. This can be likened to the random movement of graph nodes between bins until a good partition with a low value of $H$ is obtained. In fact, there is a close analogy between the forces at work during the partitioning of a graph and the cooling of molten liquids:

- **Ordered arrangement of nodes** $\rightarrow$ **Ordered arrangement of particles**
- **Data transfer edges** $\rightarrow$ **Attractive forces (Van der Wall, electrostatic)**
- **Load balancing** $\rightarrow$ **Repulsive forces (inter-atomic spacing)**

How is the cooling process of physical systems simulated in order to partition graphs?

Looking at physical systems, the elements of which they are built can be spatially arranged in a practically infinite number of ways. These arrangements can be collected to create the state
\[ p_0 = \{ \{ n_1 \} \{ n_2 n_3 n_4 n_5 n_6 n_7 \} \} \quad M = \{ n_1 n_2 n_3 n_4 n_5 n_6 n_7 \} \]
\[ \Delta_1 = \min\{ \delta_1=0.4 \quad \delta_2=0.6 \quad \delta_3=1.6 \quad \delta_4=1.6 \quad \delta_5=4.6 \quad \delta_6=2.6 \quad \delta_7=1.6 \} = 0.4 = \delta_1 : \rho(n_1) - b_2 \]

\[ p_1 = \{ \{ \} \{ n_1 n_2 n_3 n_4 n_5 n_6 n_7 \} \} \quad M = \{ n_2 n_3 n_4 n_5 n_6 n_7 \} \]
\[ \Delta_2 = \min\{ \delta_2=0.6 \quad \delta_3=1.6 \quad \delta_4=3.6 \quad \delta_5=4.6 \quad \delta_6=2.6 \quad \delta_7=1.6 \} = 0.6 = \delta_2 : \rho(n_2) - b_1 \]

\[ p_2 = \{ \{ n_2 \} \{ n_1 n_2 n_3 n_4 n_5 n_6 n_7 \} \} \quad M = \{ n_3 n_4 n_5 n_6 n_7 \} \]
\[ \Delta_3 = \min\{ \delta_3=1.6 \quad \delta_4=3.6 \quad \delta_5=2.6 \quad \delta_6=0.6 \quad \delta_7=1.6 \} = 0.6 = \delta_6 : \rho(n_6) - b_1 \]

\[ p_3 = \{ \{ n_2 n_6 \} \{ n_1 n_3 n_4 n_5 n_7 \} \} \quad M = \{ n_3 n_4 n_5 n_7 \} \]
\[ \Delta_4 = \min\{ \delta_3=1.6 \quad \delta_4=1.6 \quad \delta_5=-1.4 \quad \delta_7=1.6 \} = -1.4 = \delta_5 : \rho(n_5) - b_1 \]

\[ p_4 = \{ \{ n_3 n_5 n_6 \} \{ n_1 n_3 n_4 n_7 \} \} \quad M = \{ n_3 n_4 n_7 \} \]
\[ \Delta_5 = \min\{ \delta_3=1.0 \quad \delta_4=3.0 \quad \delta_7=-1.0 \} = -1.0 = \delta_7 : \rho(n_7) - b_1 \]

\[ p_5 = \{ \{ n_2 n_5 n_6 n_7 \} \{ n_1 n_3 n_4 \} \} \quad M = \{ n_3 n_4 \} \]
\[ \Delta_6 = \min\{ \delta_3=2.4 \quad \delta_4=2.4 \} = 2.4 = \delta_3 : \rho(n_3) - b_1 \]

\[ p_6 = \{ \{ n_2 n_3 n_5 n_6 n_7 \} \{ n_1 n_4 \} \} \quad M = \{ n_4 \} \]
\[ \Delta_7 = \min\{ \delta_4=-1.6 \} = -1.6 = \delta_4 : \rho(n_4) - b_1 \]

\[ p_7 = \{ \{ n_2 n_3 n_4 n_5 n_6 n_7 \} \{ n_1 \} \} \]

K = 5

\[ p_{\text{final}} = \{ \{ n_2 n_5 n_6 n_7 \} \{ n_1 n_3 n_4 \} \} \]

Figure 7.6: An Example of Kernighan-Lin Graph Partitioning
space

\[ S = \{ s_1, s_2, \ldots, s_{\text{large}} \} \]

where each \( s_i \) is a distinct state. For graphs, the analog to \( S \) is \( P \), the set of possible partitionings.

Given \( S \), each state \( s_i \) can be assigned a value \( E(s_i) \) corresponding to the amount of energy stored in the arrangement of particles due to attractive and repulsive forces. A state density function [Reif65], \( \eta_S(E) \), specifies the number of states with energy between \( E_1 \) and \( E_1 + \Delta E \) as:

\[
\frac{E_1 + \Delta E}{E_1} \int_{E_1}^{E_1 + \Delta E} \eta_S(E) dE
\]

Analogously, each partitioning \( p_i \) in \( P \) is assigned the value \( H(p_i) \), allowing us to arrive at an equivalent partitioning density function:

\[
\frac{H_1 + \Delta H}{H_1} \int_{H_1}^{H_1 + \Delta H} \eta_P(H) dH = \text{number of partitionings with values between } H_1 \text{ and } H_1 + \Delta H
\]

The function \( \eta_S(E) \) describes the density of states, but determining the probability that the system will ever occupy a given state requires an additional factor. Physical systems composed of many particles typically obey a Boltzmann distribution when in equilibrium [Feyn63], so the probability of a system having an energy between \( E_1 \) and \( E_1 + \Delta E \) is:

\[
\frac{E_1 + \Delta E}{E_1} \int_{E_1}^{E_1 + \Delta E} d_S(E) dE = \frac{\frac{E_1 + \Delta E}{E_1} \int_{E_1}^{E_1 + \Delta E} \eta_S(E) e^{-\frac{E}{kT}} dE}{\frac{E}{kT} \int_{-\infty}^{\infty} \eta_S(E) e^{-\frac{E}{kT}} dE}
\]

where \( d_S(E) \) is the probability distribution of the states over energy. The ratio between the densities at two different energies is:
This equation shows that as temperature decreases $(T \rightarrow 0)$, the probability of the system being in a low energy state increases relative to the chance of occupying a more disordered, high energy state. We must realize that Eqn. (7.4) is only true when a system is in equilibrium at a particular temperature (i.e. $\partial d_\xi / \partial t = 0$). But since the system is continually fluctuating due to random thermal effects, equilibrium can only be maintained if the number of transitions between energy levels is equal:

$$d_\xi (E_1) \times \pi \{ E_1 - E_2 \} = d_\xi (E_2) \times \pi \{ E_2 - E_1 \}$$

which leads to:

$$\frac{d_\xi (E_1)}{d_\xi (E_2)} = \frac{\pi \{ E_2 - E_1 \}}{\pi \{ E_1 - E_2 \}} = \frac{\eta_\xi (E_1)}{\eta_\xi (E_2)} e^{- \frac{E_1 - E_2}{kT}}$$

In order to make the graph partitioning process act like the cooling of a liquid, the equilibrium distribution of partition values must follow the equation:

$$\frac{d_p (H_1)}{d_p (H_2)} = \frac{\pi \{ H_2 - H_1 \}}{\pi \{ H_1 - H_2 \}} = \frac{\eta_p (H_1)}{\eta_p (H_2)} e^{\frac{H_1 - H_2}{T}}$$

where the Boltzman scaling constant $k$ has been dropped. The transition probability is modeled as:

$$\pi \{ H_1 - H_2 \} = \pi_{\text{choose}} \{ H_1 - H_2 \} \times \pi_{\text{accept}} \{ H_1 - H_2 \}$$

Here, $\pi_{\text{choose}}$ is the probability of choosing to attempt a transition from a partition with value $H_1$ to a partition with value $H_2$, and it is determined by:

$$\pi_{\text{choose}} \{ H_1 - H_2 \} = \frac{\text{number of transitions leading from } H_1 \text{ to } H_2}{\text{Total number of transitions leading from } H_1}$$

$$= \frac{t_{H_1 \rightarrow H_2}}{\eta_p (H_1) \times |N| \times |B - 1|}$$

The probability of accepting the chosen move is given by $\pi_{\text{accept}}$, which is:
Using $\pi_{\text{choose}}$ and $\pi_{\text{accept}}$ we find:

$$
\pi_{\text{accept}} \{ H_1 - H_2 \} = \begin{cases} 
1 & \text{if } H_1 \geq H_2 \\
\frac{e^{-\frac{H_2 - H_1}{T}}}{\eta_p(H_2) \times |N| \times |B - 1|} & \text{if } H_1 < H_2
\end{cases}
$$

Thus, the equilibrium distribution of a graph partitioning problem can be made to look like that of a physical system if $t_{H_1 - H_2} = t_{H_2 - H_1}$. This is guaranteed so long as the moves used to perturb graph partitions are reversible such that the state transitions $p_i \rightarrow p_j$ and $p_j \rightarrow p_i$ are both possible.

Now that the rules for governing the transitions between graph partitions are established, the simulated annealing method for graph partitioning can be shown:

**Step 1 - Initialize:**

Generate a random partition $P_0$ of the graph and set $k \rightarrow 1$. Also initialize the temperature $T$ and the cooling coefficient $\alpha_T$ to their starting values.

**Step 2 - Perturb the partition:**

Generate a new partition $P_k$ by randomly selecting a node $n_i$ and moving it to a randomly selected destination bin $b_j$. Then set $\Delta = H(P_k) - H(P_{k-1})$.

**Step 3 - Check new partition:**

If the new partition is an improvement ($\Delta \leq 0$), accept $P_k$. Else perform probabilistic hill climbing by generating a random number $r$ in the interval $[0,1)$ and comparing it to $e^{-\frac{\Delta}{T}}$. If $r < e^{-\frac{\Delta}{T}}$, then accept $P_k$. Otherwise, do not hill climb and set $P_k = P_{k-1}$.

**Step 4 - Check for equilibrium:**

Set $k \rightarrow k + 1$ and check to see if the probability density has reached equilibrium for the
current temperature. If not, return to step 2 until equilibrium is attained.

*Step 5 - Cool:*

Decrease the temperature by \( \alpha_T : T \rightarrow T \). If the partitioning is frozen, then terminate the algorithm. Otherwise, set \( p_0 = p_{k-1} \) and \( k = 1 \), and return to step 2.

From the algorithm shown above, node movements which cause large increases in the objective function are readily accepted in the beginning of the annealing process when \( T \) is large. As \( T \) decreases, the graph leaves the molten state and hill climbing becomes less probable. Finally, at sufficiently low temperatures the graph becomes frozen into a particular partition which is hopefully well ordered with a low value for \( H \). Figure 7.7 shows several snapshots at different temperatures of a graph undergoing simulated annealing.

The annealing algorithm as shown leaves several points unanswered:

- At what temperature is the system molten enough to start cooling?
- How fast should the temperature decrease as set by \( \alpha_T \)?
- How is it determined when equilibrium is reached at a specific temperature?
- How is it known when the partitions are frozen?

There has been much discussion in the literature on these points from which the following guidelines are derived [Rome85, Aart85]:

* Determination of the starting temperature:

The initial temperature should be set at a level which makes even the worst partitions probable, corresponding to a molten state. Since it is as difficult to determine the worst partition as the best one, the following method is used: the temperature is raised in steps and a number of partition perturbations are performed at each temperature in order to find the average value of the configurations. When the average value no longer increases with increasing temperature, then the system is completely
Figure 7.7: Annealing a Graph
liquid and the cooling process may begin.

Magnitude of temperature decreases:

The temperature decrease factor is usually set in the range $0.8 \leq \alpha_T < 1.0$, with $\alpha_T = 0.95$ being very common. It is important that $\alpha_T$ not be too small or the temperature will decrease too rapidly and result in a poor final solution. This is analogous to the formation of dislocations and defects during crystal growth. On the other hand, too slight a temperature decrease slows the approach to the freezing point with a corresponding increase in the number of computations to be performed.

Detection of thermal equilibrium:

Thermal equilibrium is reached when the average value of the generated partitions is no longer decreasing for a set temperature. This condition can be difficult to detect, so a simplification is to assume equilibrium is established once a certain number of perturbations have been generated without seeing a new minimum value for $H$. The number of perturbations is usually set as some small multiple of the product $|N| \times |B|$.

Detection of solidification:

If the average value of the generated partitions has not decreased after some number of temperature reductions, then the partitions are assumed to be solidified and the algorithm can terminate.

From the previous paragraphs, it can be seen that simulated annealing relies on many parameters with no strict rules for how they are set. We have slightly alleviated this problem by taking a new approach. Since so many of the parameters rely in some way upon the average value of the partitions generated at a certain temperature, we analytically predict the relationship of $<H>$ versus $T$ for a given graph (see Appendix B). With the correct assumptions, the relationship has the simple form:
\[ <H(T)> = A - \frac{B}{T} \geq 0 \]

where \( A \) and \( B \) are parameters which are easily calculated from the specific graph being partitioned. This formula permits us to predict:

**The starting temperature:**

The temperature \( T_0 \) at which the graph is molten enough to begin the cooling process can be found by solving:

\[ <H(T_0)> = A - \frac{B}{T_0} = \beta_{T_0}A \]

where \( \beta_{T_0} < 1 \). In our system, we arbitrarily use \( \beta_{T_0} = 0.99 \).

**The rate of temperature decrease:**

The following equation is used to set the \( \alpha_T \) factor such that the system does not cool too rapidly:

\[ <H(\alpha_T T)> = \beta_T <H(T)> \]

where \( \beta_T < 1 \). In our system, \( \beta_T \) is set in the range \([0.9, 0.99]\).

**Thermal equilibrium:**

The onset of thermal equilibrium can be detected when the average \( H \) of the randomly generated partitions matches with \( <H(T)> \) from the analytical curve. This technique has been error prone, so we use the more common method presented previously.

**The freezing point:**

The temperature \( T_f \) at which the partitions are frozen is determined from:

\[ <H(T_f)> = 0 \]

In actuality, there is usually more improvement which can be made at this point since the analytical model deviates from reality at extremely low temperatures. Therefore, we use \( T_f \) as a threshold after which a faster algorithm (such as Kernighan-Lin or a
2.2.3. A Comparison of the Graph Partitioning Algorithms

A comparison of the performance of the previously presented algorithms is given in Table 7.1. The data was gathered by tri-partitioning randomly generated graphs with a predetermined number of nodes and edges $(|N|/|E|)$ using the previously discussed algorithms. The objective function used for each algorithm was Eqn. (7.1) with bin balance maintained using Eqn. (7.3) and $\alpha_{bal} = 0.4$.

Of particular concern to us was the quality of the solutions and the length of time needed to generate them using each algorithm. The number of crossing edges which remained upon termination of an algorithm was used as the main criterion of solution quality. The average values and standard deviations for each ensemble of graphs are given in the first half of Table 7.1. As suspected, the greedy partitioner gave the worst results while simulated annealing typically arrived at the best solutions. The Kernighan-Lin algorithm achieved results which were only marginally worse than the annealing method.

The time required to converge to a particular partitioning shows a trend opposite that found concerning solution quality. Here, greedy algorithms are naturally the fastest, using approximately half the time needed by the Kernighan-Lin technique. The simulated annealing approach used the most time by far, reflecting its use of randomness to drift toward a

| $|N|/|E|$ | Crossing Edges | Execution Time |
|-------|----------------|----------------|
|       | Greedy | Ker/Lin | Annealing | Greedy | Ker/Lin | Annealing |
| 50/100 | 28.0±2.4 | 24.3±2.0 | 22.9±2.2 | 7.7±1.1 | 9.5±1.8 | 229.1±22.7 |
| 100/200 | 54.5±4.9 | 47.4±3.4 | 45.4±2.0 | 22.6±2.8 | 37.8±8.1 | 462.6±29.3 |
| 150/225 | 56.9±3.9 | 41.9±3.7 | 37.2±2.4 | 33.8±2.6 | 78.8±20.8 | 651.0±53.3 |
| 200/400 | 110.4±5.9 | 91.6±4.3 | 86.3±3.5 | 80.6±7.8 | 152.2±40.8 | 1012.0±76.4 |
solution. The only consolation which can be had concerning the speed of the annealing method is that the run-times grow only linearly as the number of graph nodes increases, whereas the Kernighan-Lin algorithm displays a quadratic trend.

It would seem that the Kernighan-Lin algorithm offers the best trade-off between solution quality and speed of convergence. The performance data must be viewed skeptically, however, since random data does not maximally stress an algorithm [Ullm76]. This is a particular concern for us, since flow graphs of signal processing algorithms are usually very regular. The results of partitioning such regular SFGs will be presented later.

2.3. Link Assignment

Once a signal flow graph for an algorithm has been partitioned, the flow of data represented by the edges crossing between bins must be mapped into physical connections between the DSP chips in a network. Naturally, it is desirable to build links which have enough bandwidth to transfer data without creating delays which slow the total operation of the DSP system. Counterbalancing this desire is the cost of implementing the links and the limitations on total bandwidth in the system. The link assignment phase attempts to allocate the fixed total bandwidth provided by the IPC of each DSP element such that the smallest links are used without creating a communications bottleneck. The mathematical underpinning of this optimization problem is presented next.

2.3.1. Mathematical Framework for the Link Assignment Problem

The output of the partitioning phase is a signal flow graph divided between bins representing DSP elements. The amount of data flowing from processor $i$ to processor $j$ is:

$$D_{ij} = \sum_{e} w_e \delta(b_i, \rho(v(e))) \delta(b_j, \rho(v(e))) \left[ 1 - \delta(b_i, b_j) \right]$$

Getting this data between processors $i$ and $j$ requires a physical link which transfers $W_{ij}$ bits in parallel and has a corresponding transfer rate $r_{ij}$ of:
where \( K_r \) is a constant depending upon the DSP and IPC. After partitioning there is no further control over \( D_{ij} \), so increasing the data transfer rate requires making \( W_{ij} \) as large as possible. However, the IPC receivers and transmitters are limited to operating on a maximum of \( W_{IPC} \) bits in parallel (\( W_{IPC} = 16 \) in our case), so the following constraints must be satisfied:

\[
\sum_{j=1}^{B} W_{ij} \leq W_{IPC} \quad \text{for } 1 \leq i \leq |B| \quad (\text{IPC output constraint at transmitting DSP } i)
\]

\[
\sum_{i=1}^{n} W_{ij} \leq W_{IPC} \quad \text{for } 1 \leq j \leq |B| \quad (\text{IPC input constraint at receiving DSP } j)
\]

(7.5)

\[W_{ij} \in \{ 0, 1, 2, 4, \ldots, \frac{W_{IPC}}{2}, W_{IPC} \}\]

The last constraint arises from the construction of the IPC, which allows only link widths which are powers of two.

Because many signal processing systems execute an algorithm repetitively, the sampling rate of such a system is determined by its slowest component. If we assume each IPC link must transfer its assigned load of data during every repetition, then the data transfer time could become the factor which limits total system throughput. Therefore, the objective function chosen for the link assignment problem attempts to maximize the minimum link transfer rate:

\[
R_{opt} = \text{Maximize}\{ \min_{i,j} \{ r_{i,j} \} \} \quad (7.6)
\]

This objective function is usually acceptable, but there are pathological cases under which it performs poorly. For example, a highly utilized link may have the transmissions scheduled such that there is sufficient time between data bursts to prevent link blockage even with a narrow channel. Another channel might carry only two transmissions, but if they must occur close to one another the link will always block on the second transmission and create delays. Our objective function would favor assigning more bits to the first channel due to the total
amount of traffic it carries, but it is the second link which would benefit most from the extra
bandwidth. Situations like this are hopefully corrected by the scheduling phase of PaLS, so
the objective function in Eqn. (7.6) is used despite the potential problem.

2.3.2. Optimal Solution of the Link Assignment Problem

Finding the optimal solution to the problem posed in Eqn. (7.5) and Eqn. (7.6) would
appear to require a combined integer and goal programming approach [Hill67]. Luckily, the
problem is easily solved with a simple algorithm. To justify this technique, transform the
problem given by Eqn. (7.5) and Eqn. (7.6) into a goal-program:

\[
\text{Problem G: Maximize } z \\
\text{Such that: } \frac{W_{ij}}{D_{ij}} - z \geq 0 \text{ for } 1 \leq i, j \leq |B| \text{ and } i \neq j \\
\sum_{i=1}^{(|B|)} W_{ij} \leq W_{iPC} \text{ for } 1 \leq i \leq |B| \\
\sum_{j=1}^{(|B|)} W_{ij} \leq W_{iPC} \text{ for } 1 \leq j \leq |B| \\
W_{ij} \in \{0, 1, 2, 4, \ldots, \frac{W_{iPC}}{2}, W_{iPC}\} 
\] (7.7)

In problem G, the variable \( z \) is always equal to the smallest transfer rate in the present
network as defined by the set of \( W_{ij} \) values. The system throughput can be increased only by
increasing \( z \) while remaining within the constraints set by Eqn. (7.7).

The algorithm to solve problem G is:

\text{Step 1 - Initialization:}

\text{Start with a feasible solution (i.e. one which satisfies all constraints in Eqn. (7.7)) by}
\text{setting } W_{ij} = 0 \text{ for all } i \text{ and } j .

\text{Step 2 - Determine the minimum transfer rate:}

\text{Set } z = \min_{i,j} \left\{ \frac{W_{ij}}{D_{ij}} \right\} .
Step 3 - Find bottlenecks:

Find any link $W_{kl}$ such that $W_{kl} - z = 0$.

Step 4 - Remove bottlenecks:

The constraint found in step 3 is tight, so $W_{kl}$ must necessarily be increased if $z$ is to increase. Determine the increase to $W_{kl}$ as follows:

$$W_{kl}^+ = \begin{cases} 1 & \text{if } W_{kl} = 0 \\ 2W_{kl} & \text{if } W_{kl} > 0 \end{cases}$$

If $W_{kl}$ is feasible according to Eqn. (7.7), then set $W_{kl} = W_{kl}^+$ and return to step 2.

Otherwise the bottleneck cannot be relieved, so terminate the algorithm.

An example of the use of the algorithm on a link assignment problem for a graph tri-partition is given in Figure 7.8.

The solution found by the above algorithm maximizes the throughput of the worst channel, but in doing so it roughly equalizes the transfer rate on all channels. This occurs because the algorithm only increases the size of a link if it is currently causing a bottleneck. Therefore, if any $W_{ij}$ in the final solution was decreased to a lower value $W_{ij}'$, a new bottleneck would appear and the total throughput rate would possibly decrease. Mathematically stated:

$$\frac{W_{ij}}{D_{ij}} \leq z < \frac{W_{ij}}{D_{ij}} \quad \text{for non-bottleneck links}$$

$$\frac{W_{ij}}{D_{ij}} < z = \frac{W_{ij}}{D_{ij}} \quad \text{for bottleneck links}$$

The algorithm just presented also finds the optimal solution to problem $G$. This can be proved by the following argument. Assume the solution found by the link assignment algorithm is:

$$S = \{ W_{1,2}, W_{1,3}, \ldots, W_{|\mu|,|\mu|-1} \} \text{ with } R_{opt} = z$$

and that a better solution exists:
Removing bottleneck: $z = r_{1,2} = 0.00$: $W_{1,2} = 1$
Removing bottleneck: $z = r_{2,1} = 0.00$: $W_{2,1} = 1$
Removing bottleneck: $z = r_{2,3} = 0.00$: $W_{2,3} = 1$
Removing bottleneck: $z = r_{3,1} = 0.00$: $W_{3,1} = 1$
Removing bottleneck: $z = r_{3,2} = 0.00$: $W_{3,2} = 1$
Removing bottleneck: $z = r_{3,1} = 0.25$: $W_{3,1} = 2$
Removing bottleneck: $z = r_{1,2} = 0.33$: $W_{1,2} = 2$
Removing bottleneck: $z = r_{2,1} = 0.50$: $W_{2,1} = 2$
Removing bottleneck: $z = r_{3,1} = 0.50$: $W_{3,1} = 4$
Removing bottleneck: $z = r_{3,2} = 0.50$: $W_{3,2} = 2$
Removing bottleneck: $z = r_{1,2} = 0.67$: $W_{1,2} = 4$
Removing bottleneck: $z = r_{2,1} = 1.00$: $W_{2,1} = 4$
Removing bottleneck: $z = r_{2,3} = 1.00$: $W_{2,3} = 2$
Removing bottleneck: $z = r_{3,1} = 1.00$: $W_{3,1} = 8$
Removing bottleneck: $z = r_{3,2} = 1.00$: $W_{3,2} = 4$
Removing bottleneck: $z = r_{1,2} = 1.33$: $W_{1,2} = 8$
Removing bottleneck: $z = r_{2,1} = 2.00$: $W_{2,1} = 8$
Removing bottleneck: $z = r_{2,3} = 2.00$: $W_{2,3} = 4$
Unremovable bottleneck: $R_{opt} = z = r_{3,1} = 2.00$

$D_{1,2}=3 \ W_{1,2}=8 \ r_{1,2}=2.67$
$D_{2,1}=2 \ W_{2,1}=8 \ r_{2,1}=4.00$
$D_{2,3}=1 \ W_{2,3}=4 \ r_{2,3}=4.00$
$D_{3,1}=4 \ W_{3,1}=8 \ r_{3,1}=2.00$
$D_{3,2}=2 \ W_{3,2}=4 \ r_{3,2}=2.00$

**Figure 7.8:** A Link Assignment Example
\[ S' = \{ W'_{1,2}, W'_{1,3}, \ldots, W'_{|B|,|B|-1} \} \] with \( R'_{\text{opt}} = z' > z \)

If \( W'_{ij} < W_{ij} \) for any \( i \) and \( j \), then:

\[
\frac{W'_{ij}}{D_{ij}} \leq \frac{W_{ij}}{D_{ij}} \leq z < \frac{W_{ij}}{D_{ij}} \quad \text{for non-bottleneck links}
\]

\[
\frac{W'_{ij}}{D_{ij}} \leq \frac{W_{ij}}{D_{ij}} < z = \frac{W_{ij}}{D_{ij}} \quad \text{for bottleneck links}
\]

But:

\[
z' \leq \frac{W'_{ij}}{D_{ij}}
\]

which contradicts the assumption that \( z' > z \). So it must be true that \( W'_{ij} \geq W_{ij} \) for all \( i \) and \( j \), and in particular:

\[
W'_{ij} > W_{ij} \quad \text{for all bottleneck links in } S
\]

Now build a new solution \( S'' \) composed of the non-bottleneck links of \( S \) and with the remainder coming from \( S' \):

\[
S'' = \left\{ \begin{array}{c|c}
W'_{ij} & W''_{ij} = \begin{cases} 
W_{ij} & \text{if } \frac{W_{ij}}{D_{ij}} - z > 0 \\
W'_{ij} & \text{otherwise}
\end{cases}
\end{array} \right\}
\]

\( S'' \) must be feasible since it can be formed from \( S' \) by reducing the size of some links, and this action never alters feasibility. But, \( S'' \) is identical to \( S \) with the exception that all the bottleneck links have larger widths in \( S'' \) (having been replaced with the link widths from \( S' \)). But \( S \) would only be the final output of the algorithm if a bottleneck was found which could not be expanded due to the constraints. This contradiction proves that the assumption of the existence of \( S' \) was incorrect. Therefore, the algorithm produces the optimal solution to problem \( G \).

