Design of Testbed and Emulation Tools

S. F. Lundstrom and M. J. Flynn

Technical Report: CSL-86-309

September 1986

This research was sponsored by the NASA-Ames Research Center under contract NAG2-248. The continuing participation of Dr. Lundstrom was supported by MCC (Microelectronics and Computer Technology Corporation).

Design of Testbed and Emulation Tools

by

S. F. Lundstrom and M. J. Flynn

Technical Report CSL-86-309

September 1986

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

Abstract

The research summarized in this report was concerned with the design of testbed and emulation tools suitable to assist in projecting, with reasonable accuracy, the expected performance of highly concurrent computing systems on large, complete applications. Such testbed and emulation tools are intended for the eventual use of those exploring new concurrent system architectures and organizations, either as users or as designers of such systems. While a range of alternatives was considered, a software-based set of hierarchical tools was chosen to provide maximum flexibility, to ease in moving to new computers as technology improves and to take advantage of the inherent reliabiliity and availability of commercially available computing systems.

Key Words and Phrases:

Application Representation, Concurrent Systems, Emulation Tool, Graph Mapping, Interconnection Network, Intermediate Level Analysis, Performance Modeling, Performance Projection, Petri Net, Queueing Network Model, Simulation, Taxonomy, Testbed.

1 Introduction

Highly concurrent systems are expected to make a key contribution to the future of high-speed computation, both for numeric and non-numeric applications. Unfortunately, only a few highly concurrent systems exist and very few actual applications have been developed to run on such systems. Operating systems, languages, compilers and associated run-time environments are not yet mature. Therefore, those who are developing future systems and those who expect to be major users of future highly concurrent systems, have no experience with which to make reasonable projections of the expected performance and efficiency of real applications on proposed new concurrent system organizations. The research reported here, the design of testbed and emulation tools, is part of a larger project whose long-term objective is to project, with reasonable accuracy, the expected performance of highly concurrent computing systems on large, complete applications.

For those interested in the research and development of future, highly concurrent systems, the challenge is especially difficult. The discipline of the normal research cycle (see Figure 1), is to begin with a hypothesis, and then design and conduct an appropriate experiment. The researcher would then observe and measure the results of the experiment. Finally, the experimental results would be evaluated and compared to the hypothesis before starting the cycle over again. However, existing techniques to study the projected performance of large, complex applications on future, highly concurrent systems are tedious, extremely time consuming and inadequate. Consider two important factors, the time to prepare an experiment and the time to perform that experiment.

In computing systems, studies can be performed across a spectrum ranging from one extreme of detailed instruction-level simulation to another extreme of 'cut & try' where the whole system would actually be implemented. Simulation approaches are often chosen in order to reduce the time to evaluation of an experiment. However, as explained in Section 2, such simulations can be as much as six orders of magnitude slower than the actual system being studied. Therefore, some other approach is needed in order to understand the projected performance of concurrent systems.

The performance of an application on a concurrent system is dependent on many factors including

- the structure of the application,
- the language and compiler,
- the run-time environment,
- the operating system, and
- the concurrent computer system itself.

In order to develop a new approach to study of highly concurrent systems, this study first considered how each of these factors might impact the design of testbed and emulation tools. We then began the process which would lead to the design of a set of testbed and emulation tools. This report describes the research approach, the requirements for such a testbed, and the design approach chosen.

2 Research Approach

When considering how to approach the performance projection of complex applications on highly concurrent systems, a number of alternatives can be considered:

- Build System and Try It

  This approach would have the highest fidelity in projecting performance since one would really ***know*** what would happen without any approximations. At the time of this study, this approach seemed most appropriate to the study of one particular system when considerable time and expense are already involved in the implementation of the system. This approach is of no help to those who are in the process of designing their system.

- Detailed Simulation

  The detailed simulation of a system is often proposed for two reasons:

  - speed of implementation
  - flexibility in making changes

  However, consider an example where the new system being studied must execute programs two orders of magnitude (100x) faster than the performance of existing machines. In addition, assume that the memory capacity of the projected new machines are larger than existing systems. In this case, three multiplicative factors contribute to the usage cost of detailed simulation:

  - existing machine-100x slower than planned machine
  - simulation overhead-# simulation instructions 100x more than the number of native instructions of the new machine during execution
  - memory system—existing system must emulate new memory, using bulk memory techniques resulting in a 100x slow-down just from this factor.

  The result is that simulation speed will be 10^6 times slower than the projected performance of the actual system. In other words, a job which would take one hour on the new system, would consume one million hours of simulation time (in excess of 117 **years).**

- Simulate Inner Loops

  Because of the amount of time required to simulate an entire application, separate study of the computationally expensive inner loops would seem to be a reasonable approach. However, if 95% of an application executes at full speed, say at 1.00 GFLOP, and if the remaining 5% executes at 10 MFLOP, then the overall effective rate of execution is only 17Q MFLOP (0.17 GFLOP), a significant reduction. Study of inner loops will help understand the effectiveness of the utilization of the concurrent system during the execution of the loops. Projection of the overall performance of an application based on study of inner loops alone can lead to optimistic performance estimates. For example, in Figure 2, the overall performance of a multiprocessor (with varying numbers of 10 MFLOP processors) is shown as a function of FS, the fraction of code executed in serial. Note that even when the 'non-kernel' code is less than 1%, the system is unable to utilize the potential cycles (FS=O).

- Develop a Mathematical Approximation

  The time required for execution of an application is directly related to the slowest part of the system (the system bottleneck) for a particular application. In practice, the bottleneck may change from time to time during execution of an application (such as a channel, main memory access, and Arithmetic and Logic Unit). A mathematical expression could be written which is a function of the peak performance of these bottleneck units and of a set of parameters which are applications based. Unfortunately, the development of such a formula is not straight-forward and either would need to be verified with another method, or, more likely, might be developed on the basis of previous simulation tests.

- Integrated Hierarchy of Models and Tools

  The most cost effective approach to projecting the performance of highly concurrent systems seems to be a hierarchy of models and tools. In such a hierarchy, the majority of the work can be accomplished at more general levels of models. Parameters at the general levels can then be verified through use of more detailed models and tools, but without requiring execution of all of an application at the most detailed level.

Because the tools of interest in this study are intended to support the performance projection of complex applications on concurrent computer systems, this study first spent some time becoming familiar with the environment in which the tools might be used and with the process by which applications come to be executed on concurrent computer systems. These preliminary efforts led to an understanding of requirements which were then used in approaching the design of the testbed and emulation tools.