In order to speed convergence of the link assignment algorithm, the initial assignment of all link widths to zero can be made as follows:
\[ W_{ij} = \max \left\{ 0, 1, 2, \ldots, \frac{W_{ipc}}{2}, W_{jpc} \right\} \text{ such that } W_{ij} < \frac{D_{ij}}{\sum_k D_{kij}} \text{ and } W_{ij} < \frac{D_{ij}}{\sum_l D_{il}} \]

An additional refinement to the algorithm improves the solution when multiple tight constraints exist upon termination. Giving priority to increasing the size of the smallest bottlenecked links will permit the greatest number to be unblocked. The end result is a network with the same overall throughput, but with fewer total links which impose the speed limit. This reduces the overall probability of link blockages.

2.4. Scheduling

At this point, the output from the partitioning and link assignment phases consists of:

- An assignment of computational nodes to DSP elements;
- An assignment of physical link widths connecting the DSP elements.

Taken together with the original SFG, these items define a set of computational and communication tasks which must be scheduled for execution upon the DSP elements and interconnecting links. The PaLS scheduler tries to arrange the order in which the tasks are executed so that the throughput of the system is maximized. In doing so, the scheduler must operate within the constraints imposed by the signal processing algorithm as well as the preceding phases of the automatic programming system.

2.4.1. A Mathematical Framework for the Scheduling Problem

Taken together, the signal flow graph, its partitioning, and the assignment of physical link widths define a set of tasks \( T = \{ t_1, t_2, \ldots, t_{|\mathcal{T}|} \} \) with the following attributes (Figure 7.9):

**Assigned Resources:**

Each node in an SFG represents a computational task which must be executed by the DSP element corresponding to the bin containing the node. In addition, the SFG
Algorithm SFG:

Partitioning and link assignment:

Figure 7.9: The PaLS Scheduling Problem
edges crossing between bins symbolize data communication tasks to be executed using the resources of the IPC links. A DSP network contains $|B|$ processor resources and up to $16|B|$ link resources, so for each resource $r_i$, there exists a set of tasks $T_r$ which will use that resource.

Task Duration:

Each task $t_i$ requires a certain amount of time $\tau_{t_i}$ to complete. Computing tasks have a duration proportional to the weight of their associated SFG nodes (i.e. $\tau_{t_i} \propto w_e$). The duration of a data transfer is determined by the amount of data passed over the link and the transfer delay of the link (i.e. $\tau_{t_i} \propto \frac{w_e}{W_{jk}}$).

Antecedent and Successor Tasks:

Some of the tasks require the results generated by other tasks. Therefore, a task $t_i$ cannot be executed until all of the tasks in its antecedent set, $A_{t_i}$, have been completed. For example, in Figure 7.9 computational task $t_{10}$ cannot start until the results from computational task $t_9$ and data transfer task $t_6$ are available. The converse of $A_{t_i}$ is $S_{t_i}$, the set of all successor tasks which are dependent upon a result generated by $t_i$.

Assume that no task can start before time 0 and the entire signal processing algorithm is to be executed only once. Then the objective of the scheduler would be to arrange the tasks so as to minimize the time at which the last task is completed (i.e. minimize the total execution time for the algorithm). This scheduling of tasks for a DSP network can be mathematically stated as the assignment of a starting time $s_{t_i}$ to each task $t_i$ so as to solve the following goal program:
Problem S1: Minimize $z$

Such that:  
\[ z - (s_i + \tau_i) \geq 0 \quad \forall i, \]
\[ s_i \geq s_{i-1} + \tau_i \quad \forall t, \text{ and } \forall t_j \in A_i \quad \text{(precedence constraints)} \]
\[ |s_i - s_{i-1}| \geq \tau_i \quad \forall t, \text{ and } \forall t_j \in T_i \quad \text{(resource constraints)} \]
\[ s_i \geq 0 \quad \text{and } \forall t, \text{ is integer} \]

Solving S1 is not necessarily going to provide the best overall schedule for the tasks composing a signal processing algorithm, because the algorithm is not executed once, but repetitively. The iterative nature of signal processing algorithms permits them to be pipelined such that repetitions overlap in time. Although the total time to perform one iteration of the algorithm is not reduced over the optimum found by solving S1 (in fact, it may be increased), pipelining increases the average repetition rate by making use of idle resources. Figure 7.10 illustrates how a small rescheduling of tasks can increase the throughput of a DSP system. If the data dependencies of the algorithm allow task $t_4$ to be executed at an earlier time, then the throughput can be doubled even though the processing time for each repetition is unchanged.

In order to increase the throughput via pipelining, the total time each processing resource spends executing its assigned tasks for each repetition should be minimized. The resource utilized for the greatest length of time will determine the rate at which the algorithm can be iterated, so it is this maximum time which must be minimized. This forms the basis for a new scheduling problem:

Problem S2: Minimize $z$

Such that:  
\[ z - y_{r_k} \geq 0 \quad \forall r_k \]
\[ y_{r_k} - (|s_{r_k} - s_{r_{k-1}} - \tau_{r_k}|) \geq 0 \quad \forall r_k \text{ and } \forall t, t_j \in T_{r_k} \]
\[ s_i \geq s_{i-1} + \tau_i \quad \forall t, \text{ and } \forall t_j \in A_i \]
\[ |s_i - s_{i-1}| \geq \tau_i \quad \forall r_k \text{ and } \forall t, t_j \in T_{r_k} \]
\[ s_i \geq 0 \quad \text{and integer} \quad \forall t, \]
Problem 52 assumes that, while the collection of processing elements as a whole may be simultaneously working on several repetitions of an algorithm, each individual element completes its assigned tasks for a repetition before beginning work on another. Thus, the system executes in an interleaved manner, but the system components do not. If an algorithm contains data dependencies which cause processing elements to be unused for large segments of time, then a schedule found using 52 may give far smaller throughputs than necessary. Figure 7.11 shows how interleaving the iterations within individual processing elements can give higher repetition rates. The data dependencies and the task partitioning lead to a single iteration execution time of 6 units. Repetition interleaving within processing elements, however, doubles the repetition rate.

Deriving schedules which have minimum single repetition execution times, minimum processing element execution times, or maximum repetition rates is usually an NP-complete
problem for all but the simplest types of SFGs. The next section will describe some simple heuristics which can provide good solutions without expending inordinate amounts of effort.

2.4.2. Heuristics for Solving Scheduling Problems

This section will examine several heuristics which can be used to schedule a set of dependent tasks upon a set of processors operating in parallel.

2.4.2.1. Hu's Algorithm

An algorithm by Hu [Hu61] can be used to find good schedules which minimize the computation time for a single repetition of an algorithm upon a set of \(|B|\) processing elements. The basic idea behind the procedure is to give priority to scheduling those activities which are on paths through the SFG with the longest computational delay. The description of
the algorithm is as follows:

**Step 1 - Initialize the maximum task starting times:**

For each terminal task $t_i$ (i.e. a task with no successors), set the maximum starting time equal to the duration of the task: $s_{t_i}^{\text{max}} = \tau_i$. Do not set the maximum starting time for non-terminal tasks.

**Step 2 - Set the maximum task starting times:**

Find a task $t_i$, all of whose successors have been assigned a maximum starting time. Then, set the maximum starting time for $t_i$ as follows:

$$s_{t_i}^{\text{max}} = \tau_i + \max \{ s_{t_j}^{\text{max}} | t_j \in S_i \}$$

Perform this step until all tasks have been assigned a maximum starting time.

**Step 3 - Schedule tasks for execution:**

Determine $C$, the set of all tasks which have no antecedents:

$$C = \{ t_i | A_{t_i} = \emptyset \}$$

If $|C| \leq |B|$, then schedule each of these tasks on a processing element, leaving excess processors idle if necessary. Otherwise, if $|C| > |B|$, then schedule the set of tasks which have the largest values for their maximum starting times, breaking ties at random.

**Step 4 - Remove completed tasks:**

If there are no more tasks to schedule, then terminate the algorithm. Otherwise, remove each of the tasks scheduled for execution in Step 3 from the antecedent and successor lists of the unscheduled tasks, and return to Step 3.

Figure 7.12 shows the result of applying Hu's algorithm with $|B|=3$ to an SFG composed of tasks with unit duration.
Hu's algorithm gives an optimal minimal length schedule for unit length tasks arranged as an assembly tree (i.e. no task has more than one immediate successor). However, the procedure also gives schedules which are very close to optimal for arbitrary precedence structures composed of tasks with varying durations.

While Hu's algorithm provides good schedules and does not require a large amount of computation, it suffers from several problems with regard to the specific scheduling problem we considered in the previous section:

- It will work only on SFGs which contain no directed cycles.
- It requires that any task be schedulable on any processor.

These problems prevent the direct use of Hu's algorithm for our problem, but the principles underlying the procedure are useful, as will be seen.
2.4.2.2. SPAR-1

The SPAR-1 heuristic program [Wies67] attempts to schedule tasks upon a limited set of resources (i.e. processors) based upon a time window assigned to each task. The window associated with a task \( t_i \) is bounded by two times:

- \( s_{i_{\text{min}}} \) - the minimum time at which the task could possibly begin execution.
- \( s_{i_{\text{max}}} \) - the maximum time task execution may begin without delaying the entire algorithm.

These times are assigned at the start of the algorithm through an analysis of the SFG which proceeds as follows:

**Step 1 - Initialize for calculating minimum starting times:**

Each task \( t_i \) with no antecedents has its minimum starting time \( s_{i_{\text{min}}} \) set to zero. The minimum starting times for the remaining tasks remain unset.

**Step 2 - Calculate minimum starting times:**

Find a task \( t_i \) whose antecedents have all had their minimum starting times set. Then, set the minimum starting time for \( t_i \) as follows:

\[
s_{i_{\text{min}}} = \max \{ s_{j_{\text{min}}} + \tau_j \mid \forall t_j \in A_i \}
\]

Perform this operation until the minimum starting time has been set for all tasks.

**Step 3 - Initialize for calculating maximum starting times:**

For each task \( t_i \) with no successors, set \( s_{i_{\text{max}}} = s_{i_{\text{min}}} \). The maximum starting times for the remaining tasks remain unset.

**Step 4 - Calculate maximum starting times:**

Set \( s_{i_{\text{max}}} \) for a task \( t_i \), whose successors have all had their maximum starting times set as follows:
\[
\mathbf{s}_{t_i}^\text{max} = \min \{ \mathbf{s}_{t_i}^\text{max} - \tau_j \mid \forall t_j \in S_{t_i} \}
\]

Once the time window is set for each task, the main scheduling portion of SPAR-1 begins its operation:

**Step 1 - Initialization:**

The set of currently executing tasks \( E \) and the set of completed tasks \( C \) are both initialized to \( \{ \emptyset \} \), and the current time \( t \) is set to zero. In addition, all the system elements are placed in the set of currently free resources \( R \).

**Step 2 - Finish tasks and free resources:**

Find all tasks \( t_i \in E \) which have completion times \( c_t \) equal to the current time \( t \). For each such task, do the following:

- Add the task to the set of completed tasks: \( C = C + \{ t_i \} \);
- Free the resources used by the task: \( R = R + R_{t_i} \);
- Remove the task from the active set: \( E = E - \{ t_i \} \).

**Step 3 - Create an ordered list of tasks ready to begin execution:**

Create a list \( L \) of all inactive and uncompleted tasks whose antecedent tasks are completed:

\[
L = \{ t_i \mid A_{t_i} \subseteq C \text{ and } t_i \notin E \text{ and } t_i \notin C \}
\]

For each \( t_i \in L \), calculate the float time \( f_{t_i} \) as follows:

\[
f_{t_i} = \mathbf{s}_{t_i}^\text{max} - \max \{ t \mid s_{t_i}^\text{min} \}
\]

Then, rearrange the tasks on \( L \) in order of increasing float times.

**Step 4 - Schedule tasks for execution:**

Remove the head task \( t_h \) from \( L \) (i.e., the task with the least float time). If the resources to execute this task are available (i.e. \( R_{t_i} \subseteq R \)), then do the following:
• Reserve the resources for the task: \( R = R - R_t \);

• Place the task in the active set: \( E = E + \{ t_h \} \);

• Set the completion time for the task: \( c_t = t + \tau_t \).

If the resources needed by \( t_h \) are not available, do not schedule the task for execution at this time. Perform this step until \( L \) is empty.

**Step 5 - Go to next time step:**

If tasks remain to be scheduled, then increment the current time \( (t = t + 1) \) and return to Step 2. Otherwise, terminate the algorithm.

As in Hu's algorithm, SPAR-1 first attempts to schedule tasks which, if they were delayed, might cause a delay in the completion of the entire job. It differs from Hu's procedure in that SPAR-1 also incorporates a backtracking phase which tries to improve the schedule via reallocation of resources. More importantly, SPAR-1 associates tasks with specific resources and does not assume that any task may be executed on any processing element.

2.4.3. The PaLS Scheduling Algorithm

The PaLS scheduling algorithm uses some basic ideas from both Hu's algorithm and SPAR-1, but also incorporates features specific to our particular scheduling problem.

Because both of these algorithms require that the set of tasks be expressed as a directed, acyclic graph (DAG), a transformation of the original SFG must be performed. Figure 7.13 gives an example of the transformation applied to a biquad filter section. Basically, each delay element node in the SFG is broken into a storage operation (\( z_{\text{store}} \)) and an associated updating operation (\( z_{\text{update}} \)). Precedence arcs (which are assigned a weight of zero since no real data is sent) are added to the graph to prevent the updating operation from taking place before the previously stored value is used. This transformation always produces a DAG
Figure 7.13: Transforming an SFG to a DAG
if the underlying flow graph is *computable* [Oppe75] (i.e. no zero-delay directed cycles exist in the SFG).

Once the DAG is created, the data transfer tasks are explicitly entered into it. Each unit-edge crossing between bins is replaced by a string of three interconnected nodes (Figure 7.14):

*The link_load node:*

This node represents the task of loading the data word into the transmitting IPC. Since loading an IPC link takes only a single cycle, the weight of the node (or duration of the task) is set to 1. The loading task is assigned to be executed by the DSP associated with the bin from which the edge originates.

*The link_delay node:*

The actual transfer of data between processors is symbolized by this node. The duration of the task equals the delay of the link, which is determined from the previously
assigned link bandwidth between the originating and terminating bins of the edge. The transfer task is executed by a link resource, there being one such resource for every pair of DSPs which must communicate.

**The link_unload node:**

Unloading the data at the receiving IPC is signified by this node. Unloading, like loading, requires but a single cycle, so the associated node is given a weight of 1. The DSP resource at which the edge terminates is assigned the unloading task.

An edge $e_i$ which has $w_e > 1$ (signifying that multiple pieces of data must be transmitted), is split into $w_e$ unit-arcs, and the procedure outlined above is then performed on each of these.

Once the SFG has been transformed into a DAG and the data transfer tasks have been added, scheduling of the tasks can begin. The PaLS scheduler ranks the currently executable tasks according to their maximum starting times, as is done in Hu's algorithm and SPAR-1. Unlike these algorithms, however, it does not move through time in one unit increments while scheduling tasks. Instead, for each system resource the scheduler maintains a *time window* over which a set of tasks assigned to that resource become executable. Several methods can then be used to select from the set the next task to be executed on the resource. After the task is scheduled, the beginning of the time window for the resource is moved to coincide with the completion time of the scheduled task. The detailed PaLS scheduling algorithm will now be explained:

**Step 1 - Calculation of minimum and maximum starting times:**

Determine $s_{t_i}^{min}$ and $s_{t_i}^{max}$ for each task $t_i$ as was done at the start of the SPAR-1 algorithm.

**Step 2 - Initialize for actual scheduling:**

For each resource $r_k$ (i.e. for each DSP and for each inter-DSP link), set the beginning
of the time window $W_r$ to zero. Also, initialize the set of completed tasks $C$ to $\{ \phi \}$.

**Step 3 - Schedule a task on a resource:**

Select a particular resource $r_k$ and determine the subset $E_{r_k}$ of tasks assigned to that resource which are ready to execute:

$$E_{r_k} = \{ t_i \mid t_i \in T_{r_k} \text{ and } A_{t_i} \subseteq C \}$$

If $E_{r_k} = \{ \phi \}$, then skip the remainder of Step 3. Otherwise, select a task $t_s$ to be scheduled for execution at time:

$$s_i = \max \{ W_{r_k}, s_{i_{\min}} \}$$

(The method of selection will be discussed later.) The selected task is added to the list of completed tasks:

$$C = C + \{ t_s \}$$

and the minimum starting times for the successors of $t_s$ are updated as follows:

$$s_{i_{\min}} = \max \{ s_{i_{\min}}, s_i + \tau_i \} \quad \forall t_i \in S_i$$

The starting time for the window is now set to be the completion time of the scheduled task:

$$W_{r_k} = s_i + \tau_i$$

**Step 4 - Iterate over resources:**

Perform Step 3 for all resources. Determining the order in which the resources are handled will be discussed later.

**Step 5 - Schedule all tasks:**

Return to Step 3 until all tasks have been entered into the completion set.

Unlike SPAR-1 which handles tasks requiring multiple resources, the PaLS scheduler assumes that each task needs only one resource in order to begin processing. This is a reasonable restriction since the DAG nodes represent processing chores that require the facilities
of only a single DSP or IPC link.

An additional feature added to the PaLS scheduler attempts to remove the wasted time caused by IPC link blockages. As was discussed previously, the IPC will suspend the operations of its associated DSP if the DSP attempts to load a transmitter link which is still in the process of sending previously entered data. While this feature eases the problems associated with synchronizing parallel processes, it is desirable to avoid the suspension of DSP activity so that other tasks may be scheduled into this "deadtime". To this end, whenever a DSP resource schedules a link-loading task, all other link-loading tasks for that particular link and DSP are removed from the list of schedulable activities. This is accomplished by entering the link-delay task associated with the just scheduled link-loading task into the antecedent sets of the other link-loading tasks. Only after the link-delay task has been scheduled will the remaining link-loading tasks become executable, thus preventing them from ever being scheduled at a time which will suspend the activity of the DSP. Figure 7.15 illustrates the technique just described.

One of several heuristics may be used to select a task from the set of tasks schedulable on a particular resource:

\( H1 - \text{Minimum potential starting time:} \)

This heuristic selects the task \( t_i \) with the lowest value of \( s_{i, \text{min}} \) as assigned in Step 1 of the PaLS scheduling algorithm. The rationale behind this heuristic is to give priority to executing those tasks which are potentially capable of being initiated very early in the SFG. Of course, Step 1 determines \( s_{i, \text{min}} \) based upon the assumption that any task may execute on any one of an infinite number of resources. Thus, the initial value of \( s_{i, \text{min}} \) may be quite different from the actual minimum starting time for \( t_i \), given the resource restrictions of our model.
(a) Schedule t4 on DSP1

(b) Add new precedence edges to block scheduling of t1 and t7

(c) Schedule t5 on LINK 1-2

(d) Remove precedence edges to make t1 and t7 schedulable

**Figure 7.15:** Eliminating Suspension of DSP Activities
**H2 - Maximum potential starting time:**

The maximum time at which a task can begin execution without delaying the termination of the entire algorithm is used by this heuristic when selecting tasks to schedule. Of the currently schedulable tasks, the task $t_i$ with the lowest value of $s_{i,\text{max}}$ is selected. This heuristic tries to prevent tasks from being scheduled later than their respective $s_{\text{max}}$ times, which, if achieved, would lead to an optimal schedule. Note that this heuristic is not the same as that of SPAR-1 or Hu's algorithm, since the task $t_i$ selected by the PaLS scheduler may have $s_{i,\text{max}}>W_i$. Thus, the task cannot be started until some time after the current leading edge of the time window of the resource. While this can waste some time, it can also prevent the scheduling of a task of long duration which causes a subsequent task to miss its deadline.

**H3, H4 - Minimum actual starting time:**

This heuristic selects the task which has the earliest time at which it can execute, based upon the actual starting times of its antecedents. Since several tasks may meet this criterion, they can be differentiated based upon their values for $s_{\text{min}}$ (H3) or $s_{\text{max}}$ (H4). If the $s_{\text{max}}$ times are used, then the PaLS scheduler exhibits a behavior like that of SPAR-1 or Hu's algorithm.

Since heuristics H1 and H2 select tasks based upon their minimum or maximum potential starting times instead of actual times, it is possible that some usable time will be wasted, as was mentioned above. Therefore, the PaLS scheduler can employ task selection heuristics H1p and H2p, which attempt to "pack" another of the currently schedulable tasks into the unused time period between the start of the resource time window and the start of the selected task. If this is accomplished, the originally selected task is returned to the set of schedulable tasks, since it is possible that the execution of the packed task will make it possible to schedule an even more desirable task during the next iteration through the set of
resources.

The tasks selected for execution by a resource affect the set of schedulable tasks for other resources via the precedence constraints of the DAG. Thus, changing the order in which resources are handled can improve the final task schedule. Some heuristics used to affect the order might be:

- Random ordering;
- Give priority to resources with the greatest number of schedulable tasks;
- Give priority to the resource with the earliest schedulable task.

These techniques are not used at present, and the PaLS scheduler iterates through the DSP and IPC link resources in a deterministic sequence.

2.4.4. A Comparison of the Scheduling Heuristics

The various heuristics used in the PaLS scheduler were compared in the following manner. A sample space was created by generating 25 random graphs with 50 nodes having random weights from 1 to 10, and 200 edges from which deletions were made until all the directed cycles were broken. The resulting DAGs were then randomly partitioned over 5 bins and optimum link widths were assigned. A lower bound on the completion time for a single execution of the entire DAG, $c_{LB}$, was determined (including communication costs) assuming that infinite resources were available. While this lower bound might not be achievable given the actual resource constraints, there was no computationally tractable method of finding the true optimum duration of each DAG. Heuristics can then be compared by forming the ratio of the actual execution time ($c_{actual}$) found by each scheduling heuristic with $c_{LB}$.

When scheduling a set of tasks for a single execution, the heuristics which used $s_{max}$ to select amongst the schedulable tasks (H2, H2p, and H4) performed much better than those which used $s_{min}$ (within <5% of the lower bound versus >20%) (Table 7.2). In this case,
paying attention to those activities which are close to missing their deadlines is superior to scheduling the earliest available tasks.

The previous comparison assumed that the IPC links could not be loaded until any previously entered data was transferred. If queues were placed on each link transmitter, then link loading tasks could be scheduled any time the queue had room, with no concern about suspending the activity of the DSP. It would seem that this additional freedom would lead to shorter schedules, but this does not occur (Table 7.3). The irregular nature of the randomly generated DAGs makes the occurrence of successive, closely spaced data transfers unlikely, thus reducing the effectiveness of the hardware queues. Queues would probably be more valuable when scheduling very ordered flow graphs (a common case in digital signal process-

Table 7.2: A Comparison of the PaLS Scheduling Heuristics

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>$C_{actual}/C_{LB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>1.32±0.07</td>
</tr>
<tr>
<td>H1p</td>
<td>1.23±0.05</td>
</tr>
<tr>
<td>H2</td>
<td>1.21±0.05</td>
</tr>
<tr>
<td>H2p</td>
<td>1.05±0.03</td>
</tr>
<tr>
<td>H3</td>
<td>1.03±0.03</td>
</tr>
</tbody>
</table>

Table 7.3: Scheduling in the Presence of Hardware Queues

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Link Queue Size</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td></td>
<td>1.29±0.07</td>
<td>1.23±0.06</td>
<td>1.23±0.06</td>
<td>1.23±0.06</td>
<td>1.23±0.06</td>
</tr>
<tr>
<td>H1p</td>
<td></td>
<td>1.21±0.03</td>
<td>1.20±0.04</td>
<td>1.20±0.04</td>
<td>1.20±0.04</td>
<td>1.20±0.04</td>
</tr>
<tr>
<td>H2</td>
<td></td>
<td>1.19±0.03</td>
<td>1.18±0.03</td>
<td>1.18±0.03</td>
<td>1.18±0.03</td>
<td>1.18±0.03</td>
</tr>
<tr>
<td>H2p</td>
<td></td>
<td>1.03±0.04</td>
<td>1.02±0.02</td>
<td>1.02±0.02</td>
<td>1.02±0.02</td>
<td>1.02±0.02</td>
</tr>
<tr>
<td>H4</td>
<td></td>
<td>1.03±0.02</td>
<td>1.03±0.03</td>
<td>1.03±0.03</td>
<td>1.03±0.03</td>
<td>1.03±0.03</td>
</tr>
</tbody>
</table>
ing), where they would aid in creating dense, regular code for each DSP while helping to avoid the suspension of DSP operations.

In order to gauge the effects of pipelining and interleaving upon the scheduling, each of the DAGs in the sample set was "unwound" by connecting several repetitions of each DAG together while preserving data dependencies. Schedules were created for the new DAG using each heuristic (Table 7.4). This brute-force approach demonstrated that interleaving and pipelining decreased the time between repetition initiations to as little as 45% of that required by a single iteration, increasing throughput by over a factor of two. Much of the performance gain was apparent after only 3-4 unwindings. Interestingly, some heuristics which produced poor schedules for the single repetition case also produced the best schedules when multiple iterations were scheduled (H1p and H3).

Adding hardware queues capable of holding four data words did not greatly improve the schedules obtained with unwinding (Table 7.5). Only heuristic H2 showed significant improvement, and it performed so poorly to begin with that it still could not match the results of the remaining heuristics.

Since scheduling is a difficult combinatorial problem, the question might arise as to why simulated annealing is not included amongst the PaLS scheduling heuristics. One reason is

| Table 7.4: Scheduling with Pipelining and Interleaving of Several Repetitions |
|---|---|---|---|---|---|
| Heuristic | 1 | 2 | 3 | 4 | 5 | 6 |
| H1 | 1.22±0.08 | 0.83±0.09 | 0.71±0.07 | 0.64±0.06 | 0.59±0.06 | 0.57±0.07 |
| H1p | 1.17±0.05 | 0.76±0.07 | 0.61±0.06 | 0.51±0.06 | 0.47±0.04 | 0.46±0.05 |
| H3 | 1.16±0.06 | 0.74±0.05 | 0.58±0.05 | 0.51±0.05 | 0.47±0.05 | 0.47±0.06 |
| H2 | 1.06±0.05 | 1.05±0.06 | 1.06±0.07 | 1.07±0.08 | 1.08±0.08 | 1.08±0.08 |
| H2p | 1.03±0.03 | 0.68±0.07 | 0.62±0.09 | 0.56±0.09 | 0.55±0.08 | 0.52±0.08 |
| H4 | 1.03±0.03 | 0.67±0.07 | 0.58±0.08 | 0.55±0.07 | 0.53±0.08 | 0.52±0.07 |
Table 7.5: Scheduling of Unwound Repetitions in the Presence of Hardware Queues

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>1.22±0.08</td>
<td>0.80±0.08</td>
<td>0.65±0.07</td>
<td>0.58±0.08</td>
<td>0.54±0.08</td>
<td>0.51±0.07</td>
</tr>
<tr>
<td>H1p</td>
<td>1.19±0.06</td>
<td>0.75±0.06</td>
<td>0.59±0.06</td>
<td>0.54±0.07</td>
<td>0.50±0.07</td>
<td>0.47±0.07</td>
</tr>
<tr>
<td>H3</td>
<td>1.17±0.06</td>
<td>0.75±0.06</td>
<td>0.60±0.06</td>
<td>0.53±0.07</td>
<td>0.49±0.07</td>
<td>0.46±0.07</td>
</tr>
<tr>
<td>H2</td>
<td>1.04±0.04</td>
<td>0.97±0.03</td>
<td>0.94±0.03</td>
<td>0.94±0.04</td>
<td>0.93±0.05</td>
<td>0.93±0.05</td>
</tr>
<tr>
<td>H2p</td>
<td>1.03±0.03</td>
<td>0.75±0.07</td>
<td>0.66±0.09</td>
<td>0.63±0.12</td>
<td>0.60±0.12</td>
<td>0.60±0.13</td>
</tr>
<tr>
<td>H4</td>
<td>1.03±0.03</td>
<td>0.69±0.06</td>
<td>0.61±0.08</td>
<td>0.57±0.08</td>
<td>0.56±0.08</td>
<td>0.57±0.09</td>
</tr>
</tbody>
</table>

that the simple heuristics presented are already capable of producing schedules with runtimes within 5% of the theoretical lower bound. While a simulated annealing technique might better this, the large runtimes required are usually not worth the possible improvement in the schedule (Table 7.6). The poor time performance of simulated annealing in scheduling problems can be attributed to the following factors:

- Each task must be randomly moved many times within the schedule to reach thermal equilibrium.
- Moving a task requires a search to find a free time interval of sufficient size to hold the task.
- Precedence relations must be checked when moving a task to prevent violations.

Table 7.6: Execution Times for Various Scheduling Heuristics

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Computation Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>36.9</td>
</tr>
<tr>
<td>H1p</td>
<td>36.2</td>
</tr>
<tr>
<td>H3</td>
<td>36.1</td>
</tr>
<tr>
<td>H2p</td>
<td>29.3</td>
</tr>
<tr>
<td>H2</td>
<td>28.1</td>
</tr>
<tr>
<td>H4</td>
<td>29.4</td>
</tr>
<tr>
<td>Annealing</td>
<td>&gt;3600</td>
</tr>
</tbody>
</table>
The searching and checking required to generate a valid perturbation of a schedule requires much more time and effort than an equivalent operation during partitioning. Therefore, the simulated annealing technique was discarded for this phase of PaLS.

2.5. Code Generation

Once the scheduling of the partitioned tasks has been completed, the actual code for each task must be generated for the DSP elements. For SFGs in which the nodes represent simple operations such as adding or multiplying, no true code generation is necessary. More complex algorithms, where a single node may stand for an involved calculation (such as searching an array), are required to be hand-coded into simpler operations in the present PaLS system.

The output from the PaLS scheduling phase lists the order of execution for each activity on a particular processing element. The simple tasks are hand-converted into an algebraic form of register transfer language (RTL) and are fed, in order, into an assembler which creates executable microcode.

The DSP microcode assembler is very similar to the one used in the development of the HECTOR microprocessor [Vand86]. A file describing the DSP datapath is used to transform the RTL source statement into a bit string representing a microinstruction. The translator searches for a match between a component of the source statement and a set of expressions contained in the description file. If a match is detected, the component is removed and replaced with the lower-level expression paired with the matching expression in the description file. Simultaneously, the bits which control the datapath resources used by the transformed operation are set to their proper values in the microcode string. These actions are carried out recursively upon the transformed source statement and bit string until one of two things happens:
• A conflict is encountered in setting bits within the microinstruction string.

• The RTL source statement is reduced to nothing.

The first case occurs when the translator has specified interfering uses of a datapath resource, such as trying to transfer the contents of two registers over a single bus. If this happens, the translator backtracks and tries a different transformation in an attempt to clear the conflicts. If a correct transformation cannot be found (and one will be found if it exists), the user is informed of the error so that a correction can be made. In the second case, a source statement has been successfully converted into a microinstruction bit string. The collection of bit strings can then be loaded into the Axe description of the DSP network to drive the simulation.

3. Problems with PaLS

A number of problems exist with the PaLS system just described, all of which originate within the initial partitioning phase. Because the output of the partitioning phase is used by both the link assignment and scheduling portions of PaLS, errors in the partitioning of an SFG can skew the entire parallelization process. The problems which will be examined in this section are:

• Inability to correctly set the bin balancing coefficient $\alpha_{bal}$.

• Inability to account for the temporal nature of flow graphs.

• Inability to generate regular partitionings of flow graphs.

3.1. Setting the Balancing Coefficient

Once a system designer has entered a SFG for a particular signal processing algorithm and selected the number of processors, the value of $\alpha_{bal}$ must be set in the partitioning objective function:
The balancing coefficient affects the overall partitioning of the SFG by expressing the relative cost of communications compared to computations. When speaking of systems where the main concern is throughput, a suitable value for $\alpha_{bal}$ is the ratio of the times taken to perform a computation versus a communication:

$$\alpha_{bal} = \frac{\tau_{\text{computation}}}{\tau_{\text{communication}}}$$

For example, setting $\alpha_{bal} = 0.5$ states that interprocessor communications take twice the time that a computation does, so they are half as desirable. Thus, a change in the partitioning which increased the bin imbalance by $n$ units would be looked upon favorably if it also resulted in a decrease of at least $0.5n$ interprocessor data transfers. Conversely, setting $\alpha_{bal} = 2.0$ would cause the PaLS partitioner to emphasize the maintenance of bin balance, even if extra communications were required as a result.

The first problem with setting the $\alpha_{bal}$ coefficient is the non-intuitive nature of the act. The system designer has a clear objective: find a system configuration capable of achieving maximum throughput. This objective does not directly translate into a parameter expressing the cost of communications versus actual calculation.

Given that the designer does set a value for $\alpha_{bal}$, the value itself will affect the final partitioning. Setting $\alpha_{bal}$ to a high value will create a partitioning in which all processors have the same number of computations to perform. However, the extra communications needed to create such an exact balance might throttle the total system throughput. On the other hand, a low value for $\alpha_{bal}$ leads to a system partitioning with very little interprocessor communication, but possibly very imbalanced processor loads. The overloading of any processor in the system will lead to a computational bottleneck which also reduces total throughput.

In a fixed network, there is the possibility of being able to derive a value for $\alpha_{bal}$ from the communication delays between processors. This is not possible here, since the network
topology and delays are not fixed until after the PaLS link assignment phase. The end result is that the designer must try several values for the balancing coefficient, and must then select the best of the generated solutions. In effect, the original search for a system with high throughput has been replaced with the search for the best value of $\alpha_{\text{bal}}$.

Unfortunately, the search for a single, suitable value of $\alpha_{\text{bal}}$ for a multiprocessor system may be fruitless. For example, movement of computations between two processors connected via a high-speed link would be controlled using an objective function with a high value for $\alpha_{\text{bal}}$. But, two processors in the same network which are connected with a slow, narrow link, would require a small value for $\alpha_{\text{bal}}$. Obviously, if the optimal network for a given signal processing algorithm contains IPC links with different widths, then a single value for the balancing coefficient will not adequately describe the relative cost of communication and computation for all pairs of processors.

The above discussion casts strong doubts on the suitability of an objective function which uses a single parameter to weigh communication costs against computational load balancing.

3.2. The Temporal Nature of Signal Flow Graphs

Aside from the difficulty in setting $\alpha_{\text{bal}}$, the partitioning objective function includes no knowledge of the time sequencing involved in the execution of the tasks composing a SFG. Minimization of communications and load balancing are the only factors affecting the partitioning of the flow graph.

Figure 7.16 illustrates the problems which can arise by ignoring the temporal nature of a SFG. Judging by the objective function, Figure 7.16 shows an optimal bipartition which exactly bisects the computational tasks and requires only one data transfer. From a global point of view the two processors form a pipeline, and the second DSP may not begin work until the first finishes and transfers the results of its calculations. Therefore, the minimum
time required to perform a single execution of the entire job is $6 \times 100 + 6 \times 100 = 1200$ time units (ignoring communication delay, which is minor compared to the task durations).

A poorer partitioning of the flow graph, at least according to the objective function, is shown in Figure 7.17. Note that now the computational load is unevenly shared, and two data transfers are needed. Despite its poor rating, the total execution time has been reduced to $7 \times 100 + 2 \times 100 + 100 = 1000$ time units, since the tasks $(A, B)$ and $(C, D)$ can now be executed in parallel.

If repetitious, instead of single, executions of an algorithm are considered, the results of the previous situation are reversed. Pipelining can be employed in the case of iterative execu-

![Figure 7.16: An Exact Bipartitioning](image)

![Figure 7.17: An Imbalanced Bipartitioning](image)
tion, which makes exact load balance important because the slowest system element deter-
nines the throughput of the pipe. Now, the partitioning of Figure 7.16 is more desirable
since a new repetition may be initiated every 600 time units, whereas a system built according
to Figure 7.17 would be limited to a new iteration once every 900 time units.

From the above discussion, it is obvious that, in addition to communications and load
balancing, scheduling of the sub-tasks must also be taken into consideration when partition-
ing an algorithm.

3.3. Partitioning Regular Flow Graphs

Up to this point, the partitioner of the PaLS system has only been applied to randomly
generated flow graphs. As was mentioned previously, most signal processing algorithms
have very regular, orderly structures when expressed in the form of a SFG. By way of exam-
ple, Figure 7.18 illustrates an infinite impulse response (IIR) filter composed of three parallel
banks of four cascaded biquad sections.

Figure 7.18: Parallel-Cascade IIR Filter
An optimal partitioning of the previous filter structure onto a three processor system is shown in Figure 7.19. The optimal solution is as well balanced as possible, and requires only four interprocessor data transfers. The partitioning phase of PaLS, however, generates a partitioning which is well balanced but globally irregular (Figure 7.20). This irregularity increases the number of interprocessor data transfers slightly over what was needed in the optimal solution.

The objective function has no notion of regularity, and quantifying such a solution quality is not trivial. But, the most regular partitionings usually require the fewest communications, and so the partitioning phase should generate ordered solutions by default. However, heuristic algorithms, such as the Kernighan-Lin method, speed-up the search for a good partitioning and thus cannot guarantee that the optimal solution will always be found (or will ever be found!). The NP-complete characteristic of the partitioning problem allows such a guarantee only if the user is willing to wait an exponentially increasing amount of time for optimal

Figure 7.19: Optimal Partitioning of the Parallel-Cascade Filter
partitionings of increasingly larger flow graphs. Still, it is tempting to imagine that the simulated annealing technique will always arrive at the optimal partitioning if the temperature is decreased slowly enough. After all, performing simulated annealing upon a flow graph bears a remarkable resemblance to the physical process by which highly ordered crystals are formed from molten materials. Such crystals would correspond to the regular bin contents depicted in Figure 7.19. But, as is shown in Appendix C, the annealing temperature must be decreased exponentially slowly in order to insure the arrival at such an optimal solution. This is necessitated by the large number of distinct SFG partitionings which have nearly the same objective function values while possessing markedly different structures. Thinking back to the physical example and equating the value of the objective function to the energy of a system, it can be said that the optimal and near-optimal partitionings correspond to degenerate states.
The irregularities in the partitionings generated by the PaLS system destroy the regularity of the final system configuration. This does not usually result in a significant increase in the execution time for an algorithm, since only a small number of extra communication tasks are typically added over the optimum number. What is damaged is the code regularity, so that large amounts of in-line code must be used instead of simple loop constructs. For this reason, it would be beneficial if a method could be found whereby the algorithm regularity is retained in the final code of each DSP.

4. Alterations to PaLS

Several alternative solutions to the problems plaguing the PaLS programming system will now be examined:

- Completely integrating the partitioning, linking, and scheduling phases;
- Giving the PaLS scheduler the ability to alter the partitioning;
- Incorporating temporal information into the initial partitioning phase.

4.1. Alternative #1: Complete Integration

If the linking and scheduling phases were integrated into the partitioning phase as in the first alternative, then the Kernighan-Lin and simulated annealing methods would need to evaluate each potential node movement between bins as follows:

- Determine the link widths for the system before the node movement;
- Determine the schedule for the system before the node movement;
- Determine the new link widths for the system after the node movement;
- Determine the new schedule resulting from the node movement and the changes in link widths;
- Determine the change in throughput between the new and original schedules.
The change in throughput would be used to evaluate the associated node movement and drive the partitioner to converge to a solution. Naturally, evaluating each node movement as described above will require a substantial amount of computation. The total effort needed to partition a reasonably sized flow graph quickly becomes excessive, since the Kernighan-Lin and annealing methods both consider many possible node movements.

4.2. Alternative #2: Re-Partitioning

Despite its impracticality, the integrated partitioning scheme has a powerful advantage: a flow graph is divided based upon the actual quantity of interest - throughput - instead of a less accurate measure derived from load balance and the gross number of communications. The basis of the second alternative for improving the PaLS system exploits the fact that the scheduling phase is concerned with the temporal behavior of the algorithm, and therefore has an estimate of the total system throughput. Possessing such an estimate, the scheduler might be able to correct some errors in the initial partitioning and link assignment which will increase the throughput. In order to make these adjustments, facilities would have to be added to the scheduling algorithm for moving tasks between processing resources and adjusting the width of the IPC links. These new abilities allow the scheduler to distribute tasks among the processors to remove excess idle time, and to accurately size link widths based upon burst communication rates rather than long-term average rates.

Adding the error-correcting capabilities to the scheduler suffers from several drawbacks. First of all, the movement of a task from one processing element to another would likely require additional data transfers. Idle time must not only exist on the destination processor so that it may execute the new task, but there must also be idle time on the appropriate links to handle the required communications. The additional data transfers may also require the use of links not currently in existence. A new link assignment would then have to be carried out, and this could cause other links to shrink in width. This shrinkage increases the transfer
delay, which can cause a violation of a dependency constraint or overloading of a link resource. The net result would be the need to compute an entirely new schedule whenever such complications occur (Figure 7.21). Since such complications would be quite common, the computing effort for this alternative would rival that found for the integrated partitioning method.

The need for such massive computing efforts could be eliminated by disallowing node movements that cause complex, interacting conflicts. Restricting the set of node movements to those which create simple, easily removed conflicts would greatly reduce the number of possible perturbations to the system and make the resulting schedules easier to derive. The restriction of the immediate search space could even be viewed in a positive light were it not for the following fact: the initial result output by the PaLS partitioner can be quite different from the optimal solution (compare Figure 7.19 and Figure 7.20). The small number of immediately feasible moves could increase the amount of time taken by the scheduler to find the optimal configuration, since the allowable sequence of node movements might follow a convoluted path through the space of solutions. Worse yet, the node movements necessary to create the optimal configuration might not be allowed due to their complexity. Add to these drawbacks the difficulty of selecting the simple node movements from the large set of possible moves, and this method of fixing PaLS becomes unworkable.

4.3. Alternative #3: Use of Temporal Information During Partitioning

The difficulties with the last alternative stemmed from its underlying principle of operation: fixing the errors in the initial partitioning and link assignment while maintaining a feasible schedule with respect to the precedence constraints of the algorithm. A correction method based upon this principle is defeated by the complexity of dealing with a large number of conflicting, interacting constraints coupled with a large deviation of the initial partitioning from the optimal solution. However, instead of trying to correct the errors
Before Node Movement -

<table>
<thead>
<tr>
<th>DSP1</th>
<th>DSP2</th>
<th>DSP3</th>
<th>LINK 1-3</th>
<th>LINK 2-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>ll(bd)</td>
<td>a</td>
<td>ll(ad)</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>ll(cd)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lu(bd)</td>
<td>lu(cd)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lu(ad)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ld(bd)</td>
<td>ld(ad)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ld(cd)</td>
<td></td>
</tr>
</tbody>
</table>

Schedule Before Node Movement -

DSP1: | DSP2: | DSP3: | LINK 1-3: | LINK 2-1: |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>ll(ba)</td>
<td>ll(bd)</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>ll(ba)</td>
<td>ll(bd)</td>
<td>c</td>
<td>ll(cd)</td>
<td></td>
</tr>
<tr>
<td>ll(ba)</td>
<td>ll(bd)</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l(ba)</td>
<td>ll(ba)</td>
<td>c</td>
<td>ll(cd)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ll(ba)</td>
<td>ll(ba)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ll(ba)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ll(ba)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ll(ba)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schedule After Node Movement -

Figure 7.21: The Effects of Re-Partitioning During Scheduling
introduced by the preceding phases of PaLS, better results will certainly be found with less effort by creating a good initial partitioning in the first place. The heart of the third alternative for improving the PaLS system relies upon supplying some crude scheduling and linking information to guide the initial partitioning phase.

By adding some information on the timing behavior of the tasks and the interprocessor communication delays, the partitioner can take actions to directly improve the system throughput without being concerned with a great number of active constraints. In order to include the temporal information, changes are made in the signal flow graph topology and in the partitioning algorithms.

4.3.1. Changes in the Signal Flow Graph Topology

As was previously illustrated, the standard Kernighan-Lin and simulated annealing algorithms do not account for timing and regularity factors in the partitionings they produce. By adding edges to the original flow graph which reflect these factors, the standard partitioning algorithms can be used to obtain better results. It is important to note that these phantom edges do not represent the flow of data, and so do not effect the assignment of IPC link widths or create new precedence constraints during scheduling.

4.3.1.1. Scheduling Edges

The PaLS scheduling heuristics prioritize the scheduling of tasks based upon their minimum and maximum starting times in the absence of resource constraints. These \( s^{\text{min}} \) and \( s^{\text{max}} \) times can also be used to create edges between nodes which reflect the desirability of assigning them to the same processing element.

Figure 7.22 shows a flow graph with \( s^{\text{min}} \) and \( s^{\text{max}} \) assigned for each unit-duration task in the algorithm. (Note that no communication tasks are included here, since no partitioning has been performed yet.) Arranging the tasks on a time-line, it can be seen that certain tasks
can be executed in parallel ((A,B), (B,C,D), (D,E)). Placing enough of these tasks in the same processor can remove enough parallelism to slow the entire algorithm.

By adding edges of the appropriate weight between nodes whose tasks overlap in time, it is possible to make the partitioning algorithms avoid configurations with low levels of parallelism and poor schedules. These scheduling edges have negative weights as opposed to the positively weighted edges seen so far. Whereas positive real edges crossing between bins are to be avoided since they correspond to time consuming data transfers, negative scheduling edges between bins are favorable because they signal that potentially concurrent tasks are being placed in separate processors.

![Representation of Parallelism in an SFG](image)

Figure 7.22: Representation of Parallelism in an SFG
The effect of scheduling edges can be seen in Figure 7.23. A good bipartitioning of the flowgraph which has a poor schedule is shown in (a). Only two crossing edges are needed and the computational load is nearly balanced, but the second processor cannot begin to execute until some results are passed from the first DSP. This lengthens the time needed for a single execution of the algorithm.

Once scheduling edges (dashed lines) are added in (b), the rating for partitioning (a) is lowered because only one scheduling edge crosses between the bins, indicating a low level of parallelism. In the presence of scheduling edges, (c) is a better partitioning since four negatively weighted edges now cross between the bins while the number of data transfers remains the same. A good load balance is also achieved, but more importantly, the parallelism is increased since tasks (A,C,E) can execute in parallel with tasks (B,D).

The example shows that scheduling edges act like repelling forces between the connected nodes. This repelling force acts to distribute the tasks between the processing elements, just as the original load balancing term in the objective function does. However, the load balancing repulsion is *amorphous* - its action is based upon the bulk loading of each partition rather than the temporal properties of individual SFG tasks. The scheduling edges permit a finer distinction to be made in determining which nodes should be moved to increase the parallelism and throughput of the system.

The calculation of the scheduling edge weight between two nodes $n_1$ and $n_2$ can be done by estimating the time overlap between their tasks $t_1$ and $t_2$, respectively (Figure 7.24). Because the exact schedule is not fixed at this point in time, a probabilistic time overlap must be computed. Assume each task $t_i$ has a duration $\tau_i$ and must begin within a specific time window $([s_{t_i}^{\min}, s_{t_i}^{\max}])$ if the total algorithm is to terminate at the minimum possible time. If the time $s_i$ at which $t_i$ begins execution within the time window is a random variable with a probability density function $(\pi_i(s_i))$, then the weight assigned to the scheduling edge between
Figure 7.23: Partitionings with and without Scheduling Edges

(a) Good Partitioning Without Scheduling Edges

(b) Poor Partitioning Once Scheduling Edges Are Added

(c) Good Partitioning In The Presence Of Scheduling Edges
and $n_2$ is calculated as follows:

$$w_{s_1}^{n_1} = - \sum_{s_1 = S_1^\text{min}}^{s_1} \pi_1(s_1) \cdot \sum_{s_2 = S_2^\text{min}}^{s_2} \pi_2(s_2) \cdot \max \left\{ 0, \min \{ s_1 + \tau_{i_1} - s_2, s_2 + \tau_{i_2} - s_1 \} \right\}$$

Naturally, the choice of the probability density function for the task starting time has a major effect upon the weight of the scheduling edges between graph nodes. Since the scheduling algorithms used in the PaLS system give the highest priority to scheduling tasks which are closest to missing their deadlines, the majority of tasks begin at or near their respective $s_{\text{max}}$ time. Therefore, a density function which is heavily weighted toward the end of the time window is usually selected (Figure 7.25).

4.3.1.2. Regularity Edges

Regularity edges, like scheduling edges, are added to aid the PaLS partitioning algorithm in determining some quality of the configuration which is not immediately apparent using the original SFG. In this case, the regularity edges attempt to prevent the formation of fragmented partitions such as those in Figure 7.20.

Figure 7.20 shows that while the partitioning algorithms detect short-range order within the SFG (e.g. the individual biquad sections are not split in the partitioning of the parallel-
cascade filter), the long-range order of non-primitive components is missed. This can be attributed to the fact that the nodes composing a biquad section are densely connected, but the collection of biquad sections is loosely tied together using comparatively few edges. If the attractive force of these edges can be increased, it is more likely that the regular configuration of Figure 7.19 will be generated.

The attractive forces can be increased by adding edges between a node \( n_i \) and any successor nodes which are dependent upon results generated by \( n_i \). The algorithm proceeds as follows:

**Step 1 - Initialize:**

Mark every node which has no successors or is a delay (1/z) node. Unmark the remaining nodes.

**Step 2 - Find expansion node:**

Find an unmarked node \( n_i \) whose successor nodes have all been marked.

**Step 3 - Add regularity edges:**

Select a successor node \( s_{n_i} \) of \( n_i \) and let \( w_{c_i} \) be the weight of the connecting edge.
Determine the total weight of the real incoming edges to $s_n$:

$$w_{\text{tot}} = \sum_{i} w_{e_i} \cdot [ \delta(v_-(e_i), s_n) ]$$

Now, for each edge $e_k$ originating from $s_n$, add a regularity edge $n_i \rightarrow v_-(e_k)$ with a weight of:

$$w_{\text{re}} = \frac{w_{e_k}}{w_{\text{tot}}} \cdot w_0$$

**Step 4 - Iterate over successor nodes:**

Perform Step 3 until all the successor nodes have been processed. Then mark node $n_i$.

**Step 5 - Iterate over unmarked nodes:**

Return to Step 2 until all nodes are marked.

The philosophy behind the regularity edge weight calculation is that the weight should be proportional to the fraction of the input data to the terminating node which originates at the source node. For example, in Figure 7.26 node A provides $4/(4+6) = 40\%$ of the input data to node C, so the outputs from C are $40\%$ dependent on the output from A. Therefore, regularity edges are drawn from A to nodes D, E, and F which are $40\%$ as strong as those from C.

The effect of adding regularity edges to the previously seen parallel-cascade filter is depicted in Figure 7.27. Whereas the components were loosely connected initially, the three parallel groups of cascaded sections become enmeshed in a net once the regularity edges are added. It then becomes much more likely that the regularity will be preserved in the output from the partitioning phase of PaLS.

Several points should be noted about the regularity edge creation algorithm. First of all, regularity edges are not propagated beyond delay ($1/z$) nodes. This prevents infinite cycles from occurring and maintains the computability of the flow graph, but also represents the fact
Figure 7.26: Creation of Regularity Edges
that operations separated by delay operators execute during different iterations. Therefore, these node operations should not become more tightly connected. Also, the edge weight...
calculations are carried out with fixed point arithmetic. The resulting fraction truncation reduces the regularity edge weights between well separated nodes to zero, thus preventing the proliferation of many edges with very small weights.

4.3.1.3. Critical-Path Strengthening

A flow graph can also be augmented with critical edges which are a combination of scheduling and regularity edges. These edges are added to strengthen connectivity along the critical path of the SFG. The critical path is composed of a set of tasks $T_{CP}$ whose minimum and maximum starting times are equal:

$$T_{CP} = \{ t_i | s_{t_i}^{\text{min}} = s_{t_i}^{\text{max}} \}$$

From the definition, if any task in $T_{CP}$ is delayed, then the time required to complete a single execution of the algorithm will increase by the same amount.