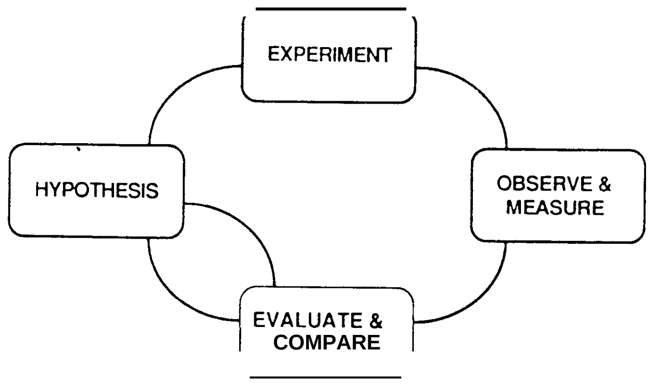


Figure 1: Research Cycle

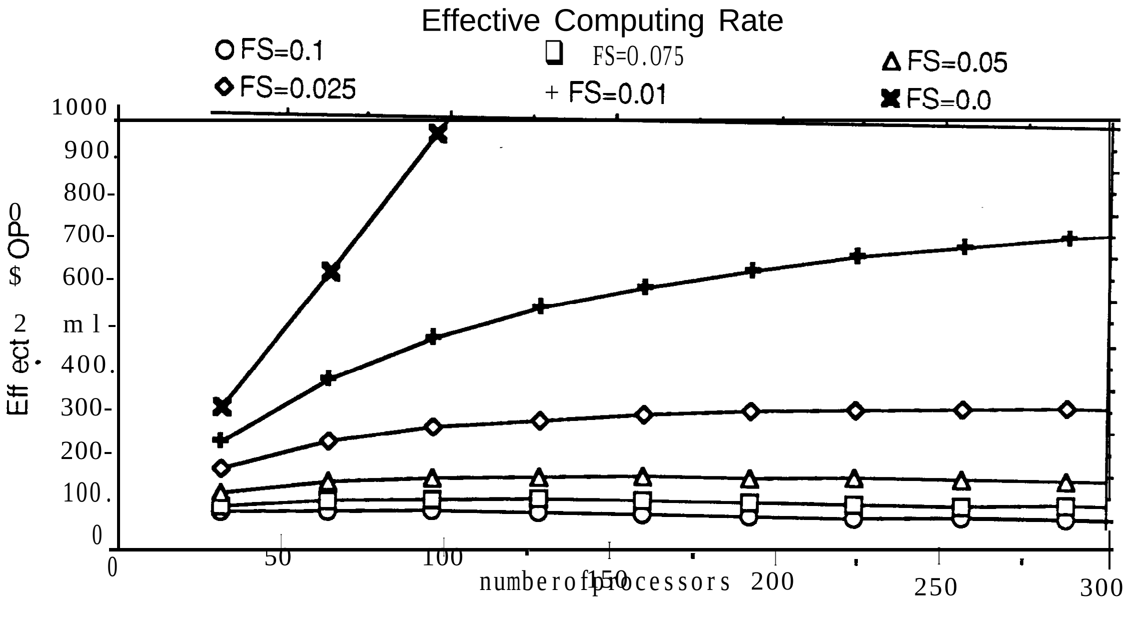


Figure 2: Available Computational Power as a Function of Serial Fraction in Code

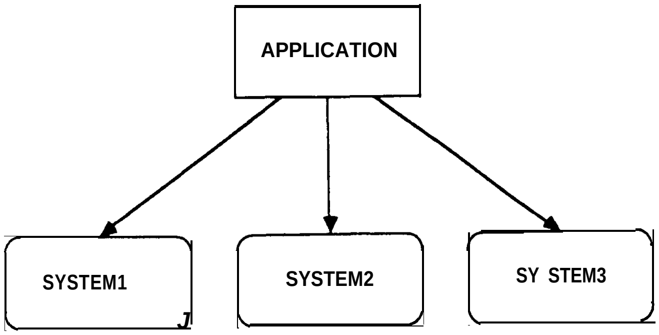


Figure 3: Best Target for Application

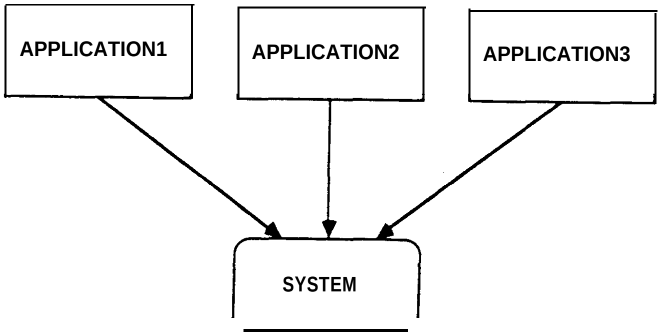


Figure 4: Application Challenges for a System

3 Requirements

3.1 Expected Usage

The testbed and emulation tools are expected to attract two basic types of users; those interested in determining which of a number of concurrent systems would be most effective in executing a particular application of interest (Figure 3), and those interested in determining how well various applications would execute on a particular concurrent system (Figure 4).

This research was concerned with applications as defined by a high-level language source code. Emphasis was placed on FORTRAN because of its common usage in applications already making significant use of parallel computational resources. A framework was established for understanding the application heritage of the source code representations.

The systems considered were those best described as concurrent processors. That is, those systems which might be classified as MIMD (Multiple-Instruction, Multiple-Data) stream systems. SIMD systems, such as vector or pipelined computers, were not considered.

3.2 Accuracy of Results

Since no capabilities to project performance of complex applications existed at the time of this work, the establishment of any bounds on a performance estimate would be a significant step forward. The accuracy of a performance estimate is a complex function of the ability to accurately estimate application needs (itself a function of input data in most cases), of the process of mapping the application onto the system through compilers, linkers, and loaders, of the management of the concurrent system by the operating system, and of the underlying concurrent system itself. Because of these complexities, our objective in making performance projections is to understand something about the accuracy of the projection. Definite bounds would be desirable, but estimates presented as probability distribution functions would also be useful.

3.3 Integrated Hierarchy of Tools

As mentioned in an earlier section, the time required to do a complete instruction-level simulation of a large, complex application can easily be more than six orders of magnitude slower than the time required by the real system, if it were available. While this level of detail would clearly give the most accurate results, if feasible, the testbed and emulation tools were considered at a number of levels of detail. High-level, general system functional models will be capable of full application/system models. By themselves, high-level models will necessarily have limited accuracy. More detailed model levels can be used judiciously in order to balance the benefit of the use of the detailed models with the time demands on the modeling resources. Selected questions raised at the system model level can be studied in more detail in lower level models in an integrated hierarchy of models.

3.4 Level of Detail Expected

As mentioned earlier, this testbed and emulation tool design project is part of a long-term research program to learn how to project the performance of complex applications on highly concurrent machines. The long-term project expects to either:

- make a performance projection within some known bounds, or
- estimate performance with a probability distribution function

Because of the emphasis on performance projection of a whole application, the focus of this design project is to provide tools to study those portions of a system which have the most significant effects on performance. Specific details of application execution are generally not needed in order to meet the overall objectives.

4 Background Studies

Before the design of the testbed and emulation tools could be considered, a review of available technology and of possible general approaches to the design was conducted. The technologies reviewed included existing tools, specialized hardware, and software. The process involved in application execution was reviewed in order to better understand the relationships between different levels of implementation.