Obviously, if any two connected nodes associated with tasks in $T_{CP}$ are placed in different bins, then a data transfer delay will be incurred which will cause at least one of the tasks to miss its execution deadline. Since this is undesirable, a critical edge is added between the two critical nodes with a weight of $S_{CP} \cdot w_{c}$, where $w_{c}$ is the weight of the original connecting edge and $S_{CP}$ is the strengthening factor. The attractive force of the added critical edges tends to pull the critical tasks into a single bin, thus eliminating communication delays where they create the most damage.

In most cases, load balancing and minimization of total communication delay prevents all critical tasks from occupying the same bin. However, another benefit of the critical edges is their ability to act as a strongly-connected skeletal framework around which the rest of the partitioning can congeal. The framework thus serves the same function as a seed crystal does when drawing an ingot from a vat of molten material, in that it provides a template which can be filled in to produce the final configuration. An example of this can be seen by highlighting the critical paths of the parallel-cascade filter (Figure 7.28). A counter example,
however, involves the inner product tree (Figure 7.29) where every task is on a critical path. Thus, every path is strengthened and no net gain is made in determining a good partition.

Figure 7.29: Critical Paths in an Inner Product Tree
One of the main difficulties with the critical path strengthening technique is the determination of an appropriate value of $S$. As of this time, the user must set $S$ by trial-and-error, thus creating the same problems encountered when setting $\alpha_{bal}$.

4.3.2. Changes in the PaLS Partitioning Algorithms

Alterations must be made in the PaLS partitioning algorithms in order to make efficient use of the augmented flow graph and to bring the algorithm model closer to reality. These changes concern the form of the objective function and the way it is computed, as well as the techniques used to guide the partitioning process.

4.3.2.1. The New Objective Function

The objective function used with the augmented flow graph is:

$$H' = H'_{comm} + H'_{para}$$

$H'_{comm} = \text{Total time used for data transfers}$

$H'_{para} = \text{Total time lost due to parallelism conflicts}$

The penalty for interprocessor communications, which used to be expressed merely as the number of edges crossing between bins ($H_{cross}$), is now given by the sum of all the delays for data transfers ($H'_{comm}$). Counterbalancing $H'_{comm}$ is $H'_{para}$, the total time lost by placing potentially concurrent tasks on the same processor, which is akin to the load balancing term $H_{bal}$. The methods used to calculate these two terms will now be examined.

4.3.2.1.1. Calculating $H'_{comm}$

A first cut at an expression for $H'_{comm}$ might be:

$$H'_{comm} = \sum_{E \in \mathcal{E}} w_{r_e} \cdot \tau_{r_e} \tag{7.8}$$

The summation in Eqn. (7.8) is taken over only the real edges, since phantom edges do not correspond to the actual flow of data. The number of data values being transferred ($w_{r_e}$) is multiplied by the delay of the link to which the edge is assigned ($\tau_{r_e}$).
Eqn. (7.8) can only be applied if the link delays are known. Edges which have terminating nodes in the same bin can use a delay of zero, but the remaining link delays are determined by the link widths. Therefore, it is necessary to incorporate the link assignment phase into the partitioning algorithms. The Kernighan-Lin and simulated annealing algorithms require the generation of a new link assignment for every node movement considered in order to assess the change in $H'_{comm}$. This is not as great a burden as it seems, since the link assignment algorithm converges very rapidly when the initial assignment is close to the optimum. Because the movement of one or two nodes does not usually cause a radical change in the optimal link assignment, the increase in computing effort is modest. The advantage of doing the link assignment during the partitioning phase is the greater accuracy obtained by knowing the link delays. As was mentioned previously, this accuracy is usually not obtainable using a single parameter such as $\alpha_{bil}$.

Correctly computing a realistic value with Eqn. (7.8) requires a solution to a more subtle problem. Multiple real edges originating from a simple operation node (e.g. a multiplier) carry identical output data values. If two or more of these edges extend to another bin, the data does not have to be sent repeatedly, but only once. The receiving processor can temporarily store the single transmitted value for multiple uses and save the time which would be wasted on the extra data transfers. Therefore, when computing Eqn. (7.8), duplicate edges from a single node to the same receiving bin should only be counted once (Figure 7.30).

The regularity edges and critical edges can be included in the computation of Eqn. (7.8), so that they can help preserve regularity. These edges do not represent true data flow, however, and should not be used when calculating link widths.

4.3.2.1.2. Calculating $H'_{para}$

The time wasted due to assigning potentially parallel tasks to the same processor can be calculated as the sum of the weights for scheduling edges with terminating nodes occupying
Count only one of these edges

Count this edge

Figure 7.30: Correctly Accounting for Multiple Data Transfer Edges

An alternate viewpoint states that Eqn. (7.9) may overestimate the loss of parallelism. In Figure 7.31, for example, if the task represented by node A is moved into the same processor as nodes B, C, and D, the loss of parallelism within the individual processor is not nine time units (the sum of the scheduling edges). In fact, the addition of node A can delay the

Figure 7.31: Loss of Parallelism Calculated from Minimum Scheduling Edges
start of the remaining tasks no more than 5 time units (the minimum scheduling edge weight). Therefore, when calculating $H'_{\text{para}}$, the most negative scheduling edge weight between the node being moved and any node in the destination bin is used to determine the loss of parallelism. Similarly, the minimum weight of the scheduling edge between the node and the members of the bin from which it is removed gives the potential gain in parallelism.

A problem with the just given viewpoint is illustrated in Figure 7.32. Taking into account only the minimum scheduling edges shows that no increase in parallelism occurs by moving node A into the more sparsely populated bin, which is obviously false. An empirical comparison of the two competing methods of calculating loss of parallelism must therefore be carried out. The results of this comparison will be shown later.

The tasks associated with loading and unloading IPC links can be factored into $H'_{\text{para}}$ using the following approximation. If a task $t_i$ has duration $\tau_i$, and also loads and unloads $l$ and $u$ pieces of data to and from IPC links, respectively, then assign a new duration to the task of:

$$\tau'_{t_i} = \tau_i + l + u$$

(7.10)

In effect, Eqn. (7.10) assumes all the required unit-duration link loading and unloading tasks are contiguous with the execution of the original task. Making the further assumption that

![Figure 7.32: Problems with the Use of Minimum Scheduling Edges](image)
the overlap with other tasks which can proceed in parallel with \( t_i \) increases proportionately with \( \tau'_i \), we can update the weights of the scheduling edges attached to the node \( n_i \) associated with task \( t_i \) as follows:

\[
w_{e_i}^{(t')} = w_{e_i}^{(t)} \cdot \frac{\tau'_i}{\tau_i}
\]

The new scheduling edge weights can now be used to determine the additional execution time incurred due to loss of parallelism. Note that when figuring \( l \) and \( u \), the same care must be taken as when computing Eqn. (7.8), in that multiple transmissions from a single task to the same partition are only counted once. Naturally, \( l \) and \( u \) will vary as nodes move between the bins in the partitioning.

4.3.2.2. Guiding the Partitioning

While there is little leeway to alter the operation of the Kernighan-Lin partitioning algorithm, the heuristics guiding the simulated annealing algorithm can be changed to alter its performance. Here, we consider changes made necessary by the addition of new types of edges, the new objective function, and by the need to speed convergence to a solution.

4.3.2.2.1. Prediction of Annealing Curves II

The ability to predict the relationship between \( \langle H \rangle \) and the temperature \( T \) allows the simulated annealing algorithm to adjust the rate of decrease of the temperature. This speeds convergence by permitting \( T \) to be rapidly decreased in regions where the value of \( \partial \langle H \rangle / \partial T \) is low. The shape of the annealing curve is dependent upon the way \( H \) is calculated, so it would be thought that a new prediction formula would be needed to handle the following complications:

- Scheduling edges;
- Using only the maximally weighted scheduling edges when calculating \( H'_{\text{para}} \).
• Regular and critical edges;
• Using only one of multiple edges going to a bin from a single node when calculating $H'_{\text{comm}}$;
• Accounting for link delays.

Surprisingly, the above complications can be handled using approximations coupled with the original prediction method.

Scheduling edges are easily handled by merely adding them to the graph as described previously, and letting the prediction algorithm proceed as normal. To increase the accuracy of the prediction by accounting for the method used to determine loss of parallelism, the weights assigned to the scheduling edges can be altered. Since the prediction algorithm works with the sum of the edge weights crossing between bins and not just the maximum weight, the scheduling edge weights must be adjusted such that their sum will be close to this maximum when the graph nodes are randomly distributed amongst the $|B|$ bins. Therefore, all scheduling edges originating from a node $n_i$ are assigned the same weight, calculated as follows:

$$w_{se}'' = \max\{ w_{se} \mid v(e_k) = n_i \text{ and } e_k \in E_{se} \} \frac{\sum_{e} \delta(v(e_k), n_i)}{|B|}$$  \hspace{1cm} (7.11)

When regularity edges and critical edges are added to the flow graph, they are handled by the prediction algorithm in the same manner as edges over which real data flows. This results from their inclusion in Eqn. (7.8), where they are also treated as real edges.

The real data edge weights may be corrected for cases where a single node emits several edges which cross to the same bin. This can be done in a manner analogous to Eqn. (7.11). For the flowgraphs we have examined, this has not been necessary.
In order to account for transmission delays, the real, regularity, and critical edges must be multiplied by the appropriate link delay values. The prediction algorithm is unable to account for multiple links of different widths, so a constant delay is calculated as if all the processors were completely interconnected using identically sized links. This delay is used to calculate $H'_{\text{comm}}$.

Examples are easy to find where the above approximations are very poor. Surprisingly, the match between real and predicted values of $H$ is still very good. This can be attributed mainly to the statistical nature of the annealing process rather than the accuracy of the approximations.

4.3.2.2.2. Move Selection

The standard annealing algorithm randomly selects nodes and attempts to move them to other bins, which are also chosen at random. Such a selection strategy will sometimes cause a node to be moved from a stable configuration (Figure 7.33). These disruptions can lead to fragmented partitionings which display little regularity.

![Figure 7.33: Disruption of Bin Contents by Random Node Movement](image)

Figure 7.33: Disruption of Bin Contents by Random Node Movement
The method of node selection can be biased so as to select only nodes which possess neighbors in bins different from their own. Thus, only nodes on the peripheries of a bin are considered for movement, which prevents some of the disorder arising from random selection.

Once such a node \( n_i \) is selected, the bin to which it is moved can be chosen in at least two ways:

- Chose a bin occupied by one of the neighbors of \( n_i \);
- Random choice.

Moving a node only to a bin which already contains one of its neighbors prevents fragmentation of the partitioning, but it can also slow down the convergence to a final solution. This occurs because the contents of particular bins are continually swapped back and forth along their common interface, but new bins cannot take part unless three or more interfaces merge (Figure 7.34). Random choice of a destination bin can improve the speed and quality of the final solution by injecting new bins into an interface. The drawback is that a corresponding increase in disorder is incurred.

![Figure 7.34: Bin Interfaces and Interface Merging](image-url)
Using a single node swap, any system can move from a configuration such as that shown in Figure 7.33 (b) to the one in Figure 7.33 (a), but the reverse movement is not possible when biased node selection techniques are used during annealing. The reversibility of state changes is an essential characteristic which determines the behavior of annealed systems. Therefore, our annealing curve prediction method breaks down when the biased node selection techniques are used, and the simple temperature reduction rule must be used.

5. Results of Modifying the PaLS System

The previous modifications were incorporated into the PaLS automated programming system and tests were made upon the following flow graphs:

- a parallel-cascade IIR filter,
- a tree of adders,
- a least-mean squares (LMS) adaptive filter,
- an FFT.

5.1. IIR Filter Results

The modified PaLS system was applied to a parallel-cascade IIR flow graph under the following conditions:

- only scheduling edges were added,
- both scheduling and regularity edges were added,
- scheduling edges were added and critical paths were strengthened,
- scheduling and regularity edges were added and critical paths were fortified.

For each of the above conditions, the flow graph was partitioned using both Kernighan-Lin and simulated annealing algorithms. Link assignment was then performed and schedules were calculated for a single iteration of the flowgraph using heuristic H2p (i.e. favor the
scheduling of those tasks close to missing their deadlines and pack other tasks into any unused space). The duration $\tau$ of each schedule for each partitioning method was then found, giving $\tau_{KL}$ and $\tau_{SA}$ for the Kernighan-Lin and simulated annealing algorithms, respectively. Taking the reciprocal of each duration gives the speed $S$ of each schedule ($S_{KL}$ and $S_{SA}$).

A "yardstick" is now needed against which the merit of the resulting schedule speeds can be measured. An obvious and easy to calculate lower bound on the schedule duration can be found by dividing the total computational load of the flow graph by the number of available processors, thus leading to an upper bound on the speed:

$$S_{UB} = \frac{1}{\tau_{LB}} = \frac{1}{\frac{\sum w_n}{N \cdot B}}$$

The ratios $S_{KL}/S_{UB}$ and $S_{SA}/S_{UB}$ then relay how much of the maximum theoretical speedup was found by the PaLS system using each partitioning technique. The resulting ratios for the partitioning of the parallel-cascade filter of Figure 7.18 unto three processors ($B = 3$) are shown in Table 7.7. Both the Kernighan-Lin and simulated annealing partitioners found approximately the same amount of the maximum speedup for each alteration of the flow graph. The use of scheduling edges alone gives the poorest results due to the fragmented nature of the

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>$S_{KL}/S_{UB}$</th>
<th>$S_{SA}/S_{UB}$</th>
<th>$S_{KL}/S_{human}$</th>
<th>$S_{SA}/S_{human}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling edges</td>
<td>regularity edges</td>
<td>critical-path strengthening</td>
<td>$S_{KL}$</td>
<td>$S_{SA}$</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.69</td>
<td>0.68</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.76</td>
<td>0.74</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.70</td>
<td>0.72</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.71</td>
<td>0.72</td>
</tr>
</tbody>
</table>
partitioning, and the strengthening of critical paths provides little benefit. Addition of regularity edges, however, increases the amount of order in the resulting partition while decreasing the number of communications, allowing more speed to be obtained. However, the combined attractive effect of regularity edges and critical-path strengthening overpowers the repulsion of the scheduling edges, leading to an imbalanced partition with correspondingly lower speed.

Because the cost of communications is ignored when calculating $S_{UB}$, the comparison of $S_{KL}$ and $S_{SA}$ to $S_{UB}$ gives a pessimistic view of the amount of the potential speedup found by the PaLS system. In fact, the maximum realizable speed may be much lower than $S_{UB}$. Therefore, a comparison is made between $S_{KL}$ and $S_{SA}$ and $S_{human}$, the maximum speed which can be found by a human who manually partitions and schedules the flow graph while taking into account for communication delays. As can be seen from the last two columns of Table 7.7, the solutions provided by PaLS are nearly as good as those found by an intelligent human.

The data presented in Table 7.7 was collected using the scheduling edge summation technique (Eqn. (7.8)) during partitioning. Table 7.8 reflects the results of using only minimum scheduling edges when judging the movement of computational nodes between processor bins. As was mentioned previously, this technique can fail to balance computational loads. This is exactly what happened, and the deterioration in the system speed is evident for all graph alterations and types of partitioning. Therefore, only the scheduling edge summation method will be used henceforth.

The flow graph of Figure 7.18 has a natural, regular partitioning among three bins. The results of using PaLS to distribute the flow graph among four processors are given in Table 7.9. The lack of a regular, four-way, flow graph partitioning reduces the percentage of the maximum possible speedup achieved by the solutions from the PaLS system. However,
Table 7.8: PaLS Results Using Minimum Scheduling Edges

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>S_{KL}^{\frac{S_{KL}}{S_{UB}}}</th>
<th>S_{SA}^{\frac{S_{SA}}{S_{UB}}}</th>
<th>S_{KL}^{\frac{S_{KL}}{S_{human}}}</th>
<th>S_{SA}^{\frac{S_{SA}}{S_{human}}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling edges</td>
<td>regularity edges</td>
<td>critical-path strengthening</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.47</td>
<td>0.45</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.50</td>
<td>0.41</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.49</td>
<td>0.48</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.53</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 7.9: PaLS Results in the Presence of Irregularity \((B = 4)\)

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>S_{KL}^{\frac{S_{KL}}{S_{UB}}}</th>
<th>S_{SA}^{\frac{S_{SA}}{S_{UB}}}</th>
<th>S_{KL}^{\frac{S_{KL}}{S_{human}}}</th>
<th>S_{SA}^{\frac{S_{SA}}{S_{human}}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling edges</td>
<td>regularity edges</td>
<td>critical-path strengthening</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.63</td>
<td>0.60</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.59</td>
<td>0.60</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.54</td>
<td>0.51</td>
</tr>
</tbody>
</table>

PaLS performs nearly as well as a human even in the presence of irregularity. The solution found in the absence of regularity edges and critical-path strengthening is the best for both partitioning methods, since the optimal solution is not regular.

Application of techniques for extracting regularity tend to attract the flow graph nodes into only three of the bins, leaving the fourth bin sparsely populated. The effects of this load imbalance are especially noticeable when both regularity edges and critical-path strengthening are used. The possibility exists of using this deterioration of the solution as regularity measures are added to inform the user that the current number of processors and the flow graph are mismatched. This idea has not been pursued as yet.
5.2. Adder Tree Results

Table 7.10 presents the results of using the PaLS system to distribute a binary tree composed of 127 adders over a set of four processors. Since the flow graph has a natural four-way partitioning, the best results are obtained in the presence of regularity edges and critical-path strengthening (the manually generated solution is optimal, in this case).

When \( B = 3 \), no regular solution exists for the binary adder tree. Figure 7.35 depicts the solution found by PaLS with the regularity extraction options disabled. No manually

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>( S_{KL} ) / ( S_{UB} )</th>
<th>( S_{SA} ) / ( S_{UB} )</th>
<th>( S_{KL} ) / ( S_{human} )</th>
<th>( S_{SA} ) / ( S_{human} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling</td>
<td>regularity</td>
<td>critical-path</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edges</td>
<td>edges</td>
<td>strengthening</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.65</td>
<td>0.60</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.62</td>
<td>0.71</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.56</td>
<td>0.71</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0.77</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Figure 7.35: PaLS Solution for a Binary Adder Tree with \( B = 3 \)
generated solution has been found which surpasses the execution speed resulting from this non-intuitive partitioning. This gives evidence that the PaLS system will not only conserve manual effort in creating parallel systems, but may also find better solutions.

5.3. LMS Estimator Results

The fundamental LMS adaptive filter algorithm proceeds as follows:

\[
\text{for } n = 1 \text{ to } \infty \{ \\
\quad X(n) \leftarrow s(n) \cdot X(n-1) \\
\quad y(n) \leftarrow X(n)^T \cdot W(n-1) \\
\quad e(n) \leftarrow d(n) - y(n) \\
\quad W(n) \leftarrow W(n-1) + \lambda \cdot e(n) \cdot X(n) \}
\]

In the above algorithm, \(s(n)\) represents the \(n\)th sample of an unknown signal while \(d(n)\) is the desired filter output. The convolution of \(W(n-1)\) (the previous adaptive filter coefficient vector) and \(X(n)\) (the current input vector including \(s(n)\)) is used to calculate \(y(n)\) (the actual output of the adaptive filter). The difference between \(y(n)\) and \(d(n)\) can then be used to update the filter coefficients such that succeeding outputs more closely match the desired values. Figure 7.36 illustrates the topology of a small LMS adaptive filter.

The results of distributing a 64-tap LMS filter over four processing elements using the PaLS system are given in Table 7.11. The choice of a natural, four way partitioning allows a large portion of the available speedup to be found for all graph alterations, with the use of regularity edges giving the best final system configuration.

5.4. FFT Results

The final flow graph upon which PaLS was used was a 32-point FFT. Table 7.12 shows the results generated for a four processor parallel system using PaLS. Once again, the regularity edges have improved the solution to the point where it is nearly as good as one generated manually.
Figure 7.36: A 4-Tap LMS Adaptive Filter

Table 7.11: Results of Applying PaLS to a 64 Tap LMS Adaptive Filter with $B = 4$

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>$S_{KL}$ (S_{UB})</th>
<th>$S_{SA}$ (S_{UB})</th>
<th>$S_{KL}$ (S_{human})</th>
<th>$S_{SA}$ (S_{human})</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling edges</td>
<td>regularity edges</td>
<td>critical-path strengthening</td>
<td></td>
<td></td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.57 0.63 0.83 0.92</td>
<td></td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.66 0.64 0.95 0.93</td>
<td></td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.61 0.64 0.89 0.93</td>
<td></td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>0.60 0.64 0.87 0.93</td>
<td></td>
</tr>
</tbody>
</table>
Table 7.12: Results of Applying PaLS to a 32 Point FFT

<table>
<thead>
<tr>
<th>Graph Alteration</th>
<th>( \frac{S_{KL}}{S_{UH}} )</th>
<th>( \frac{S_{SA}}{S_{UH}} )</th>
<th>( \frac{S_{KL}}{S_{human}} )</th>
<th>( \frac{S_{SA}}{S_{human}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>0.58</td>
<td>0.75</td>
<td>0.72</td>
<td>0.94</td>
</tr>
<tr>
<td>•</td>
<td>0.69</td>
<td>0.74</td>
<td>0.86</td>
<td>0.93</td>
</tr>
<tr>
<td>•</td>
<td>0.62</td>
<td>0.71</td>
<td>0.78</td>
<td>0.89</td>
</tr>
<tr>
<td>•</td>
<td>0.76</td>
<td>0.77</td>
<td>0.95</td>
<td>0.96</td>
</tr>
</tbody>
</table>
1. Introduction

The main products of this research are:

- a DSP architecture which efficiently supports both uniprocessor and multiprocessor system configurations;
- the PaLS automated programming system, which eases the effort required to transform signal processing algorithms into working parallel systems.

This final chapter will try to draw some conclusions from the work which has been described, as well as point out some areas for further exploration.

2. Conclusions About the DSP

The major design goals driving the DSP architecture were:

- high throughput in uniprocessor and multiprocessor operation modes;
- efficiency and feasibility of implementation;
- ease of programming.

Any architecture design benefits from the efforts of those who have gone before. Chapter 2 presented the results of an extensive survey of existing signal processing systems. These systems suffer from one or more of the following shortcomings:

- insufficient support for multiprocessing;
- excessive specialization for a particular type of signal processing;
- high cost.
The mixture of successes and failures possessed by previous systems made it difficult to separate the important features needed by a DSP from those which were irrelevant. Therefore, an analysis was made of a collection of signal processing algorithms expressed in a high-level language (see Chapter 4). The new method of quasi-dynamic analysis (which eliminates some of the drawbacks of both static and dynamic program analysis) was applied to the program set to gather statistics concerning:

- types and sizes of data variables used;
- frequency of reference to various types of data variables;
- frequency of use of various types of operations;
- frequency of use of various types of flow control;
- nesting of subroutines and loops.

The program analysis detected several important qualities of signal processing programs which could be exploited in the design of uniprocessor aspects of the DSP. However, the specification of multiprocessor capabilities in the DSP could not be done using the results of the analysis. Therefore, a restricted dataflow method of communications was chosen as the underlying model for multiprocessing systems constructed with our DSP. This was justified based upon the use of the dataflow model (both restricted and unrestricted) by many other researchers and by the close match between the model and the signal processing application area. Unfortunately, the systems envisioned and built by these other researchers have the following difficulties:

- high cost;
- inefficient use of system resources;
- overloading of the communication facilities;
- overloading of the central system controller.
In order to combat these problems, the interprocessor communications port was designed (see Chapter 5). The IPC supports the restricted dataflow model, yet has a very efficient implementation that allows it to be included within the DSP at a very low cost. The simple method of interconnecting IPCs reduces system cost still further by eliminating external logic needed to support communications. Furthermore, the ability to incrementally allocate the communication bandwidth of the IPC over several data links permits a designer to achieve greater utilization of the system resources. Thus, multiprocessing systems can be built in which the DSPs communicate with each other over many low speed links, a few high speed links, or a mixture of the two. This ability to support numerous local buses reduces the bus contention problems encountered in dataflow systems built with a small number of global buses. The synchronization provided by the IPC also removes the need for a central communications controller.

The results of the program analysis and the IPC design were used to specify the architecture of the DSP (see Chapter 6). The natural division of the DSP datapath into separate data manipulator and address generation components was indicated by the program analysis and has also been used by other DSP designers to increase throughput. Contrary to other design efforts, however, the DSP described in this thesis includes a very sparse ALU in the data section and very general dual ALUs in the address generator. This permits the address generation section to aid the data ALU in performing computations (such as floating point arithmetic) as well as provide data addresses. In addition, the increased capabilities of the address generation section are used to control program sequencing through operations performed on the program counter and/or loop control hardware registers. This is made possible by the mapping of the program counter and loop control registers into the address generator register space, something not done by other DSP designers. This mapping, along with the non-pipelined nature of the address generation section, eases the task of specifying flow control in a signal processing application. As an added advantage, the loop counter register
values can be used to modify addresses as is commonly done in signal processing programs expressed in a high level language.

The data memory and IPC were placed close to the data manipulator in order to provide easy access to internal and external data. Small applications can be supported with only the resources of the internal RAM and ROM, but for large applications and multiprocessing systems the IPC and external memory are needed. The delays created when accessing external memory or by dataflow synchronization between DSPs using the IPC necessitate the ability to abort instructions until the data is available to complete the operation. Such a feature removes the need for the programmer to explicitly specify the synchronization of data transfers between the DSP and the external environment. Aborting instructions is supported by a modest amount of hardware included in both the data memory and IPC sections, and by a simple timing sequence which prevents the update of the processor state. The low software and hardware overhead of this technique is not found in any commercially available DSPs.

The ability of the IPC and data memory sections to independently clear the faults causing an instruction abort greatly simplifies the design of the main instruction sequencer. Further simplification is achieved through the exclusive use of wide, single-cycle instructions which are stored internally. These properties also increase the throughput by providing more control bandwidth.

In the final analysis, many of the features included in the DSP architecture to increase the throughput were also found to lessen the programming effort while simplifying the total design.

3. Further Work on the DSP

Several areas exist in which the DSP architecture can be examined, improved, or extended for greater performance. These areas will now be examined.
3.1. Further Benchmarking

The performance of the DSP must be examined in a greater variety of applications. Particular attention should be paid to more recent developments in signal processing algorithms, which will make more use of the advanced flow control and subroutine support available in our DSP than do simple filters or FFT algorithms.

3.2. Instruction Encoding

The wide instruction format used by the DSP increases throughput by providing a large control bandwidth. However, it might be possible to decrease the instruction width while preserving datapath parallelism by encoding combinations of operations which frequently occur simultaneously. It would then become possible to store larger applications internally without a loss of throughput. Fetching of instructions from an external source might also become feasible if enough width reduction occurred.

The detection of instruction combinations can be found by:

Low level analysis:

Many signal processing applications could be coded for execution on the current DSP architecture and examined for frequently used combinations of operations.

High level analysis:

The quasi-dynamic analysis which was used to guide the design of the initial DSP architecture could be extended to search the signal processing program set for groups of operations which can occur in parallel. The frequency of use of the operational sequences could then be used to determine the desirability of encoding them into specialized instructions.
3.3. Support for Floating Point Operations

Most of the hardware necessary to perform floating point operations is included in the DSP architecture. However, increased support is needed for operations which separate the mantissas and exponents for processing in the data manipulator and address generator ALUs. The transfer of the exponent calculation results from the addressing section to the data ALU is also a bottleneck when normalization is attempted during floating point addition and subtraction.

3.4. Indexed Addressing

The current DSP architecture supports address generation with post modification of the address register. However, the inclusion of the loop counters within the address register set makes it advantageous to support indexed addressing. The bus timing requires the indexed address to be output from the address ALU during phase $\phi_2$, which is impossible since the operands have not even been received by that time. The write-back of the address ALU accumulator to the register set must also be disabled.

The best way to provide indexed addressing without building a great deal of control logic is to let the data memory section handle the timing through the use of its instruction abort capabilities. A tag bit in the instruction would indicate the start of indexed addressing, and the data memory control logic would initiate a memory access but would defer the acceptance of the address until a later phase. The tag would also inform the address ALU not to write the generated address back into the address register set. Meanwhile, if a subsequent attempt is made to read the indexed data or write more data to the memory, then the data memory controller will issue a memory fault until the request can be satisfied.
3.5. Block IPC Data Transfers

The placement of the IPC next to the data memory creates the possibility of performing block transmissions and receptions of data independently of the main DSP controller. This would reduce the communications overhead by eliminating the current need for the DSP to explicitly handle loading and unloading of the IPC ports.

The block transfer facility requires the specification of additional control logic for the IPC. The data memory must also be modified such that attempts to use selected memory regions which have not yet been loaded or unloaded by the IPC cannot be read or written, respectively, by the remainder of the datapath.

3.6. Global Memory Support

Very fast transfers of blocks of data between DSPs can be supported through the construction of global memory based systems. The DSP already contains control logic which will suspend its operations until an external memory read or write can be completed. The extension of these facilities to support controlled access to an arbitrated global memory would permit the DSP to be used in an even greater variety of configurations.

4. Conclusions About PaLS

The PaLS automated programming system was motivated by the design of the IPC. Systems of arbitrarily interconnected DSPs could now be created using the IPC, but there was no easy way of determining a system configuration which provided high throughput. The PaLS system is an attempt at a solution to this problem.

There have been several earlier examples of automated programming systems for both signal processing and general applications. PaLS differs from these in several respects:

Irregular mapping:

Unlike programming systems which only map very regular computational structures
or the innermost loops of programs onto well ordered arrays of processing elements [Mold83, Lin85, Weis, Quin84, Berm85]. PaLS can imbed any computational topology into any arrangement of processing elements. Of course, the cost of this generality is the need for increased computing effort to obtain a mapping which is not guaranteed to be as regular as those found by the more restrictive techniques.

**Minimal user direction:**

Many automated programming systems require the user to specify extra information to aid in the program mapping. This information typically concerns the partitioning of the computations [Lee87, Goos86, Pope84] or the overall structure of the computation [Berm85]. The PaLS system includes two partitioning algorithms to reduce the need for such information.

**System generation:**

Most automated programming systems map algorithms onto a fixed architecture where interprocessor communication delays are determinate [Fox86]. While PaLS can operate in this mode (it reduces the difficulty), it also can generate a configuration of processors from scratch or fill in the gaps in a partially specified network.

**Algorithm specification:**

Some programming systems require the algorithm to be specified at a fine level of detail [Deny84] or a level corresponding to coarse grain dataflow [Lee87]. PaLS can work at either level or a mixture of the two.

**Extensibility:**

The mathematical formalisms used by some programming systems prevent their application to programs displaying irregularity [Mold83, Lin85], containing conditionals [Lee87], or whose structure does not fit in a family of mappable programs [Berm85]. The general nature of the Kernighan-Lin and simulated annealing partition-
ing algorithms permits the use of PaLS to be extended into these areas as well as oth-
ers.

The use of the Kernighan-Lin and simulated annealing algorithms to perform the parti-
tioning in PaLS was inspired by their use in circuit partitioning problems [Schw72, Sech85].
These same algorithms have been incorporated into several other automated programming
systems [Berm85, Fox86]. These systems, however, use the simple objective function seen
previously, which attempts to minimize the communications load between processing ele-
ments while maintaining computational balance. The tradeoff between these two quantities is
controlled through a coefficient (\(\alpha_{\text{bal}}\)), but the setting of this coefficient is non-intuitive (at
least for the systems generation task assumed for PaLS). In addition, the setting of the vari-
ous control parameters for the annealing algorithm is a non-trivial effort. Therefore, PaLS
incorporates the following innovations which remove the need for the user to set these con-
trol parameters:

*Communication delay calculation:*

In order for PaLS to optimize the objective function, it must know the communication
delays between the processing elements contained in the system. A fast algorithm for
determining the optimal widths of the IPC links interconnecting the DSPs in the sys-
tem is incorporated into the partitioning phase for this purpose.

*Use of temporal information:*

Incorporation of temporal information into the flow graph by the addition of schedul-
ing edges was used to replace the coarser measure of computational balance. This
has several advantages:

- The objective to be minimized can now be expressed as a measure of time
  only, rather than a mixture of communication delays (time) and computational
  balance (unknown units). This eliminates the need for the weighting
  coefficient.
• The partitioning phase of PaLS can now use information regarding the scheduling of tasks for execution. This can improve the overall time performance of the resulting partitioning.

Annealing curve prediction:

The change in objective function as a system is annealed (cooled) has been analytically (and sometimes crudely) predicted for wire length minimization component placement problems [Whit84] and the Travelling Salesman Problem [Bono84]. In the case of PaLS, an analytic derivation of the cost of the partitioned system versus temperature was derived and shown to closely match empirical results (Appendix B). Unlike other systems which use simulated annealing, PaLS employs the analytic prediction of the annealing curve to set the control parameters to their appropriate values. Based upon the shape of the curve, PaLS can adaptively alter these parameters to speed the convergence to a final solution.

Even with the optimal link assignment, inclusion of temporal information, and adaptive control, the PaLS system is often unable to find the most regular, optimal partitioning of a given algorithm. A simple system was analyzed which explained the failure of simulated annealing to find the optimum state in systems containing many sub-optimal solutions with near-optimal objective function values (Appendix C). In an attempt to increase the separation between the optimal and sub-optimal system configurations, regularity edges and critical-path strengthening were developed and applied to the flow graphs. These measures helped somewhat but were not complete successes.

5. Further Work on PaLS

Much work remains to be done to improve and extend the functionality of the PaLS programming system. The following sections will outline some of these areas for improvement, the problems involved, and some proposed solutions (where known).
5.1. Algorithm Input

The present methods of specifying algorithms in the PaLS system suffer from various drawbacks. A textual description of an algorithm can be difficult to decipher and does not express the flow of data as clearly as a graphical depiction. Graphical entry of an algorithm, however, can be tedious and difficult to change unless some notion of hierarchy is employed. A method fusing graphical and textual program input techniques would offer the advantages of each.

A low-level detail which must be addressed in a future version of PaLS is the tagging of multiple output edges from single computational nodes in the SFG so that the subsequent phases can determine which outputs carry identical data. This aids the partitioning, link assignment, and scheduling phases in the removal of redundant data transfers.

5.2. Setting the Number of DSPs for a System

Current users of PaLS manually select the number of processors contained in the system executing the signal processing algorithm. It was hoped that by setting this number to a large value, the partitioning phase would place tasks in a subset of the DSPs and leave the remaining ones empty, based upon how much speedup was obtained by using additional processors. Unfortunately, this technique has not worked well with the current PaLS system since additional processing elements are used as long as they provide any decrease in the execution time of the algorithm. Therefore, an improved heuristic is needed to determine when the gains are sufficient to justify the use of additional processors.

5.3. Annealing Curve Prediction

At present, the analytically derived annealing curves, which are used to guide the simulated annealing partitioner, do not employ any lower bound on the objective function. This precludes the prediction of the temperature at which the actual objective function saturates.
A prediction of this freezing point is useful because it allows the annealing to be deterministically terminated when further gains are unlikely. Therefore, the following items are needed:

- a method of calculating a lower bound on the objective function for any given flow graph;
- a prediction algorithm which can make use of the lower bound on the objective function.

5.4. Heterogeneous Processing Elements

PaLS currently creates system architectures which are composed of processing elements with identical capabilities. The ability to handle processors which are specialized to perform certain tasks is desirable. Adding this feature requires that computational nodes be assigned variable weights, corresponding to the amount of time required for their execution on the various types of available processing elements. For example, a computational node representing an FFT would have a low weight if assigned to be executed by a special-purpose FFT processor, and a higher weight if processing were to be performed by a standard DSP. Nodes with variable weights complicate the calculation of the annealing curve and the scheduling edge weights since the tasks no longer have set execution times.

5.5. Conditionals

Conditional execution of algorithm tasks is becoming more commonplace as signal processing tasks become more complex. Conditionally executed groups of tasks which are split across multiple processors will require extra communications to inform each processor as to which tasks to perform. These transfers of conditional control information must be represented in the flow graph so as to affect the partitioning. Another solution, though not as good, is to add heavily weighted edges between conditional tasks and their associated conditional control node. This forces each group of conditionally executed tasks to reside within the same processing element.
The scheduling edges must also be adjusted to account for conditional tasks. These tasks are less likely to interfere with the operation of normal tasks and cause a loss of parallelism, simply because conditional tasks are executed less frequently. Therefore, the scheduling edges between conditional and normal tasks, and between conditional tasks themselves, must be adjusted proportionally to the frequency of execution of the conditional tasks. To this end, the user must have a way of specifying the frequency of execution of conditional tasks so that the proper edge weights may be computed.

5.6. Handling Large Signal Flow Graphs

Complex signal processing algorithms may contain millions of simple operations. Obviously, using a node for each simple computation will create a very large flow graph which will require a substantial amount of time to partition and schedule. Therefore, there must be some way to group simple tasks together to form fewer, more complex nodes.

The easiest way to group computations is to require the user to specify the algorithm in a hierarchical manner. Then, aggregates of the nodes could be formed based upon the hierarchy of the algorithm. Unfortunately, the user will not always select the best hierarchy for partitioning the algorithm onto a multiprocessing system. Also, a user specified hierarchy would be relatively fixed, thus preventing the exploration of other alternatives.

Having the partitioner automatically assemble the computations into more complex tasks relieves the user of the effort and eases the exploration of the possible groupings. This can be done as follows:

*Step 1 - Initialize edge counters:*

Associate a counter with every real edge in the flow graph being partitioned. Set all of these counters to zero. Also, set \( k \), the iteration counter, to some reasonable value.
Step 2 - Perform a fast partitioning:

Use a fast partitioning algorithm upon the flow graph. A greedy algorithm or some other heuristic technique can be employed here for maximum speed.

Step 3 - Record crossing edges:

After the fast partitioning is complete, increment the counters for every real edge in the graph which crosses between bins.

Step 4 - Iterate:

Decrement $k$ and goto Step 2 if $k \neq 0$.

Step 5 - Remove crossing edges:

Remove every edge from the flowgraph whose counter exceeds a threshold value (possibly zero).

Step 6 - Assemble nodes into complex tasks:

Create a complex node for every set of nodes which remains connected after the edge removal of Step 5. These complex nodes are substituted back into the original signal flow graph.

Step 7 - Final partitioning:

Partition the compacted flow graph using one the Kernighan-Lin or simulated annealing partitioning algorithms.

Using the above algorithm on the parallel-cascade filter seen in Chapter 7 would result in removal of most of the edges connecting the individual biquad sections. These internally connected sections would then be recast as complex nodes and partitioning would proceed on a graph with $\sim 16$ nodes instead of 124 nodes.

The node collection algorithm may even be used upon its own output, thus allowing a graph to be compacted several times. Naturally there are questions as to how many fast partitionings are needed and what counter thresholds to use so as to achieve the best results.
An alternative to the above algorithm would be to detect the smaller isomorphic subgraphs which are combined to form the entire flow graph. These subgraphs could then be condensed into complex nodes to reduce the total node count. Unfortunately, the detection of subgraph isomorphism is another NP-complete problem [Ullm76]. Still, some hope exists for this technique by using extra information provided by the user or the regularity of the flow graph [Rouv86].

5.7. Limiting Phantom Edges

The addition of scheduling, regularity, and critical edges to a flow graph can increase the total number of graph edges by over 2000%! This increases the amount of memory needed to store the flow graph, and also increases the execution time for the partitioning phase of PaLS. These problems could be lessened if a method were developed for thinning the number of these phantom edges without destroying their functionality. A simple example of such a technique is shown in Figure 8.1. Here, node pairs (A,B) and (C,D) are connected via real edges, and scheduling edges exist between (A,C) and (B,D). However, the scheduling edge between A and C might be removed with only a minor effect since nodes B and D still repel each other. The attraction of A to B and C to D might be enough to compensate for the removal of the scheduling edge.

\[
\begin{array}{c}
\text{Figure 8.1: Thinning Scheduling Edges}
\end{array}
\]
5.8. Pipelining

Currently, the addition of scheduling edges to the flow graph is guided by the overlap in time between the tasks. Thus, tasks which are well separated in time will have no repelling scheduling edge between them, and so are more likely to reside in the same processor. This tends to increase the parallelism of the system if a single iteration of the program is being considered. However, as was shown previously, systems built for minimum single execution time do not necessarily give the highest throughput when repetitive execution is needed. A pipeline is usually the best system organization in such cases. Therefore, a technique is needed to create pipeline edges (analogous to scheduling edges) to guide the partitioning phase in the formation of good pipeline sections.

5.9. Link Assignment

The current link assignment algorithm attempts to create channels between each pair of processors which exchange data. If there are more than 17 DSPs in a system, it may be impossible to assign all the required links (since the IPC as shown will only support a maximum of 16 transmit and 16 receive links). Therefore, an improved algorithm which allows processors to act as relay stations between other DSPs can lead to realizable networks with reduced communication delays. The improved link assignment algorithm must execute quickly in order to be used in the partitioning phase for calculating link delays. The calculation of link delays is also complicated since the amount of time the data waits in a relay station before re-transmission is indeterminate unless a special hardware assist is built into the IPC of each DSP.

5.10. Scheduling Regularity

As was the case in partitioning, regularity is a desirable quality in the final schedule output by the PaLS system. Such regularity allows similar task sequences to be coded in the
form of loops or subroutines, thus reducing the memory needed to store the program and allowing full use to be made of the hardware features of the DSP.

5.11. New Objective Functions

The primary emphasis of the current PaLS automatic programmer is to generate a system configuration which executes an algorithm at maximum speed. There are many other system characteristics which can be optimized (memory space, quantization error, etc.) and which require objective functions to drive the phases of PaLS. An unwelcome outgrowth of trying to optimize several disparate quantities is the need to combine them into a single objective function. This requires the specification of weighting coefficients for each component of the objective function. The correct values for these coefficients are hard to determine, as we saw previously.

5.12. Artificial Intelligence

Artificial intelligence (AI) techniques, such as expert systems, could be used to assist the current PaLS system and speed the convergence to a solution. A problem with this technique, however, is the lack of a human who is proficient in building parallel systems with the given DSP and IPC architectures after which to model the expert system. We have already seen examples where the optimal algorithm partitioning is not intuitively obvious (such as in the binary adder tree example). One of the main advantages of the Kernighan-Lin and simulated annealing partitioning algorithms is their unbiased nature, which causes them to examine many strange and possibly advantageous configurations during the search for the optimum solution. Any use of artificial intelligence within PaLS should strive to maintain this unbiased quality (although this may be impossible, by definition of AI).
5.13. Source Transformations

Rearranging the source flow graph is another avenue by which PaLS can be improved. Certain transformations, such as re-timing, are already well known [Leis83]. In addition, groups of tasks could be reconnected to create more parallelism, such as by transforming a serial sequence of additions into a tree of additions. It might also be advantageous to duplicate certain tasks in different processing elements, rather than executing the task in a single processor and then incurring the cost of shipping the result where it is needed.
References

Aart85.


Abd-74.


Acke79.


Alle85.


Barn.

R. Barnes and S. Ardalan, Multiprocessor Architecture for Implementing Adaptive Digital Filters.

Batc80.


Berm85.

F. Berman, M. Goodrich, C. Koelbel, W. J. Robinson III, and K. Showell, "Prep-P: A

Bono84.


Brow85.


Deny84.


Eich86.


Elli85.


Feyn63.


Fish83.

Fish84.


Fish84.


Fox86.


Fran86.


Gare79.


Goos86.


Gree86.

Hart86.


Hays85.


Henn82.


Henn83.


Hill67.


Hu61.


Kate83.


Kawa80.


Kern70.


Kirk83.


Klok86.


Knud84.


Knut71.


Kung79.


Kung84.

Lee87.


Leis83.


Lin85.


Lind85.


Lyon84.


McCo80.


Mead80.


Mill86.

T. Miller and S. Ardalan, "A Multiprocessor Configuration for the Fast Kalman

Mitu86.

Mint81.

Mint83.

Mold83.

NEC85.

O'Le85.

Oppe75.
Patt82.


Patt84.


Pope84.


Quin84.


Reif65.


Roes86.


Rome85.


Rouv86.


Schw72.

Sech85.


Ullm76.


Vand86.


Wadg85.


Wase82.


Weis.

U. Weiser and A. Davis, *A Wavefront Notation Tool for VLSI Array Design*.

Whit84.


Wies67.

Wino78.


Wu83.

APPENDIX A

The Axe Hardware Simulation Language

1. Introduction

This appendix describes Axe, a new simulation language built upon the foundations of C language. Axe allows the user to hierarchically describe and simulate a digital system at the functional level, gate level, or a mixture of the two. Detailed statistics can be gathered during a simulation, and the timing parameters of system components can be rapidly modified to gauge the effects upon system performance. This power is achieved while maintaining the syntax of C language, so anyone familiar with C can begin programming in Axe after learning a handful of new commands.

The following topics are discussed in the remainder of this appendix:

- The difficulty in using standard C language for hardware simulations;
- The new features added to the C language to handle simulation of hardware;
- How to use the Axe compiler.

2. C Is Not a Hardware Simulation Language

At first glance, C would seem to be a good choice with which to program simulations of hardware systems. C allows the creation of functional modules which could mimic the operation of hardware components. Entire systems could be defined and simulated by joining these modules in a hierarchical fashion with function calls where the parameter lists represent input signals to the hardware component while the computed result stands for the output signal. State variables would be easily obtainable by using static variables and arrays (i.e. their
values would remain unchanged between successive invocations of the function module). C even allows the use of bit-level operations and subscripting upon properly declared data items. So, what's the problem?

The problems arise when you stop theorizing and start writing C programs to simulate hardware systems. As a simple example, let's try to create a C program to model an S-R flip-flop (Figure A.1). The flip-flop schematic shows us that a two-level hierarchy can be used where the lower level models the individual NAND gate and the upper level simulates the entire flip-flop by invoking the NAND function twice. The operation of the NAND gate can be imitated using a software routine such as this:

```c
unsigned NAND(in1, in2)
unsigned in1, in2;
{
    return( ~(in1 & in2) & 1 );
}
```

This certainly seems simple. The function accepts two inputs, ANDs them together, negates the result, and returns the least-significant bit as the output.

![Figure A.1: S-R Flip-Flop and Truth-Table](image-url)

<table>
<thead>
<tr>
<th>r</th>
<th>s</th>
<th>q(0)</th>
<th>qb(0)</th>
<th>q(1)</th>
<th>qb(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td><em><strong>illegal state</strong></em></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The simulation software for the entire flip-flop is not as straightforward:

```c
unsigned FLIP_FLOP(s, r)
unsigned s,r;
{
    static unsigned ff_state; /* state variable */
    unsigned temp;
    while(unstable) /* do until components are stable */
    {
        temp = NAND(r, ff_state); /* cross-coupled */
        ff_state = NAND(s, temp); /* NAND gates */
        unstable = chk_stability(); /* how? */
    }
    return(state); /* output = state */
}
```

First of all, the previous state of the flip-flop must be known when the FLIP_FLOP module is activated in order to achieve correct operation. In this case, the flip-flop state can be stored in a static variable called `ff_state` which is declared in the FLIP_FLOP module. But suppose at some higher level in the system hierarchy several flip-flops are used as components. Each of these flip-flops is a separate entity and has its own individual state, but the software module which simulates the flip-flops only has one `ff_state` variable. This would obviously cause erroneous operations as each flip-flop would be referring to the state of whichever flip-flop was previously activated, rather than to its own previous state. This problem is solved by declaring the `ff_state` variable to be a static array, but then the flip-flop must receive not only the present values of its inputs, but also a pointer which indicates where to find its particular state variable. Thus, some extra book-keeping is required which has more to do with the underlying software than with simulating the hardware system.

Given that the previous state is available, then the problem of modeling the cross-coupled NAND gates must be solved. NAND[1] is controlled by the FLIP_FLOP input `r` and the output of NAND[2]. Then, NAND[2] outputs a new state value dependent upon the FLIP_FLOP input `s` and the output of NAND[1]. But this may cause the output of NAND[1] to change, which in turn might cause the output of NAND[2] to change, etc., etc. Obviously,
some type of conditional loop is needed to simulate the feedback involved in the operation of NAND[1] and NAND[2], and which terminates when a stable state is achieved. But how is a stable state detected? In this case, the state is stable and the loop should terminate if neither of the outputs of NAND[1] and NAND[2] changes value within the loop. A more general way of stating this would be that the loop terminates when none of the inputs to any of the modules changes value during an iteration of the loop. In effect, the NAND gates are looked upon as data-driven elements which compute new results whenever they are given new data, and the loop is ended whenever all of the elements are quiescent, meaning no new data items can be generated to cause new activity.

However, the detection of changes in input values requires that the old input values be retained for comparison. The retention and comparison can be carried out by either the FLIP_FLOP module or the NAND module. If the NAND module handles the task itself, then the same problem that arose with the state variable is also present here: the module must store the inputs for each separate instance of itself and must receive an index into the list of data each time it is activated in order to extract the correct input values. Then, NAND must compare the past and present inputs and return a flag to FLIP_FLOP indicating whether the inputs have changed or not.

If the FLIP_FLOP module is responsible for detecting input changes, then a set of temporary variables must be declared and maintained during the execution of the loop. The number of variables needed will be affected by the number of components being simulated, and the maintenance of such a data structure could become troublesome if the hardware system is fluid and subject to change. Once again, the user is becoming bogged down in software details rather than concentrating upon the task of simulating and analyzing a system.
The flip-flop example will be discontinued now without even attempting to show how to handle multiple outputs, gather timing information and performance statistics, or output data concerning the hierarchical structure of the simulated system. By now the problems in using straight C language as a hardware simulation tool should be apparent. However, the reader may be saying to himself: "Sure, these are problems at the gate-level, but I don't need to be concerned about such low-level details since I am modeling systems at the macro-level which contain microprocessors, memories, etc." Unfortunately, while these details were illustrated at the low-level, they must also be accounted for when modeling large systems. For example, if a user is simulating a systolic array of processors, then ideally he/she would wish to define the processing element just once and then create multiple instances of it within the array. But, this creates the same problem with regard to internal states and input values as was found in the flip-flop example above.

Feedback problems also arise during the macro-level simulations of common systems, such as a microprocessor-ROM combination. Here, the operation of the microprocessor depends upon the instruction output by the ROM and the output of the ROM depends upon the address given by the microprocessor. In addition, systems exist in which there is no feedback, but the order of events is variable. Such a case is a processor connected to a read/write RAM. Sometimes, the processor writes to the RAM, in which case the output of the processor should be set first and then the RAM should be activated. But, when the processor reads from the RAM, the RAM output should be set initially, and then the processor should be activated in order to accept the RAM data.

The next section will detail the facilities of Axe which relieve the user of the drudgery of creating the software framework for a hardware simulation.
3. **Axe: Design Goals and Facilities**

The **Axe** language is an extension of C language which provides facilities for performing hardware simulation. **Axe** was designed such that the following goals were met:

*Hierarchical description:*

The user is able to partition the simulated system into separate modules which are combined to form larger modules in a tree-like manner.

*Submergence of software:*

The programs written with **Axe** reflect the hardware system configuration and are not excessively distorted by the underlying software necessary to drive the simulation. Thus, the user writes **Axe** programs as if he were actually building the system, except program modules with parameter input and output lists are used in place of chips with I/O pins. Only in special cases is the user concerned with the software infrastructure of the simulation.

*Multi-level simulation:*

Functional and gate-level modules are simultaneously supported during a simulation. Thus, during the first stages of a design, functional modules can be used for system components. As the design progresses, the modules can be partially or wholly instantiated at the gate-level and their correspondence with the functional models can be checked.

*Flexible I/O:*

The format of data input and output to the simulation is flexible and non-fixed because **Axe** has no standard I/O format or procedures. As in C, the user is required to program his own I/O functions. At the expense of slightly more work, the user has the advantage of transferring data in any form, thus facilitating the use of any pre-existing programs in concert with the simulation.
Software compatibility:

Pre-existing software modules are callable from the simulation program without the need to modify their source or object code. This gives the user of Axe immediate access to the functions in the C libraries as well as any private programs.

Portability:

The simulation module created by Axe is portable. This allows the creator of the simulation to pass the module to others for evaluation without the requirement that everyone possess (or even know) the Axe language system.

The following sections describe the enhancements to C provided by Axe and give illustrations of their use. These examples should convince the reader that Axe does indeed satisfy the above design goals.

3.1. Program Modules in Axe

Standard C language recognizes only one basic type of program module which shall be designated as a software module. Axe accepts not only software modules, but hardware modules as well. The rules for defining and using each type of module will now be given.

3.1.1. Software Modules in Axe

Software modules in Axe are identical to those in C in terms of definition and usage. This should be a great comfort to the new Axe user, because every C program he/she has ever written is also a valid Axe program. No modification of source or object code is required, so pre-existing C routines can be used in any Axe program without additional effort.

3.1.2. Hardware Modules in Axe

Hardware modules are somewhat similar to software modules in appearance, but they incorporate additional features needed for the simulation of hardware components (as illus-
trated in the previous flip-flop example). These added features include:

*Isolation of hardware entities:*

*Axe* has the ability to maintain separate lists of inputs, outputs and state (memory) variables for individual instantiations of all the hardware modules (such as the two NAND gates in the flip-flop example).

*Module stability checks:*

Hardware modules receive information regarding the stability of the sub-modules of which they are composed.

As an example of the definition and use of hardware modules, the flip-flop example will now be coded in *Axe*. The NAND gate is the lowest part of the hierarchy and is defined as follows:

\[
\text{NAND}([\text{in1, in2}](\text{out})
\]

inputs in1, in2;
outputs out;
{
    out = !(in1 & in2) & 1;
}

This definition looks quite like the C software module given previously. The first line contains the name of the hardware module (*NAND*) followed in series by empty brackets, a list of inputs, and a list of outputs. The input and output lists are enclosed in parentheses, and the individual items in each list are declared to be inputs or outputs on the next two lines. In this case, there are two inputs (*in1* and *in2*) and one output (*out*).

Following the I/O declarations is the function body enclosed in braces. In this case, the function body consists of a single line which performs the NAND function as was previously described. Note that while the inputs and outputs in this case are thought of as single bits by the user, in *Axe* they are passed to the function as multi-bit values (C unsigned integers), and so they are masked such that only the least significant bit is used.
The **NAND** gate definition can now be used to define the S-R flip-flop:

```axe
FLIP_FLOP[](s,r)(q,qb)
inputs  s, r;
outputs q, qb;
{
parallel
    { 
        NAND[1]( r, NAND[2]()(out) )();
        NAND[2]( s, NAND[1]()(out) )();
    }
    qb= NAND[1]()(out);
    q = NAND[2]()(out);
}
```

The first three lines of the **FLIP_FLOP** definition are almost identical to those of the **NAND** gate module except here there are two outputs (q and qb) instead of just one. The function body, however, shows some new features of Axe. The body starts off with the **Axe parallel** keyword followed by a block of code enclosed in braces. Suffice it to say here that this is an **Axe** hardware conditional looping construct which causes the block of code to be iteratively executed until all of the enclosed hardware modules are stable.

The **NAND** gate function is used twice within the **parallel** loop. The first call to the **NAND** module (NAND[1]) realizes a **NAND** gate whose inputs are the reset input (r) of the S-R flip-flop and the output of the second **NAND** gate in the flip-flop. The second call to the **NAND** module (NAND[2]) simulates a **NAND** gate whose inputs are the set input (s) of the flip-flop and the output of the first **NAND** gate. These two lines of **Axe** code illustrate several points:

*Creation of hardware module instantiations:*

When a hardware module is defined, the brackets following the module name are left empty ([]). When that module is used within some other hardware or software module, a string of characters is placed within the brackets to identify the particular instantiation of that hardware module. These names allow the creation of separate,
non-interfering instances of a hardware module without the user having to be concerned about the management of their input, output, and state variable storage. In this case, there are two instantiations of the NAND gate function and their names are 1 and 2, but they could as easily have been named Fred and Wilma. Identifying names for hardware module instantiations consist of arbitrary length strings of the characters a-z, A-Z, 0-9 and _.

**Executing and using the results of hardware modules:**

There are two main operations in which a hardware module is involved:

- calculating the value of its outputs based upon the value of its inputs,
- using the value of its outputs.

When calculation of the outputs is desired, the inputs to the module are placed in the first set of parentheses and the second set of parentheses are left empty. When the value of an output is used, the first set of parentheses is left empty and the name of the desired output is placed within the second set of parentheses. For example, the line:

```
NAND[1]( 1, 0 )();
```

causes the NAND function to execute and update its outputs based upon the given inputs. But, the line:

```
x = NAND[1]()(out);
```

does not cause the NAND function to execute, but instead stores in x the contents of the NAND[1] output variable named out.

**Automatic isolation of hardware entities:**

There is no need to explicitly store the state of the flip-flop as when simulation using straight C code was attempted. This requirement is eliminated because the NAND
gate hardware module maintains a list of its inputs, state variables, and outputs for each instantiation, and the outputs of the NAND gates are equivalent to the flip-flop state. Likewise, since the NAND hardware module stores the values of previous inputs to each NAND gate, it can check for stability in concert with the parallel loop construct.

The final two lines of the FLIP_FLOP code body set the outputs of the module to the values of the NAND gate outputs. This is necessary in order for the FLIP_FLOP module to communicate its results to whatever module called it. Note that these statements could have been placed within the parallel conditional loop, but we really need to set the outputs only after the cross-coupled NAND gates have stabilized.

In addition to the their slightly altered syntax, Axe hardware modules exhibit behavior quite unlike that shown by software modules. Just because a hardware module is called with a list of inputs does not imply that the body of code for that module is executed. A hardware module always has available the previous set of inputs, and it compares the present and previous inputs to see if there has been any change. If there has been a change, then the module executes its code, calculates new output and state values, and returns a flag to the calling routine indicating the occurrence of an input transition. If the present and previous inputs are identical, however, then the hardware module does not change any of its outputs or state variables, aborts its operation, and returns a flag indicating the inputs are stable. This behavior mimics the operation of actual hardware: if a hardware component is stable after a set of inputs are applied, then no activity occurs in the component until one or more inputs change value. The behavior of the hardware modules not only matches that of the physical component, but it also saves computational effort in many cases since the outputs do not have to be re-computed.
The use of hardware modules is the fundamental concept which must be mastered in Axe. The remaining sections examine the support features of Axe.

3.2. Hardware Type Declarations

The special declarations for the hardware module inputs and outputs have been seen in the previous section. The rules governing their use will now be given, and the state variable declaration will also be examined.

3.2.1. Input Declarations

Variables declared using input correspond to the physical inputs of a hardware component. The declaration must follow the initial hardware module declaration with its input and output lists, but it must precede the beginning brace of the hardware module code body. The input declaration cannot be used in any other location. Only scalar input variables are presently allowed in Axe. No arrays, structures, or pointers to input variables are allowed.

Input variables are converted to unsigned integers when the simulation is running. Therefore, values up to 16-bits (PDP-11) or 32-bits (VAX-11) may be passed to hardware modules. If inputs composed of fewer bits are desired, the user may use bit masks or the bit manipulation features of Axe (to be described later). Wider inputs can only be created using additional input variables to handle the overflow.

If the user wishes to pass variables other than inputs to a hardware module, he may do so. The entire spectrum of C data types may be used, including those defined using typedef. However, these variables will not be replicated for each instantiation of the hardware module, nor will their previous values be retained for comparison. As an example of such a declaration, consider the following:
byte_memory[](address,in_data,rd,init_mem)(out_data)
inputs address; /* address of memory location */
inputs indata; /* input data to memory */
inputs rd; /* read/write line to memory */
FILE *init_mem; /* pointer to file containing memory initialization data */
outputs out_data; /* output data from memory */
{
    . /* code body for describing memory operation */
    .
}

As can be seen, not only are hardware inputs passed to the simulated byte-wide memory module (address, in_data, rd), but also a pointer to a file containing the initialization data for the memory is transferred.

3.2.2. Output Declarations

Variables declared using output correspond to the physical outputs of a hardware component. The declaration must follow the initial hardware module declaration with its input and output lists, but it must precede the beginning brace of the hardware module code body. The output declaration cannot be used in any other location. Only scalar output variables are presently allowed in Axe. No arrays, structures or pointers to outputs are allowed.

Output variables are converted to unsigned integers when the simulation is running. Therefore, values up to 16-bits (PDP-11) or 32-bits (VAX-11) may be output from hardware modules. If outputs composed of fewer bits are desired, bit masks or the bit manipulation features of Axe may be used. Wider outputs must be explicitly created by declaring extra output variables to handle the overflow.

Unlike the case with inputs, only variables declared to be outputs can be included in the output list of the hardware module declaration. If the user wants to return other types of data from a hardware module, then a pointer to a storage location must be passed to the module in its input list (similar to the case where a C software routine must return more than
3.2.3. Memory Declarations

State variables are declared using the memory data type. An example of declaring state variables is shown using the byte-wide memory:

```c
byte_memory[] (address, in_data, rd, init_mem)(out_data)
inputs  address; /* address of memory location */
inputs  indata; /* input data to memory */
inputs  rd; /* read/write line to memory */
FILE  *init_mem; /* pointer to file containing memory initialization data */
outputs out_data; /* output data from memory */
{
  memory byte_mem[4096]; /* declare memory storage */
  /* code body for describing memory operation */
}
```

The memory declaration must occur between the opening and closing braces of a hardware module code block, and the declaration must precede the first use of the declared state variables. Scalars or arrays of state variables may be declared using the memory data type. Structures and pointers to memory variables are presently illegal.

As was the case with hardware input/output variables, state variables are converted to unsigned integers of 16- or 32-bits (depending upon the computer). Memory variables with fewer bits/word can be implemented using bit masks or the bit manipulation functions provided by Axe. Memory systems with greater word widths require the user to explicitly concatenate state variables.

Also, like I/O variables, each instantiation of a hardware module receives a separate storage area for state variables. So be careful about allocating 64K bytes of memory to a hardware module and then instantiating the module 10 times!
As a final example of the use of a state variable, a toggle flip-flop will be described in a functional style:

```c
toggle[](in)(q, qb)
inputs  in;
outputs q, qb;
{
  memory  bit;  /* flip-flop stored bit */
  if(in == 0)
    ;  /* don't change bit value */
  else
    bit=!bit;  /* toggle bit value */
  q = bit;  /* output bit value */
  qb = !bit;  /* output cmp. bit value */
}
```

3.3. The Parallel Loop Construct

The Axe parallel loop construct was first seen in the S-R flip-flop example. It is used to iterate a sequence of calls to hardware modules until all of the modules become stable. In effect, this imitates the parallel operation of the actual hardware components.

The parallel loop finds its greatest use in situations where feedback is present (as in the flip-flop with cross-coupled NAND gates), or where the order of execution of the components is not fixed (as in the case of a processor coupled to R/W RAM). It can also be used where there are several cascaded levels of combinational logic. With a sequential programming language, the outputs of the initial level of logic would have to be computed first, then the second level, and so on. Using the parallel loop, the levels can be executed in any order and eventually the system will stabilize to the correct output value. While this can lead to an increase in computational effort, actual operation of the system is more closely modeled.

Because the parallel loop only terminates when all of the enclosed hardware components are stable, the presence of an oscillation within the simulated system would cause an infinite loop. Therefore, the parallel construct allows the user to specify an upper limit on the number of iterations through the loop, and an action to be performed upon exceeding the
An example of this feature follows:

```c
parallel( 5; err_hndl("flip-flop oscillation!")) {
  NAND[1](r, NAND[2]()(out));
  NAND[2](s, NAND[1]()(out));
}
```

Here, the limit is set at 5 iterations. If the limit is exceeded, the software function `err_hndl` is passed an error message to inform the user of the event. After this action completes, the `parallel` loop is terminated and execution proceeds with the code which follows the loop.

The iteration limit can be expressed by any legal C expression which evaluates to an integer. Thus, the limit may be changed dynamically during the simulation if desired. The action to be taken upon exceeding the limit may be any legal C or Axe statement except the C `return` statement.

While it has not yet been shown, C statements and calls to software modules may reside in a `parallel` loop. Such statements may be used to functionally simulate some hardware component or output the values of system variables, as will be shown later. However, the user must be aware that these software statements will be executed a variable number of times, just like the hardware components. Therefore, the statement `x++;` within a `parallel` loop would increment `x` an unknown number of times. While this might be desirable if one wanted to know how many iterations were required before stability was reached, it can also be an unwelcome surprise to those who are unaware of the feature.

### 3.4. Bit Manipulation

**Axe** provides facilities for reading and writing a single bit or a bit-field of a variable. This feature is useful for decoding fields of an opcode or for creating a new piece of data from fields contained within several separate variables. Several examples follow which show the uses of bit-fields in **Axe**:
x = in1[0]; /* x receives the least-significant bit of in1 (either 1 or 0) */

x = y[5][4:7]; /* x is set to the value of the second nibble of y[5] (0 through 15) */

x[5] = 0; /* the sixth least significant bit of x is cleared */

y[3][8:11] += y[3][4:7]; /* the second nibble of y[3] is added into its third nibble */

x[0:7] = adder[1]()(sum)[8:15]; /* the low byte of x is loaded with the high byte of the adder output */

Several rules govern bit-field operations in Axe:

**Format:**

When accessing a bit field, the lower index comes first within the braces, separated from the higher index by a colon. When accessing a single bit, only the index of that bit is required.

**Word lengths:**

The minimum bit index is 0 (least significant bit), while the maximum index is 15 (PDP-11) or 31 (VAX-11). Values outside this range may cause error messages or erroneous operation.

**Reading and writing bit fields:**

When reading the value of a bit field, the bit field is extracted, shifted right and the upper bits are set to zero. Thus, if:

\[
x = 205;
\]

then:

\[
x[4:7] = 12 /* 12 = 00001100 */
\]

When setting the value of a bit field, if the input value exceeds the precision obtainable in the field, then the excess upper bits of the input value are removed. For instance, the operation:
\[ x\{0:3\} = 121; \quad /* 121 = 1111001 */ \]

becomes:

\[ x\{0:3\} = 9; \quad /* 9 = 1001 */ \]

**Domain of bit field operations:**

Bit-field operations are not restricted to input, output, or state variable hardware data types. They can be applied to any variable with an integer-like nature. Thus, bit-fields are applicable to integer and unsigned scalars, arrays, and structures. They can also be used on the outputs from hardware and software modules. However, care must be taken if bit-fields are used with an expression with side-effects since the results are unexpected. Therefore, stay clear of operations such as:

\[
x = (y+ + )\{2:5\}; \quad /* do not do this! */ \]
\[
x = (y+ 1)\{2:5\}; \quad /* this is OK */
\]

### 3.5. Input and Output of Simulation Data

Axe has no explicit provisions for input and output of data. Instead, the I/O routines provided by C are used. This has the disadvantage that the user may have to do slightly more work to get data into and out of his/her simulation. However, the following advantages are realized:

**Formatting freedom:**

There are no restrictions on the format of input or output data. Therefore, convenient formats for pre-processing and post-processing of data may be selected. This will possibly save the user work in the final tally, since there is no requirement for pre-existing programs to be re-written to match the data format of the simulation.

**Input freedom:**

Data can be input to the simulation at any time. This data can consist of new input
vectors, state variable initializations, performance parameters, etc.

Output freedom:

The user can examine the operation of any component of the simulated system by merely adding output statements to the component module definition. This allows the user not only to debug the simulation, but also to selectively examine the performance of system components.

As an example of the flexibility of I/O in Axe, the operation of the S-R flip flop can be monitored as follows:

```c
FLIP_FLOP[](s,r)(q,qb)
inputs s, r;
outputs q, qb;
{
    parallel
    {  
        NAND[1]( r, NAND[2]() );
        NAND[2]( s, NAND[1]() );
    }
    q = NAND[1]();
    qb= NAND[2]();
    printf("flip_flop: q=%ld qb=%ld",q,qb);
}
```

Here, the value of the flip-flop output and its complement are printed on the standard output after these values are stable. If the user has an interest in the outputs as the flip-flop searches for stability, then these transient values may be monitored like this:
The user can examine these values as the simulation runs, or the output can be stored in a file by passing the required file pointer to the FLIP_FLOP module.

An example demonstrating how data is input to the simulation will be given in a later section. Here too, the simplicity and flexibility of the method will be apparent.

3.6. Specialized Axe Variables

Programs processed by the Axe compiler contain several internal variables which can also be accessed by the user. The following sections will explain the use of these variables. Unless otherwise stated, these variables equate to C unsigned integers.

3.6.1. Hardware Module Initialization

Simulations of systems containing RAMs, ROMs, PLAs, etc. usually require these structures to be initialized to a particular state in order to function properly. This initialization should be carried out the first time the particular hardware module is activated. The first activation of a module could be detected by the user via explicitly declared flags. But since Axe automatically implants a flag for detection of the initial activation (for reasons not discussed here), the flag is also made available to the user to eliminate some duplicated effort and wasted memory space. The INIT flag evaluates to a true (non-zero) condition if it is the first time that a particular instantiation of a hardware module has been entered. Thereafter,
the flag evaluates to a false (zero) condition. Any attempt to overwrite the INIT flag will result in an error. Multiple instantiations of a hardware module receive multiple INIT flags so that each instance of the module can perform its own initialization procedure. An example of the use of the INIT flag is shown below:

```c
PLA[](in)(out) /* this module simulates a PLA */
inputs in; /* input to PLA */
outputs out; /* output from PLA */
{
    memory pla_prog[64]; /* PLA personality matrix */
    if(INIT)
    {
        /* first time in, so initialize the PLA truth table */
        rd_pla("prog.tpla", pla_prog);
    }
    /* now, output the value selected by the input */
    out = pla_prog[in(0:5)];
}
```

The first activation of the PLA module (as indicated by INIT!=0) causes the initialization of the personality matrix with the contents of the file prog.tpla. The rd_pla routine is a user-defined program which merely reads the data from the named file and deposits it in an array. The routine is written to accept the data format of the file, instead of forcing the user to create his files in accordance with a pre-defined simulator format. This allows the simulator to accept data from the same file which will be used to drive a software tool for generating the VLSI masks of the PLA. Thus, changes in the design file are immediately reflected in the simulation as well as the VLSI artwork. This coherence is made possible by the fluid I/O interface of Axe.

### 3.6.2. Hardware Module Names

Sometimes it is useful to be able to identify a particular instantiation of a hardware module. Axe maintains two character strings for this purpose:
GLOBAL_NAME:

This is a pointer to a character string which shows the hierarchical nesting for a particular instantiation of a module.

LOCAL_NAME:

This is a pointer to a character string which gives the module name and instantiation name for the currently executing hardware module.

As an example of the difference between these identifiers, suppose a logic circuit consisting of two S-R flip-flops named Ward and June is simulated (Figure A.2). Then, the global and local names for the NAND gates within the flip-flops are:

<table>
<thead>
<tr>
<th>FLIP_FLOP[Ward] NAND gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>LOCAL_NAMES:</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FLIP_FLOP[June] NAND gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOBAL_NAMES:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LOCAL_NAMES:</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

3.6.3. Hardware Module Statistics

Axe presently provides two variables for gauging the utilization of hardware modules. These two variables and their functions are:

ATTEMPTS:

Records the number of times the hardware module was called, regardless of whether the inputs have changed value.

ENTRIES:

Records the number of times the hardware module was called and activated by a
change in the value of its inputs. The value in this counter is always less than or equal to the value contained in ATTEMPTS.

These counters are maintained for every instantiation of a hardware module. Their values may be read and output at any time in order to get an estimate of how module utilization varies over time. New values can also be loaded into either counter, thus allowing the user to reset their values to zero.

Output of the values contained in the utilization counters for each hardware module instantiation is completely up to the user. There are presently no facilities planned to allow the dumping of the counter values at periodic intervals or at the end of a simulation.

3.6.4. System Timing

System timing analysis is accomplished through the use of the TIME Axe variable (which is a long, signed, integer C variable). This variable may be read or written, allowing a great deal of flexibility in reporting and altering system timing.

TIME can be considered as a local variable within a given instantiation of a hardware module. When the module is entered, TIME is set to zero. Then, if the module calls other
modules in a sequential manner, their execution times are added to \texttt{TIME} as follows:

```c
// example of sequential timing
int inputs in1;
int outputs out1;
{
    \texttt{TIME} = 0 at this point
    \texttt{hmod[1]}(in1); /* \texttt{TIME} = \texttt{TIME} + \texttt{time(hmod[1])} */
    \texttt{hmod[2]}(in1); /* \texttt{TIME} = \texttt{TIME} + \texttt{time(hmod[2])} */
    \texttt{hmod[3]}(in1); /* \texttt{TIME} = \texttt{TIME} + \texttt{time(hmod[3])} */
    \texttt{TIME} = \texttt{time(hmod[1])} + \texttt{time(hmod[2])} + \texttt{time(hmod[3])} */
}
```

The situation changes if hardware modules are activated concurrently using the \texttt{parallel} construct. In this case, the maximum execution time of all the modules within the \texttt{parallel} loop is added to \texttt{TIME} at the end of each loop iteration as follows:

```c
// example of concurrent timing
int inputs in1;
int outputs out1;
{
    \texttt{TIME} = 0 */
    \texttt{parallel}
    {
        \texttt{hmod[1]}(in1);
        \texttt{hmod[2]}(in1);
        \texttt{hmod[3]}(in1);
        \texttt{TIME} = \texttt{TIME} + \texttt{MAX(time(hmod[1]), time(hmod[2]), time(hmod[3]))} */
    }
    \texttt{TIME} = \texttt{MAX(iteration \#1)} +
    \texttt{MAX(iteration \#2)} +
    \texttt{MAX(iteration \#3)} + ... */
}
```

Naturally, combinations of sequential and concurrent hardware module activations may be used within a given module and the \texttt{TIME} variable will be modified accordingly.

Once a hardware module has completed execution, the value stored in \texttt{TIME} will be passed back to the calling module so that it may be added to the \texttt{TIME} variable for that module. In this way, the execution times for all the lower levels of the system hierarchy are
eventually reflected in the TIME variable at the top of the hierarchy. This value may be output to indicate the timing performance of the system.

Because TIME may also be over-written by the user, it provides a means of rapidly altering component performance in order to check the effect upon overall system timing. For example, suppose the tst_mod hardware module was written as follows:

```c
#include <stdio.h>

void tst_mod( in1, out1 )
{
    hmod[1](in1); /* TIME = =0 at this point */
    TIME += time(hmod[1]); /* TIME = TIME + time(hmod[1]) */
    hmod[2](in1); /* TIME = TIME + time(hmod[2]) */
    hmod[3](in1); /* TIME = TIME + time(hmod[3]) */
    TIME = TIME/2;
    out1 = TIME;
}
```

Now, the execution time for tst_mod has been halved, and the effect this has on total system timing can be gauged to determine the benefits. Any arithmetic operation can be carried out on TIME in such a manner.

Software modules handle TIME in a different manner than hardware modules. Software modules record the execution times of any sequentially or concurrently activated hardware modules in the same manner as hardware modules do. However, software modules never pass their TIME value back to the calling module. Thus, all software modules can be thought of as having execution times of zero (i.e. software is infinitely fast), and therefore they have no effect upon system timing.

4. Using the Axe Compiler

Figure A.3 depicts the overall sequence of actions required to perform a simulation using Axe. The Axe compiler operates upon the user's source file to create a corresponding C program. This C program is then compiled using the standard C compiler, and the resulting
executable file performs the simulation.

The format for calling the Axe compiler is as follows:

axe [-clnst] file.axe

The user first enters axe, then whatever options are desired, and then the name of the source file (which should end with the extension .axe). The options and their associated actions are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Suppress the generation of the final Axe code (used for linking)</td>
</tr>
<tr>
<td>l</td>
<td>Enlarge input, output, and memory variables to 32-bits</td>
</tr>
<tr>
<td>n</td>
<td>Disable global and local module names</td>
</tr>
<tr>
<td>s</td>
<td>Retain Axe comments in the generated C code</td>
</tr>
<tr>
<td>t</td>
<td>Enable timing analysis</td>
</tr>
</tbody>
</table>

The Axe compiler produces two output files:

file.tree:

This file contains a hierarchical description of the simulated system. Eventually, tools will be created which allow the user to examine this file in a pictorial manner.

file.c:

This file contains the C code created from the Axe code as well as the necessary structure definitions, storage allocations, and #define statements. Applying the C compiler to this file will create an executable module which simulates the system.
described in Axe.

Once the C code file has been generated, the user need only compile it using the command:

c  file.c

Then, the system can be simulated by running the resultant executable module (in this case a.out).

5. Conclusions

The preceding sections have outlined the need for extending C language to handle the simulation of hardware systems, and the constructs provided by Axe to meet these needs. Axe shields the user from the need to create a software infrastructure to support simulations while allowing the description of hardware in a hierarchical fashion.

Axe is newly created and mutates on a daily basis as additional requirements are found. A five-bit counter (Figure A.4) and the associated Axe description are given as a non-trivial example of the use of Axe. Examination of this source code will give further examples of how to use the constructs previously described. But if questions on usage still remain, just try it and see what happens!

/* 5-bit Counter Example:
 *
 * This Axe code simulates a 5-bit counter using NAND gates as the basic building block. */
/* main software routine which must be present in every Axe program */
main()
{
    int i; /* use LSB of i to generate a clock signal */
    for(i=0; i<100; i++)
    {
        counter[1](i&1); /* run counter for 50 cycles */
    }
}
Figure A.4: Module Hierarchy for a 5-bit Counter
/* basic NAND gate */
  nand[](a,b)(out)
  inputs a,b;
  outputs out;
  {
    out = ~(a&b)&1;
  }

/* basic flip-flop made from cross-coupled NAND gates */
  ff[](s,r)(q,qb)
  inputs s,r;
  outputs q,qb;
  {
    parallel
      {
        nand[1](r, nand[2]()(out))();
        nand[2](s, nand[1]()(out))();
      }
    q = nand[2]()(out);
    qb = nand[1]()(out);
  }

/* clocked flip-flop which accepts new s,r inputs when clock is high */
cff[](s,r,clk)(q,qb)
inputs s,r,clk;
outputs q,qb;
{
  parallel
    {
      /* clocked input NAND gates */
      nand[1](s,clk)();
      nand[2](r,clk)();
      /* basic flip-flop module */
      ff[1](nand[1]()(out), nand[2]()(out))();
    }
    q = ff[1]()(q);
    qb = ff[1]()(qb);
  }
/* toggle flip-flop built from two cascaded clocked flip-flops */
tgl[](clk)(out)
inputs clk;
outputs out;
{
    parallel
    {
        cff1[cff2]()(qb), cff[2]()q, clk();
        cff[2]()cff1[](q), cff[1]()qb!),clk();
    }
    out = cff[2]()q;
}

/* 5-bit counter built from 5 cascaded toggle flip-flops */
counter[](clk)(out)
inputs clk;
outputs out;
{
    parallel
    {
        tgl[0]()clk();
        tgl[1]()tgl[0]()out();
        tgl[2]()tgl[1]()out();
        tgl[3]()tgl[2]()out();
        tgl[4]()tgl[3]()out();
    }
    out[0] = tgl[0]()out();
    out[1] = tgl[1]()out();
    out[2] = tgl[2]()out();
    out[3] = tgl[3]()out();
    out[4] = tgl[4]()out();
    /* output counter value when clock is high */
    if(clk)
    {
        printf("%s: %d,GLOBAL_NAME", out);
    }
}
APPENDIX B

Prediction of Simulated Annealing Curves for Graph Partitioning

1. Motivation

Applying the technique of simulated annealing to an optimization problem such as graph partitioning requires knowledge of certain parameters:

- The starting temperature, \( T_0 \);
- The rate of temperature decrease, \( \alpha_T \);
- The number of iterations needed to reach equilibrium at a given temperature;
- The freezing temperature of the system, \( T_f \).

The values the parameters should have are usually not intuitively obvious from an inspection of the problem. However, the function detailing the relationship between the average objective function value and temperature, \( <H(T)> \), can be used to intelligently set the values without resorting to trial annealing runs.

The average energy of a physical system which follows Boltzman statistics is:

\[
<E(T)> = \frac{\int_{-\infty}^{\infty} E \eta_S(E) e^{-\frac{E}{kT}} dE}{\int_{-\infty}^{\infty} \eta_S(E) e^{-\frac{E}{kT}} dE}
\]

Thus, the average energy of the system is directly dependent on the density of states, \( \eta_S(E) \), at various energies. In order to predict the analogous \( <H(T)> \), we need the density of graph partitionings, \( \eta_P(H) \), with which to calculate:
This appendix details how $\eta_P(H)$ and $<H(T)>$ are calculated for graph partitioning problems, and how this information can be used to set the parameters for simulated annealing.

2. A Simple Case: Graph Bipartitioning

When faced with a complex task, it is usually better to attempt to solve a simpler subcase of the main problem in order to develop some insight. For this reason, the problem of sectioning a graph into two pieces (bipartitioning) such that an objective function $H$ is minimized will be considered.

The objective function used is $H = H_{\text{cross}} + \alpha_{\text{bal}} H_{\text{bal}}$, which consists of components reflecting the cost of edges crossing between bins and the imbalance of the partitioning. If we could evaluate $H(p_i)$ for each possible partitioning $p_i$, then $\eta_P(H)$ would be proportional to the histogram of the number of partitionings at each value of $H$. Unfortunately, even a small graph with 100 nodes has $2^{100} \approx 10^{30}$ possible bipartitionings. Thus, direct calculation of $\eta_P(H)$ is impractical and a probabilistic technique must be used. Imagine that a random choice is made from the set of all possible bipartitionings of a particular graph. The $H$ value of the chosen partitioning is recorded, the partition is replaced, and another selection is made. Eventually, a probability distribution $\pi(H)$ of the costs of the graph bipartitions will take shape. This probability distribution is proportional to the state density, so it will serve our purposes if we can determine it (or even approximate it). The objective function consists of two components, so we must find the distributions of the edge crossing cost, $\pi(H_{\text{cross}})$, and the balance cost, $\pi(H_{\text{bal}})$, in order to arrive at $\pi(H)$.
Determining the probability density of the crossing edge contribution to the bipartitioning cost of a graph has several complicating factors:

- The edges do not all have the same weight;
- The number of nodes in each bin is not fixed;
- The nodes usually have different edge degrees.

Therefore, we will simplify a bit more. First of all, assume all edges have a weight of 1. The distribution of edge weights will be factored in later. We may also fix the number of nodes in each bin at \( |b_1| \) and \( |b_2| \), and determine the distribution of crossing edges for this condition. Once this simpler distribution is found, it can be used to calculate \( \pi(H_{\text{trans}}) \) when the bin contents are allowed to change.

The problem of nodes with differing numbers of entering and exiting edges cannot be handled by assuming all of the original nodes have the same edge degree. But, if each node is envisioned as a collection of simpler nodes each of which is adjacent to just one edge, then the graph can be simplified such that all the new nodes have the same degree (Figure B.1). The price we pay for this simplification is the loss of some correlation terms in the distribution function, since the new nodes can move independently of one another, whereas they

![Figure B.1: Decomposing Nodes](image-url)
were bound in groups in the original graph.

Since each graph edge contributes two simple nodes, there are $2|E|$ such nodes after the original graph is decomposed. Within each of the partition bins $b_1$ and $b_2$, there are $\Delta_1$ and $\Delta_2 = 2|E| - \Delta_1$ simple nodes, respectively. We must determine the function which expresses the probability $\pi(c \mid \Delta_1, \Delta_2)$ of having $c$ edges crossing between $b_1$ and $b_2$ for a given $\Delta_1$ and $\Delta_2$.

It is not intuitively obvious what $\pi(c \mid \Delta_1, \Delta_2)$ should be, but it is possible to define it recursively. Assume that we randomly pick a simple node $n_i$ from the graph. This node will be connected to one other node $n_j$ through their common edge. If all the nodes have been assigned to a particular partition, the edge could be in one of four possible states:

- Case 1: the edge is internal to $b_1$ with $\rho(n_i) = \rho(n_j) = b_1$
- Case 2: the edge is internal to $b_2$ with $\rho(n_i) = \rho(n_j) = b_2$
- Case 3: the edge is external with $\rho(n_i) = b_1$ and $\rho(n_j) = b_2$
- Case 4: the edge is external with $\rho(n_i) = b_2$ and $\rho(n_j) = b_1$

The probability of each of the preceding cases is:

$$
\pi_1 = \frac{\Delta_1}{\Delta_1 + \Delta_2} \cdot \frac{\Delta_1 - 1}{\Delta_1 + \Delta_2 - 1} \quad \pi_2 = \frac{\Delta_2}{\Delta_1 + \Delta_2} \cdot \frac{\Delta_2 - 1}{\Delta_1 + \Delta_2 - 1}
$$

$$
\pi_3 = \frac{\Delta_1}{\Delta_1 + \Delta_2} \cdot \frac{\Delta_2}{\Delta_1 + \Delta_2 - 1} \quad \pi_4 = \frac{\Delta_2}{\Delta_1 + \Delta_2} \cdot \frac{\Delta_1}{\Delta_1 + \Delta_2 - 1}
$$

The recursive definition can now be stated as:

$$
\pi(c \mid \Delta_1, \Delta_2) = \pi_1 \pi(c \mid \Delta_1 - 2, \Delta_2) + \pi_2 \pi(c \mid \Delta_1, \Delta_2 - 2) + \pi_3 \pi(c - 1 \mid \Delta_1 - 1, \Delta_2 - 1) + \pi_4 \pi(c - 1 \mid \Delta_1 - 1, \Delta_2 - 1)
$$

(B.1)

This equation does not offer any simple solution to our problem, either, but it will at least serve as a check on any solution we do find.

In looking for $\pi(c \mid \Delta_1, \Delta_2)$, we note that the total number of distinct bipartitions with $|b_1| = \Delta_1$ and $|b_2| = \Delta_2$ is:
\[ |P_{\Delta_1, \Delta_2}| = \frac{(\Delta_1 + \Delta_2)!}{\Delta_1! \Delta_2!} \]

But, we can also look at a particular bipartitioning with \(|b_1| = \Delta_1, |b_2| = \Delta_2,\) and \(c\) crossing edges as an assignment of each edge \(e_i\) to one of three classes with the following sizes:

- Crossing edges: \(\rho(v_+(e_i)) \neq \rho(v_-(e_i))\) and \(|E_c| = c;\)
- Edges internal to \(b_1\): \(\rho(v_+(e_i)) = \rho(v_-(e_i)) = b_1\) and \(|E_{b_1}| = \frac{\Delta_1 - c}{2};\)
- Edges internal to \(b_2\): \(\rho(v_+(e_i)) = \rho(v_-(e_i)) = b_2\) and \(|E_{b_2}| = \frac{\Delta_2 - c}{2};\)

Now, the total number of partitionings with \(c\) edges is:

\[ |P_{c|\Delta_1, \Delta_2}| = \frac{2^c |E|!}{|E_c|! |E_{b_1}|! |E_{b_2}|!} \]

where the factor of \(2^c\) comes about since each edge in \(E_c\) can be "flipped" such that its terminating nodes switch bins. Thus, the probability of having \(c\) crossing edges with bin sizes of \(\Delta_1\) and \(\Delta_2\) is:

\[ \pi(c | \Delta_1, \Delta_2) = \frac{|P_{c|\Delta_1, \Delta_2}|}{|P_{\Delta_1, \Delta_2}|} = \frac{\Delta_1! \Delta_2! \left[ \frac{\Delta_1 + \Delta_2}{2} \right]! 2^c \left[ \frac{\Delta_1 - c}{2} \right]! \left[ \frac{\Delta_2 - c}{2} \right]! c!}{\Delta_1 + \Delta_2! \left[ \frac{\Delta_1}{2} \right]! \left[ \frac{\Delta_2}{2} \right]! c!} \]  

(B.2)

The validity of Eqn. (B.2) can be checked by inserting it into Eqn. (B.1).

What does \(\pi(c | \Delta_1, \Delta_2)\) look like? Figure B.2 shows that it resembles a Gaussian distribution function when \(\Delta_1 \approx \Delta_2\). The crossing edge distribution does skew as \(\Delta_1\) and \(\Delta_2\) diverge, but only when \(\Delta_1 \ll \Delta_2\). When a graph is bipartitioned, the balancing term in the objective function makes this condition unlikely. Therefore, if we can determine the average number of crossing edges \(\bar{c}_{\Delta_1, \Delta_2}\) and the standard deviation \(\sigma_{c|\Delta_1, \Delta_2}\), Eqn. (B.2) can be approximated as:

\[ \pi(c | \Delta_1, \Delta_2) \approx \frac{1}{\sqrt{2\pi} \sigma_{\bar{c}_{\Delta_1, \Delta_2}}} \exp\left[ -\frac{(c - \bar{c}_{\Delta_1, \Delta_2})^2}{2\sigma_{\bar{c}_{\Delta_1, \Delta_2}}^2} \right] \]
Assuming a Gaussian distribution, \( \bar{\tau}_{\Delta_1, \Delta_2} \) can be found by solving:

\[
\frac{\partial}{\partial c} \left[ \log_e \pi(c \mid \Delta_1, \Delta_2) \right] \bigg|_{c, \Delta_1, \Delta_2} = 0
\]

Now:

\[
\log_e \pi(c \mid \Delta_1, \Delta_2) = \log_e \Delta_1! + \log_e \Delta_2! + \log_e \left[ \frac{\Delta_1 + \Delta_2}{2} \right]! + c \log_e 2
\]

\[
- \log_e [\Delta_1 + \Delta_2]! - \log_e \left[ \frac{\Delta_1 - c}{2} \right]! - \log_e \left[ \frac{\Delta_2 - c}{2} \right]! - \log_e c
\]

which can be transformed using \( \log_e x! \approx x \log_e x - x + 1 \) and much manipulation into:

\[
\log_e \pi(c \mid \Delta_1, \Delta_2) \approx \Delta_1 \log_e \Delta_1 + \Delta_2 \log_e \Delta_2 - c \log_e c - 1
\]

\[
- \frac{1}{2} \left[ (\Delta_1 + \Delta_2) \log_e (\Delta_1 + \Delta_2) + (\Delta_1 + c) \log_e (\Delta_1 + c) + (\Delta_2 + c) \log_e (\Delta_2 + c) \right]
\]

Taking the derivative and setting it to zero gives:

\[
\frac{\partial}{\partial c} \left[ \log_e \pi(c \mid \Delta_1, \Delta_2) \right] \bigg|_{c, \Delta_1, \Delta_2} \approx \frac{1}{2} \log_e (\Delta_1 - \bar{\tau}_{\Delta_1, \Delta_2}) + \frac{1}{2} \log_e (\Delta_2 - \bar{\tau}_{\Delta_1, \Delta_2}) - \log_e \bar{\tau}_{\Delta_1, \Delta_2} = 0
\]

which can be solved to find \( \bar{\tau}_{\Delta_1, \Delta_2} \), the average number of crossing edges in a bipartitioning.
when $|b_1| = \Delta_1$ and $|b_2| = \Delta_2$:

$$
\bar{c}_{\Delta_1, \Delta_2} = \frac{\Delta_1 \Delta_2}{\Delta_1 + \Delta_2}
$$

Eqn. (B.3) is symmetrical with respect to $\Delta_1$ and $\Delta_2$ as it should be, and it closely predicts the actual crossing edge average in Figure B.2.

To derive the standard deviation for the distribution of Eqn. (B.2), we perform the substitution $c = \bar{c}_{\Delta_1, \Delta_2} + c'$ and find:

$$
\frac{\partial}{\partial c} \left[ \log_e \pi(c | \Delta_1, \Delta_2) \right] = \frac{\partial}{\partial c'} \left[ \log_e \pi(c | \Delta_1, \Delta_2) \right] 
\approx \frac{1}{2} \log_e (\Delta_1 - \bar{c}_{\Delta_1, \Delta_2} - c') + \frac{1}{2} \log_e (\Delta_2 - \bar{c}_{\Delta_1, \Delta_2} - c') - \log_e (\bar{c}_{\Delta_1, \Delta_2} + c')
$$

If we assume that the bins are roughly balanced such that:

$$
\Delta_2 = \Delta_1 + \xi \quad \text{where} \quad \xi \ll \Delta_1 \quad \text{and} \quad \xi \ll \Delta_2
$$

then $\bar{c}_{\Delta_1, \Delta_2}$ may be approximated as:

$$
\bar{c}_{\Delta_1, \Delta_2} = \frac{\Delta_1 \Delta_2}{\Delta_1 + \Delta_2} = \frac{\Delta_1 (\Delta_1 + \xi)}{\Delta_1 + (\Delta_1 + \xi)} = \frac{\Delta_1 + \xi}{2 + \frac{\xi}{\Delta_1}} \approx \frac{\Delta_1 + \xi}{2} \cdot (1 - \frac{\xi}{2 \Delta_1})
$$

Using Eqn. (B.5) we can make the following transformations:

$$
\Delta_1 - \bar{c}_{\Delta_1, \Delta_2} = \frac{\xi}{2 \Delta_1} \left( 1 - \frac{\xi}{2 \Delta_1} \right)
$$

$$
\Delta_2 - \bar{c}_{\Delta_1, \Delta_2} = \frac{3 \xi}{2 \Delta_1} \left( 1 + \frac{3 \xi}{2 \Delta_1} \right)
$$

$$
\bar{c}_{\Delta_1, \Delta_2} + c' = \frac{\xi}{2 \Delta_1} \left( 1 + \frac{\xi}{2 \Delta_1} + \frac{2 c'}{\Delta_1} \right)
$$

Substituting into Eqn. (B.4) leads to:

$$
\frac{\partial}{\partial c'} \left[ \log_e \pi(c | \Delta_1, \Delta_2) \right] \approx \frac{1}{2} \log_e \left( 1 - \frac{\xi}{2 \Delta_1} - \frac{2 c'}{\Delta_1} \right) + \frac{1}{2} \log_e \left( 1 + \frac{3 \xi}{2 \Delta_1} - \frac{2 c'}{\Delta_1} \right) - \log_e \left( 1 + \frac{\xi}{2 \Delta_1} + \frac{2 c'}{\Delta_1} \right)
$$

By assuming that $c' \ll \Delta_1$ (i.e. the deviations from $\bar{c}_{\Delta_1, \Delta_2}$ are small compared to the number of simple nodes in the smaller bin), we can use the approximation $\log_e (1 + x) \approx x$ to simplify Eqn. (B.6) to:
\[
\frac{\partial}{\partial c'} \left[ \log \pi(c \mid \Delta_1, \Delta_2) \right] \approx \frac{3c}{2\Delta_1} - \frac{2c'}{\Delta_1} + \frac{3\xi}{2\Delta_1} - \frac{2c'}{\Delta_1} + \frac{\xi}{2\Delta_1} - \frac{2c'}{\Delta_1}
\]

\[
\approx -\frac{4c'}{\Delta_1}
\]

Since we assumed the probability distribution for crossing edges was nearly Gaussian:

\[
\pi(c \mid \Delta_1, \Delta_2) \approx \frac{1}{\sqrt{2\pi}\sigma_{c \mid \Delta_1, \Delta_2}} \exp \left[ -\frac{(c' - \bar{\Delta}_c, \Delta_2)^2}{2\sigma_{c \mid \Delta_1, \Delta_2}^2} \right]
\]

\[
\log \pi(c \mid \Delta_1, \Delta_2) \approx K - \frac{(c' - \bar{\Delta}_c, \Delta_2)^2}{2\sigma_{c \mid \Delta_1, \Delta_2}^2}
\]

\[
\frac{\partial}{\partial c'} \left[ \log \pi(c \mid \Delta_1, \Delta_2) \right] \approx -\frac{c'}{\sigma_{c \mid \Delta_1, \Delta_2}}
\]

Equating Eqn. (B.7) and Eqn. (B.8) finally gives us:

\[
\sigma_{c \mid \Delta_1, \Delta_2} \approx \frac{\sqrt{\Delta_1}}{2}
\]

Unfortunately, Eqn. (B.9) is not symmetrical with respect to \(\Delta_1\) and \(\Delta_2\). This naturally leads us to question its validity. However, Figure B.3 shows that Eqn. (B.9) is a good approximation despite its asymmetry.

While the relationship between the number of crossing edges and simple nodes has been uncovered, what is really needed is the probability of having \(c\) crossing edges with the original \(|b_1|\) and \(|b_2|\) complex nodes in each bin. This can be determined if we first find the distribution of \(\Delta_1\) with respect to \(|b_1|\). The parameter \(\Delta_1\) is "built" from the edge degree contributions of the original nodes in \(b_1\):

\[
\Delta_1 = \sum_{n, \xi} \xi_{n, c}
\]

Now, the edge degree of a node is a random variable with mean \(\bar{\xi}\) and standard deviation \(\sigma_{\xi}\). If enough of these random variables are added together (say more than five), then the distribution of \(\Delta_1\) is approximately Gaussian with a mean and standard deviation of:
Figure B.3: Deviation of \(\frac{\sqrt{\Delta_1}}{2}\) from \(\sigma_{\Delta, \Delta_1}\)

\[
\bar{\Delta}_1 = \bar{\xi} |b_1|
\]

\[
\sigma_{\Delta_1} = \sigma_{\bar{\xi}} \sqrt{|b_1|}
\]

The distribution of \(c\) with respect to \(\Delta_1\) and the distribution of \(\Delta_1\) with respect to \(|b_1|\) are both nearly Gaussian, so the average number of crossing edges for a particular value of \(|b_1|\) is:

\[
\bar{c}_{|b_1|} = \frac{\int_{\Delta_{\text{min}}}^{\Delta_{\text{max}}} \int_{\epsilon_{\text{min}}}^{\epsilon_{\text{max}}} c \cdot \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] \cdot \exp \left[ -\frac{(c - \bar{\epsilon}_{\Delta_1, \Delta})^2}{2\sigma_{\epsilon_{\Delta_1, \Delta}}^2} \right] d\epsilon \cdot d\Delta_1}{\int_{\Delta_{\text{min}}}^{\Delta_{\text{max}}} \int_{\epsilon_{\text{min}}}^{\epsilon_{\text{max}}} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] \cdot \exp \left[ -\frac{(c - \bar{\epsilon}_{\Delta_1, \Delta})^2}{2\sigma_{\epsilon_{\Delta_1, \Delta}}^2} \right] d\epsilon \cdot d\Delta_1}
\]
where we have assumed that the Gaussian distribution for \( c \) becomes negligible at \( c_{\min} \) and \( c_{\max} \). Making the same supposition concerning \( \Delta_1 \) allows us to continue as follows:

\[
\bar{c}_{\mid b, l} = \frac{\int_{-\infty}^{\infty} \bar{c}_{\Delta, \Delta} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] d\Delta_1}{\int_{-\infty}^{\infty} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] d\Delta_1}
\]

\[
= \int_{-\infty}^{\infty} \left( \Delta_1 - \Delta_1^2 \right)^{1/2} \sqrt{2\pi\sigma_{\Delta_1}} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] d\Delta_1
\]

\[
= \bar{\Delta}_1 - \frac{\sigma_{\Delta_1}^2 + \bar{\Delta}_1^2}{2|E|}
\]

where we have used the relation:

\[
\bar{c}_{\Delta, \Delta} = \frac{\Delta_1^2}{\Delta_1 + \Delta_2} = \frac{\Delta_1 (2|E| - \Delta_1)}{2|E|} = \Delta_1 - \frac{\Delta_1^2}{2|E|}
\]

The variance for the crossing edge distribution is found in a similar manner:

\[
\sigma_{\bar{c}_{\mid b, l}}^2 = \frac{\int_{\Delta_{\min}}^{\Delta_{\max}} \left( \sigma_{\bar{c}_{\Delta, \Delta_1}}^2 + (\bar{c}_{\Delta, \Delta_1} - \bar{c}_{\mid b, l})^2 \right) \cdot \frac{1}{\sqrt{2\pi\sigma_{\Delta_1}}} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] d\Delta_1}{\int_{\Delta_{\min}}^{\Delta_{\max}} \frac{1}{\sqrt{2\pi\sigma_{\Delta_1}}} \exp \left[ -\frac{(\Delta_1 - \bar{\Delta}_1)^2}{2\sigma_{\Delta_1}^2} \right] d\Delta_1}
\]
After much manipulation, we get:

\[
\sigma^2_{\bar{c}|b_i|} = \frac{\Delta_1}{4} + \frac{3\sigma^4_{\Delta_i}}{2|E|^2} + \frac{\sigma^2_{\Delta_i}^2}{|E|^2} - \frac{2\Delta_1\sigma_{\Delta_i}^2}{|E|} + \sigma_{\Delta_i}^2
\]

We usually find that \( \frac{\sigma^2_{\Delta_i}}{|E|} \ll \Delta_i \), so it is possible to reduce the number of computations by using the approximations:

\[
\bar{c}_{|b_i|} \approx \bar{c}_{\Delta_i, \Delta_i} \quad \quad \sigma_{c|b_i|} \approx \sigma_{c|\Delta_i, \Delta_i}
\]

The cost of a bipartitioning with bin sizes of \( |b_1| \) and \( |b_2| \) is a random variable and can be expressed as:

\[
H_{|b_i|} = c_{|b_i|} + \alpha_{|b_i|} \left| |b_1|-|b_2| \right|
\]

when all edges are assigned unit cost. Assuming \( c \) has a Gaussian distribution, then \( H_{|b_i|} \) is Gaussian with a mean and standard deviation of:

\[
\bar{H}_{|b_i|} = \bar{c}_{|b_i|} + \alpha_{|b_i|} \left| |b_1|-|b_2| \right| \quad \quad \sigma_H = \sigma_{c|b_i|}
\]  (B.10)

Eqn. (B.10) applies for a particular value of \( |b_1| \), but \( |b_1| \) may range from zero to the total number of nodes in the original graph. Therefore, the probability distribution for \( H \) as \( |b_1| \) varies is just the weighted sum of the distributions of \( H_{|b_i|} \) for each particular value of \( |b_1| \):

\[
\pi(H) = \sum_{|b_i| = 0}^{|N|} \omega_{|b_i|} \frac{1}{\sqrt{2\pi}\sigma_{H}} \exp \left[ -\frac{(H - \bar{H}_{|b_i|})^2}{2\sigma_H^2} \right]
\]

The weighting factor \( \omega_{|b_i|} \) is the fraction of possible bipartitionings of \( |N| \) nodes which have
Now we have the probability distribution for $H$, $\pi(H)$, which is proportional to the partitioning state density $\eta_p(H)$. Finally, we are ready to determine $\langle H(T) \rangle$:

$$\langle H(T) \rangle = \frac{\int_{H_{mn}}^{H_{mn}} H \pi(H) e^{-\frac{H}{T}} dH}{\int_{H_{mn}}^{H_{mn}} \pi(H) e^{-\frac{H}{T}} dH}$$

Expanding and rearranging leads to:

$$\langle H(T) \rangle = \frac{\int_{H_{mn}}^{H_{mn}} \sum_{|b_i| = 0}^{N} \omega_{|b_i|} \frac{1}{\sqrt{2\pi\sigma_H}} \exp \left[-\frac{(H - \tilde{H}_{|b_i|})^2}{2\sigma^2_{\tilde{H}}} \right] \exp \left[-\frac{H}{T} \right] dH}{\int_{H_{mn}}^{H_{mn}} \sum_{|b_i| = 0}^{N} \omega_{|b_i|} \frac{1}{\sqrt{2\pi\sigma_H}} \exp \left[-\frac{(H - \tilde{H}_{|b_i|})^2}{2\sigma^2_{\tilde{H}}} \right] \exp \left[-\frac{H}{T} \right] dH}$$

Using the relation:

$$\exp \left[-\frac{(H - \tilde{H}_{|b_i|})^2}{2\sigma^2_{\tilde{H}}} \right] \exp \left[-\frac{H}{T} \right] = \exp \left[\frac{\sigma^2_{\tilde{H}}}{2T^2} - \frac{\tilde{H}_{|b_i|}}{T} \right] \exp \left[-\frac{(H - \tilde{H}_{|b_i|} + \frac{\sigma^2_{\tilde{H}}}{T})}{2\sigma^2_{\tilde{H}}} \right]$$

we find:
\[
<H(T)> = \frac{\sum_{|b_i| = 0}^{\lfloor N \rfloor} \omega_{|b_i|} \exp \left[ \frac{\sigma_{H}^2}{2T^2} - \frac{\tilde{H}_{|b_i|}}{T} \right] \int_{H_{\min}}^{H_{\max}} \exp \left[ -\frac{(H - \tilde{H}_{|b_i|} + \frac{\sigma_{H}^2}{T})^2}{2\sigma_{H}^2} \right] dH}{\sum_{|b_i| = 0}^{\lfloor N \rfloor} \omega_{|b_i|} \exp \left[ \frac{\sigma_{H}^2}{2T^2} - \frac{\tilde{H}_{|b_i|}}{T} \right] \int_{H_{\min}}^{H_{\max}} \exp \left[ -\frac{(H - \tilde{H}_{|b_i|} + \frac{\sigma_{H}^2}{T})^2}{2\sigma_{H}^2} \right] dH}
\]

If we assume that:

\[H_{\min} + 3\sigma_{H} < \tilde{H}_{|b_i|} - \frac{\sigma_{H}^2}{T} < H_{\max} - 3\sigma_{H}\]

then \(H_{\min}\) and \(H_{\max}\) can be replaced by \(-\infty\) and \(\infty\), respectively. This simplifies \(<H(T)>\) to:

\[
<H(T)> = \frac{\sum_{|b_i| = 0}^{\lfloor N \rfloor} \omega_{|b_i|} \exp \left[ \frac{\sigma_{H}^2}{2T^2} - \frac{\tilde{H}_{|b_i|}}{T} \right] \left( \tilde{H}_{|b_i|} - \frac{\sigma_{H}^2}{T} \right)}{\sum_{|b_i| = 0}^{\lfloor N \rfloor} \omega_{|b_i|} \exp \left[ \frac{\sigma_{H}^2}{2T^2} - \frac{\tilde{H}_{|b_i|}}{T} \right]}
\]

(Eqn. B.11)

Eqn. (B.11) details how the average cost of a partitioning will vary with temperature assuming unit cost edges and nodes in the graph. Not much can be determined about the relationship between \(<H(T)>\) and \(T\) from a cursory examination of Eqn. (B.11), but if we allow only exact bipartitions such that \(\omega_{\lfloor N \rfloor/2} = 1\) and the remainder are zero, then:

\[
<H(T)> \big|_{\lfloor N \rfloor/2} = \tilde{H}_{\lfloor N \rfloor/2} - \frac{\sigma_{H}^2}{T}
\]

(Eqn. B.12)

Eqn. (B.12) shows how the average cost of the exact bipartitions generated by the simulated annealing method decreases as \(T\) is lowered.

As a demonstration of the validity of Eqn. (B.11), Figure B.4 shows the actual and predicted annealing curves for a bipartitioning of a graph.

3. Multipartitioning

The previous section derived the relationship between the average cost of a graph bipartition and the annealing temperature. However, it is not easy to adapt Eqn. (B.2) for the case
Figure B.4: Annealing Curve Prediction

- Actual
- Predicted

\[ N = 150 \]
\[ \bar{H} = 300 \]
\[ \alpha = 0.4 \]
where \(|B| > 2\). Therefore, a simpler formulation will be introduced.

Suppose the original graph nodes are again decomposed into simple nodes as demonstrated in Figure B.1. Then, the probability that an edge crosses between any two bins is:

\[
\pi_{\text{cross}}(\Delta) = \sum_{\Delta, \pi_{\Delta_i}} \pi_{\Delta_i} \sum_{\Delta, \pi_{\Delta_j}} \pi_{\Delta_j},
\]

where \(\pi_{\Delta_i}\) is just the probability that a simple node will occupy bin \(b_i\). If there are \(|E|\) edges, then:

\[
\pi_{\Delta_i} = \frac{|\Delta_i|}{2|E|}
\]

Now, the number of ways to select \(c\) crossing edges from \(|E|\) total edges is:

\[
\frac{|E|!}{c! (|E|-c)!}
\]

and the probability of any of the above combinations is:

\[
(1 - \pi_{\text{cross}}(\Delta))^{\vert E \vert - c} \pi_{\text{cross}}(\Delta)
\]

which leads to a binomial distribution of crossing edges with respect to a set of given bin sizes:

\[
\pi(c | \Delta) = \frac{|E|!}{c! (|E|-c)!} (1 - \pi_{\text{cross}}(\Delta))^{\vert E \vert - c} \pi_{\text{cross}}(\Delta)
\]

The binomial distribution approximates a Gaussian for sufficiently large \(|E|\), which correlates with the Gaussian shaped crossing edge distribution found for bipartitions in Eqn. (B.2).

The mean and variance of this binomial distribution are:

\[
\bar{c}_\Delta = \pi_{\text{cross}}(\Delta) \vert E \vert \tag{B.13}
\]

\[
\sigma^2_{\bar{c}_\Delta} = \pi_{\text{cross}}(\Delta) (1 - \pi_{\text{cross}}(\Delta)) \vert E \vert \tag{B.14}
\]

It is comforting to note that if we restrict ourselves to bipartitioning such that \(\Delta' = \{\Delta_1, \Delta_2\}\), then Eqn. (B.3) can be re-derived as follows:

\[
\bar{c}_{\Delta'} = \left[ \frac{\Delta_1 \Delta_2}{\Delta_1 + \Delta_2} + \frac{\Delta_2 \Delta_1}{\Delta_1 + \Delta_2} \right] \frac{\Delta_1 + \Delta_2}{\Delta_1 + \Delta_2} = \bar{c}_{\Delta_1, \Delta_2}.
\]
If we only allow strict partitioning such that:

$$|\Delta_i| = |\Delta_j| = \frac{2|E|}{|B|} \forall i, j$$

then we find:

$$\pi(c|\Delta|) = \frac{2|B|! - 1}{(|B| - 2)! 2! |B|_2}$$  \hspace{1cm} (B.15)

Substituting Eqn. (B.15) into Eqn. (B.13) and Eqn. (B.14) leads to:

$$\frac{c^2|\gamma|}{|B|} = \frac{|B| - 1}{|B|} |E|$$  \hspace{1cm} (B.16)

$$\sigma_c^2 \frac{c^2|\gamma|}{|B|} = \frac{|B| - 1}{|B|_2} |E|$$  \hspace{1cm} (B.17)

Note that if $|B| = 2$, then $\Delta_1 = \Delta_2 = |E|$ and:

$$\sigma_c \frac{c^2|\gamma|}{|B|} = \sqrt{\frac{2 - 1}{2^2}} \Delta_1 = \frac{\Delta_1}{2} = \sigma_c \Delta_1 \Delta_1$$

which corresponds to Eqn. (B.9).

Since exact partitioning was enforced in deriving Eqn. (B.16) and Eqn. (B.17), then it can be assumed that there will be no imbalance cost term in $H$. Therefore, the objective function is determined solely by the cost of crossing edges:

$$H \frac{|N|}{|B|} = c \frac{2|\gamma|}{|B|}$$

As in the bipartitioning case, $H$ is a Gaussian random variable with mean and variance of:

$$\bar{H} \frac{|N|}{|B|} = \bar{c} \frac{2|\gamma|}{|B|}$$  \hspace{1cm} (B.18)

$$\sigma_{H} = \sigma_c \frac{c^2|\gamma|}{|B|}$$  \hspace{1cm} (B.19)

We can then predict the annealing curve for exact multipartitioning from Eqn. (B.12) with Eqn. (B.18) and Eqn. (B.19) added from above:

$$<H(T)> \frac{|N|}{|B|} = \frac{2|\gamma|}{|B|} - \frac{\sigma_c^2 \frac{c^2|\gamma|}{|B|}}{T}$$
Using Eqn. (B.20), it is now very easy to handle graphs with edges of differing weights. Suppose all of the edges had the same weight $w$, then $\tilde{H} = wH$ and $\sigma^2_{\tilde{H}} = w^2\sigma^2_H$, which transforms Eqn. (B.20) into:

$$<H(T)>_{|E|} = w \frac{|B|-1}{|B|} |E| \left[ 1 - \frac{w}{|B| T} \right]$$

Now suppose that the graph contains several sets of edges $E_i$ such that all the members of each set have the same weight $w_{E_i}$. Then we can make a family of subgraphs by decomposing the original graph into simple nodes as before, and then form separate subgraphs containing all the simple nodes connected by edges of the same weight. If each subgraph is considered to be independent of the others (which is approximately true since we ignore the dependencies between simple nodes originating from the same complex nodes), then during annealing each subgraph will follow a curve described by:

$$<H(T)>_{E_i} = w_{E_i} \frac{|B|-1}{|B|} |E_i| \left[ 1 - \frac{w_{E_i}}{|B| T} \right]$$

The objective function for the partitioning of the entire graph is just the sum of the costs of the subgraphs:

$$<H(T)>_{|E|} = \sum_{E_i} w_{E_i} \frac{|B|-1}{|B|} |E_i| \left[ 1 - \frac{w_{E_i}}{|B| T} \right]$$

$$= \frac{|B|-1}{|B|} \left[ \sum w_{E_i} |E_i| - \frac{1}{|B| T} \sum w^2_{E_i} |E_i| \right]$$
where $\bar{w}$ and $\sigma_{w}^2$ are the mean and variance of the edge weights in the original graph, respectively.

A comparison of an actual annealing curve and the curve predicted by Eqn. (B.21) is shown in Figure B.5. In this case, exact partitioning has been enforced by setting $\alpha_{bol} = 10000$ which results in a good match between the empirical and analytical curves. This is not necessary, however, as can be seen in Figure B.6 where $\alpha_{bol} = 0.4$. The match is not as good at higher temperatures, reflecting the tendency of the system to reside in unbalanced configurations because of the low balancing force. As temperature decreases balance is restored, however, since the balancing force becomes comparable to the forces exerted by the random thermal effects. As a result, a close match is achieved between the actual and theoretical annealing curves at lower temperatures.

Figure B.5: Annealing Curve Prediction for Exact Multipartitioning
4. Using the Knowledge of the Annealing Curve

Knowledge of the behavior of $<H(T)>$ as $T$ varies can be used to guide the simulated annealing process during graph partitioning. The specific parameters which can be set are:

**The starting temperature:**

The temperature $T_0$ at which $<H(T)>$ saturates can be found by solving:

$$<H(T_0)> = A - \frac{B}{T_0} = \beta_{T_0}A$$

where $\beta_{T_0} < 1$. In our system, we arbitrarily use $\beta_{T_0} = 0.99$. The system is extremely fluid at $T_0$ since the temperature is high enough to make any partitioning likely. The low probability of the system being trapped in a local optima at $T_0$ makes it a good temperature from which to start annealing.

**The rate of temperature decrease:**

The following equation is used to set the $\alpha_T$ factor such that the system does not cool
too rapidly:

\[ \langle H(\alpha_T T) \rangle = \beta_T \langle H(T) \rangle \]  

(B.22)

where \( \beta_T < 1 \). In our system, \( \beta_T \) is set in the range \([0.9, 0.99]\). Lowering \( T \) such that \( H \) undergoes only a small decrease prevents the annealing algorithm from mistakenly freezing into some local optimum, as might happen with a non-adaptive temperature scheduler. Eqn. (B.22) also allows \( T \) to be rapidly lowered where \( \frac{d}{dT} \langle H(T) \rangle \) is small. This increases the rate of convergence of the annealing algorithm.

**Thermal equilibrium:**

The onset of thermal equilibrium can be detected when the average \( H \) of the randomly generated partitions matches with \( \langle H(T) \rangle \) from the analytical curve. Unfortunately, the actual and predicted values of \( \langle H(T) \rangle \) can sometimes be significantly different as in Figure B.6. Therefore, this technique is usually not recommended.

**The freezing point:**

The temperature \( T_f \) at which the partitions are frozen is determined from:

\[ \langle H(T_f) \rangle = 0 \]

Usually there is more improvement which can be made at this point since the analytical model deviates from reality at extremely low temperatures. Therefore, \( T_f \) can be used as a threshold after which a faster algorithm (such as Kernighan-Lin or a greedy technique) is used to finish off the partitioning.

Aside from setting the parameters used in the annealing algorithm, the predicted curve of \( \langle H(T) \rangle \) can be used in conjunction with other types of partitioning algorithms to speed the convergence to a solution. It is possible to use a fast heuristic technique to find a solution with a low objective function value \( H_{\text{fast}} \), and then use simulated annealing to improve the solution [Gree86]. The analytical expression for \( H \) can be used to determine the temperature
from which annealing should start as follows:

\[ \langle H(T_0) \rangle = H_{\text{fast}} \]  

(\text{B.23})

Choosing the correct starting point is important since too high a temperature will significantly disorder the solution found by the fast heuristic, reducing the value of finding it in the first place. On the other hand, too low a starting temperature raises the probability of freezing the system into a sub-optimal solution.

Using a starting solution found by a fast heuristic does have a drawback: if the solution is sub-optimal but has a low value for \( H_{\text{fast}} \), then the starting temperature found from solving Eqn. (\text{B.23}) will probably be too low to "jump" the system out of the potential valley in which it resides. Thus, the quality of the solution may be determined mainly by the starting heuristic algorithm and the annealing method will have no significant effect.
APPENDIX C

The Approach to Thermal Equilibrium

1. Introduction

The standard simulated annealing technique seldom partitions signal flowgraphs such that minimal interprocessor communication is achieved. Instead, a sub-optimal partition is usually generated which is very close to the true optimum. The analysis of a simplified system as it undergoes annealing can show why the optimum solution is seldom found.

2. Markov Analysis for a Simplified System

Each unique partitioning of a graph corresponds to a state within the total state space $P$. A cost $H(p_i)$ is assigned to each state $p_i$, reflecting the merit of the associated partitioning with respect to the number of edges crossing between bins and the load balance of the bins. Therefore, finding the optimum partition is the same as finding the state with the minimum cost.

A flowgraph with $|N|$ nodes to be partitioned amongst $|B|$ bins would have an $|N|$-dimensional state space populated with $|P| = |B|^{|N|}$ unique states. Even small flowgraphs (~100 nodes) which are bipartitioned would have enormous state spaces ($2^{100} \approx 10^{30}$) distributed over a many-dimensional (and non-visualizable) state space. In order to lessen the difficulties of the upcoming analysis, it is desirable to examine a simplified system which retains the following salient features of the flowgraph partitioning problem state space:

- One or more optimum states exist;
- Sub-optimal states exist with costs that are close to optimum;
- Moving from a sub-optimal to optimal state requires the traversal of states with much
higher costs.

The cost-state diagram for a system meeting the above criteria is shown in Figure C.1. The state space contains one optimum state \( (p_0) \), one sub-optimal state \( (p_{|p|-1}) \), and \( |p|-2 \) states of equal energy which form a barrier between the optimum and sub-optimal states. The system may transfer from one state to another as long as the states are adjacent (i.e., a transfer such as \( p_1-p_2 \) is legal, but \( p_1-p_3 \) is not). The probability of the system transferring from state \( p_i \) to \( p_j \) is affected by the number of possible candidate transitions and by the energies associated with each state:

\[
\pi(p_i\rightarrow p_j) = \pi_{\text{choose}}\{p_i\rightarrow p_j\} \times \min\left\{1.0, \exp\left(-\frac{H(p_i) - H(p_j)}{T}\right)\right\}
\]

As can be seen, transfers to a state with a lower cost are always accepted, while transfers in the reverse direction are moderated by the annealing temperature \( T \). For large values of \( T \), if a cost-increasing jump is selected, then the probability of accepting it is \( \sim 1 \); while decreasing \( T \) lowers the acceptance probability. The ability to jump to states of higher cost allows the

![Cost-State Diagram for a Simplified System](image-url)
system to cross over barriers and arrive in the optimum state. Curtailing this ability, via
decreases in the temperature, hopefully causes the system to reside in the optimum state at
the end of the annealing process.

During the annealing process, the selection of the next state is dependent only upon the
current state of the system. Therefore, the system behavior may be described using a *Markov
chain*:

Each node represents a state and the directed arcs are labeled with the probability of transferring from one state to another. All state transitions are equiprobable and temperature independent with $\pi(p_i-p_j) = \frac{1}{n}$, except for:

\[
\pi(p_0-p_1) = \alpha = \exp \left[ -\frac{\Delta}{T} \right]
\]
\[
\pi(p_0-p_0) = 1 - \alpha = 1 - \exp \left[ -\frac{\Delta}{T} \right]
\]
\[
\pi(p_{|p|-1}-p_{|p|-2}) = \beta = \exp \left[ -\frac{\delta}{T} \right]
\]
\[
\pi(p_{|p|-1}-p_{|p|-1}) = 1 - \beta = 1 - \exp \left[ -\frac{\delta}{T} \right]
\]

The transition probability matrix $Q$ which represents the above Markov chain is:

\[
Q = \begin{bmatrix}
1-\alpha & \frac{1}{n} & 0 & 0 \\
\alpha & 0 & \frac{1}{n} & 0 \\
0 & \frac{1}{n} & 0 & \frac{1}{n} \\
0 & 0 & \frac{1}{n} & 0 \\
\end{bmatrix}
\]

\[
Q = \begin{bmatrix}
0 & \frac{1}{n} & 0 & 0 \\
\frac{1}{n} & 0 & \frac{1}{n} & 0 \\
0 & \frac{1}{n} & 0 & \beta \\
0 & 0 & \frac{1}{n} & 1-\beta \\
\end{bmatrix}
\]
The probability of the system occupying any given state after \( n \) transitions is contained in the vector:

\[
\Pi(n) = \begin{bmatrix}
\pi_p(n) \\
\pi_{p_1}(n) \\
\vdots \\
\pi_{p_{n-2}}(n) \\
\pi_{p_{n-1}}(n)
\end{bmatrix}
\]

Because of the Markovian nature of the system, the probability of occupation of any state is determined solely from the previous state probabilities and the transition matrix as follows:

\[
\Pi(n) = Q \Pi(n-1)
\]

The recursive aspect of the above equation leads to:

\[
\Pi(n) = Q \Pi(n-1) = QQ \Pi(n-2) = \cdots = Q^n \Pi(0)
\]

where \( \Pi(0) \) is the initial system state probability distribution. Taking the z-transform of this equation gives:

\[
z\{\Pi\} = [I - Qz^{-1}]^{-1} \Pi(0)
\]

\[
= \frac{D(z)\Pi(0)}{\det(I - Qz^{-1})}
\]

\[
= \frac{D(z)\Pi(0)}{(\lambda_0 - z^{-1})(\lambda_1 - z^{-1}) \cdots (\lambda_{|P|-2} - z^{-1})(\lambda_{|P|-1} - z^{-1})}
\]

\[
= \frac{D(z)\Pi(0)}{(1 - z^{-1})(\lambda_1 - z^{-1}) \cdots (\lambda_{|P|-2} - z^{-1})(\lambda_{|P|-1} - z^{-1})}
\]

where the final equation results from the fact that \( \det(I - P) = 0 \) so the eigenvalue \( \lambda_0 = 1 \).

Doing a partial fraction expansion leads to:

\[
z\{\Pi\} = \frac{D_0(z)\Pi(0)}{1 - z^{-1}} + \frac{D_1(z)\Pi(0)}{\lambda_1 - z^{-1}} + \cdots + \frac{D_{|P|-2}(z)\Pi(0)}{\lambda_{|P|-2} - z^{-1}} + \frac{D_{|P|-1}(z)\Pi(0)}{\lambda_{|P|-1} - z^{-1}}
\]

Performing the inverse transform gives:
\[ \Pi(n) = \Pi_\infty + \frac{1}{\lambda_1} \Pi_1 + \frac{1}{\lambda_2} \Pi_2 + \cdots + \frac{1}{\lambda_{|P| - 2}} \Pi_{|P| - 2} + \frac{1}{\lambda_{|P| - 1}} \Pi_{|P| - 1} = \Pi_\infty + \sum_{i=1}^{|P| - 1} \frac{1}{\lambda_i} \Pi_i \]

Thus, the state probability vector consists of the constant term \( \Pi_\infty \), along with a set of transient terms which decay to zero as \( n \to \infty \). Remembering the physical basis for the annealing technique, \( \Pi_\infty \) corresponds to the state distribution when the system reaches thermal equilibrium. The transient terms reflect the difference between the current and equilibrium distributions after \( n \) perturbations of the system.

Now assume that the system attained equilibrium at a temperature \( T \), after which the temperature was decreased by \( \Delta T \). Then, the initial value of the transient term as annealing begins at the new temperature would be:

\[ \Pi_\infty(T) - \Pi_\infty(T - \Delta T) \]

Ideally, this transient must be reduced to zero (i.e. thermal equilibrium must be attained at temperature \( T - \Delta T \)) before the simulated annealing technique can schedule another temperature decrease. Completely eliminating the transient term, however, requires an infinite number of system perturbations, which is impossible. Therefore, the annealed system eventually falls out of thermal equilibrium as a result of the limited number of state changes allowed at each temperature. The tendency of the annealing technique to converge on sub-optimal solutions to the partitioning problem can be traced to this failure to maintain thermal equilibrium.

3. Divergence from Equilibrium

Applying the above ideas to an even simpler system containing only three states (i.e. one optimal state \( p_0 \), one sub-optimal state \( p_2 \), and a barrier state \( p_1 \) separating them) gives the following equations:
Solving for the equilibrium distribution probabilities gives:

\[
Q = \begin{bmatrix}
1 - \alpha & \frac{\beta}{\alpha} & 0 \\
\alpha & 0 & \beta \\
0 & \frac{\beta}{\alpha} & 1 - \beta
\end{bmatrix} \quad \Pi = \begin{bmatrix}
\pi_{p_0} \\
\pi_{p_1} \\
\pi_{p_2}
\end{bmatrix} \quad Q\Pi = \Pi
\]

By allowing $T \to 0$, the "frozen" system distribution is found to be:

\[
\pi_{p_0} = \frac{\beta}{\alpha + \beta + 2\alpha \beta} \quad \pi_{p_1} = \frac{2\alpha \beta}{\alpha + \beta + 2\alpha \beta} \quad \pi_{p_2} = \frac{\alpha}{\alpha + \beta + 2\alpha \beta}
\]

This is exactly what is wanted: the probability that the annealing will terminate with the optimal solution is one. However, this distribution results from assuming that an infinite number of states could be visited in order to achieve thermal equilibrium. If only a finite number of perturbations are allowed at each temperature, then the system distribution diverges from the ideal case (Figure C.2). The divergence at each temperature for a specified $\Delta T$ can be quantified by comparing the norm of the system distribution after $M$ state transitions to the equilibrium distribution as follows:

\[
\Delta \Pi(M, T, \Delta T) = 1 - \frac{\Pi^T(M, T, \Delta T) \cdot \Pi(M, T, \Delta T)}{\Pi_{\alpha}^T \cdot \Pi_{\alpha}}
\]

where:

\[
\Pi(M, T, \Delta T) = Q^M \cdot \Pi_{\alpha}(T + \Delta T) \\
\Delta T = (1 - \alpha_T) \cdot T
\]

As can be seen from the graph, the annealing algorithm cannot maintain the equilibrium of the system even using a relatively large number of state transitions at each temperature ($M = 100$) and a slow temperature reduction ($\alpha_T = 0.99$). Even further increases in $M$ do not mitigate the divergence from equilibrium as the system temperature is lowered. Therefore, once annealing ends, there is a significant probability that the system will end up in the suboptimal state.
Figure C.2: Divergence of the Actual and Ideal Distributions
4. Impossibility of Maintaining Equilibrium

It might be asked if there is some way to maintain equilibrium by varying the number of states visited by the annealing algorithm at each temperature. The answer is yes, but the number of visited states grows exponentially with decreasing $T$, as will be seen.

The primary deterrent to achieving equilibrium in the 3-state example is the tendency for the system to get trapped in the sub-optimal state $p_2$. If there are $S$ identical systems in equilibrium, then at any temperature $T$ there will be $S \pi_{p_i}^e(T)$ systems which reside in state $p_2$. If the temperature is decreased by $\Delta T$, how many system perturbations are needed to re-establish thermal equilibrium such that $S \pi_{p_i}^e(T-\Delta T)$ of the systems are in state $p_2$? Making the assumption that once a system leaves state $p_2$ it will not return, we find the number of systems which have left $p_2$ after $k$ state transitions by each system is:

$$S_{p_i-p_i} = S \cdot \pi_{p_i}(T) \cdot [ \beta + (1-\beta)\beta + (1-\beta)^2\beta + \cdots + (1-\beta)^k \beta ]$$

$$= S \cdot \pi_{p_i}(T) \cdot [ 1 - (1-\beta)^k ] \quad (C.1)$$

The migration of systems out of state $p_2$ causes the eventual establishment of equilibrium at the new temperature:

$$S \cdot \pi_{p_i}(T) - S \cdot \pi_{p_i}(T) \cdot [ 1 - (1-\beta)^k ] = S \cdot \pi_{p_i}(T-\Delta T)$$

The number of state transitions needed to bring about thermal equilibrium can now be determined:

$$k_e = \log_e \pi_{p_i}(T-\Delta T) - \log_e \pi_{p_i}(T) \over \log_e (1 - \beta) \quad (C.2)$$

The assumption used to find Eqn. (C.1) gives an optimistic estimate of the number of systems leaving $p_2$, since system migration back into $p_2$ is ignored. Therefore, Eqn. (C.2) is a lower bound on the number of perturbations required to establish equilibrium. Eqn. (C.2) becomes more enlightening if we ask what the maximum temperature decrease $\Delta T_e$ is which still allows us to regain thermal equilibrium with just one perturbation of the system. To deter-
mine this, we must solve for $\Delta T_e$ in the following equation:

$$k_e = 1 = \frac{\log \pi_p(T - \Delta T_e) - \log \pi_p(T)}{\log(1 - \beta)}$$

This becomes:

$$1 - \beta = \frac{\pi_p(T - \Delta T_e)}{\pi_p(T)}$$

Performing the substitution:

$$\pi_p(T - \Delta T_e) \approx \pi_p(T) - \Delta T_e \cdot \frac{\partial \pi_p}{\partial T}$$

leads to:

$$\Delta T_e = \frac{\beta \pi_p(T)}{\left( \frac{\partial \pi_p}{\partial T} \right)}$$

Entering the expressions for $\alpha$, $\beta$, and $\pi_p$, gives:

$$\Delta T_e = \frac{T^2 \cdot \left\{ \exp \left[ -\frac{\delta}{T} \right] + \exp \left[ -\frac{\Delta}{T} \right] + 2 \cdot \exp \left[ -\frac{\Delta + \delta}{T} \right] \right\}}{\Delta - \delta + 2 \delta \cdot \exp \left[ -\frac{\Delta}{T} \right]}$$

(C.3)

Recalling that $\Delta \geq \delta \geq 0$, then as $T \to 0$ we may simplify Eqn. (C.3) to:

$$\Delta T_e \approx T^2 \cdot \exp \left[ -\frac{\delta}{T} \right]$$

The final equation shows expresses the fact that, as the system approaches the freezing point, the amount the temperature may be decreased per system perturbation exponentially decreases. Standard simulated annealing algorithms often use a fixed number of perturbations and decrease the temperature via the rule: $\Delta T' = (1 - \alpha) \cdot T$. As $T \to 0$, then $\Delta T' \gg \Delta T_e$ (see Figure C.3), so the annealed system falls out of equilibrium, thus explaining the tendency to converge to a non-optimal solution.
Figure C.3: Standard Annealing and Equilibrium-Maintaining Temperature Decrease

\[ \Delta = 50 \]
\[ \delta = 45 \]
\[ \alpha_T = 0.99 \]
3. The DSP Instruction Store and Controller

Figure 6.3 gives a more detailed view of the instruction store and control for the DSP. The controller is very simple because all instructions require minimal decoding and consume only a single instruction cycle.

The user's signal processing program is stored in the ROM as a sequence of 50-bit wide instruction words. The wide, horizontal instruction format was chosen to eliminate decoding delays and the loss of control bandwidth which come with vertically encoded instructions. The horizontal instructions also allow more simultaneous actions to be specified, which could...