4.1 Application Execution Process

Before an application can be executed, it must progress successfully through a number of stages of development, some of which are automated, until a representation of the application exists in a form which can be executed on the system. As an application passes from one level to the next during development, more and more information needed for execution is bound to the application description. If this application mapping process is clearly understood, then the testbed and emulation tools may be able to work at more general, less constrained levels with known relationships to actual execution environments. Figure 5 shows the various level of detail, which are involved in the application development and execution process.

Consider the process of mapping the application onto an execution environment first. The initial approach to a problem is usually to describe it in English. A system analyst then studies the problem and designs an algorithm to solve the problem. A programmer codes the algorithm into some high level language. From this point on, most of the mapping process is automated, first through the use of a compiler which compiles the high level language into an intermediate language. Code optimization typically takes place at this level. The intermediate language code is then translated, by the code generators, to the machine code. The machine code is usually interpreted by a microprogram stored in the control store of the host machine.

Similarly, hardware can be represented by many levels of abstraction. At the most detailed level, the hardware can be defined in terms of register transfers and basic machine instructions. That hardware, together with the microcode stored in its control memory, is a virtual machine which executes assembly language code instructions. That assembly code virtual machine, together with the operating system which manages I/O operations, memory management, and process scheduling, is an intermediate language virtual machine. The intermediate language virtual machine hides a lot of specific details unique to the basic hardware machine. At the highest level, the algorithm designer assumes the existence of a

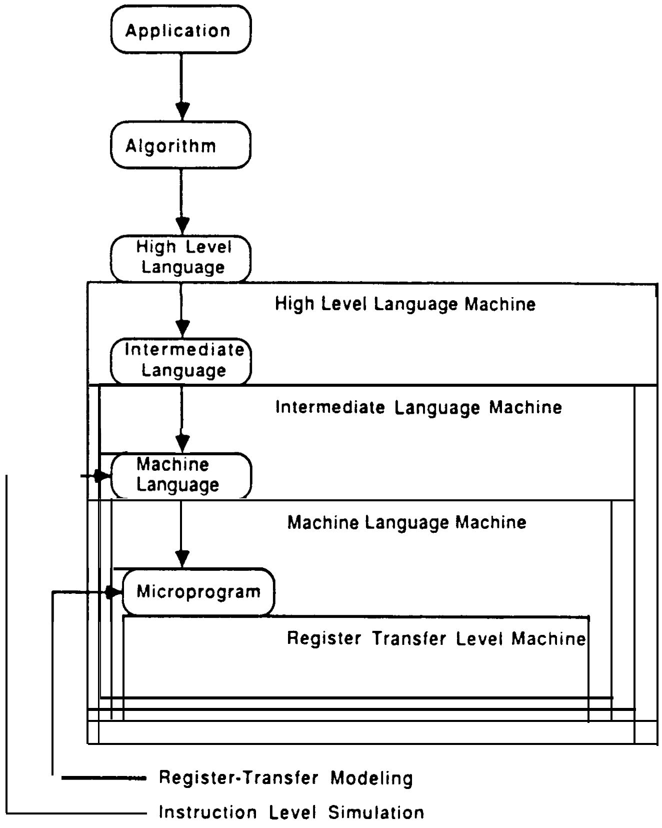

Application
Algorithm
High Level Language
High Level Language Machine
Intermediate Language
Intermediate Language Machine
Machine Language
Machine Language Machine
Microprogram
Register Transfer Level Machine
Register-Transfer Modeling
Instruction Level Simulation

Figure 5: Levels of Application Implementation Detail

high-level language virtual machine which includes the capabilities of optimization of the original high-level language.

The execution of software on some hardware can be modeled at any of these various levels. At the lowest level, the behavior of the registers and basic functional components while executing the microprograms of the underlying machine could be simulated. This level would, obviously, provide the most detailed information. Simulation at the next more general level, known as instruction level simulation, is not concerned with all the details of data movements between registers. The execution model has been abstracted such that the underlying details are hidden and the results of the underlying details are represented in (sometimes an approximate way) the timing characteristics of the machine-language instruction set being simulated.

In a similar manner, intermediate language or high-level language interpretation can be considered. These execution models must also represent the effects of operating system functions, run-time environments, compiler optimization, and many more details. These more general forms of computational models do not require as much computational effort to get results, but they hide a significant amount of information about the detailed execution process.

If modeling and emulation tools are available at each of the levels of this hierarchy, execution assumptions at one level can be validated at the next. For example, a floating point ADD instruction can be simulated in detail at the lowest level to determine the instruction execution time. This timing information can then be used by, higher levels in the hierarchy of tools.

Maintenance of the consistency of information (such as instruction timing) between various levels is very important. Unfortunately, while some simple one-to-one relationships hold between levels in the hierarchy of models, many practical system effects require consideration of more complicated (sometimes nondeterministic) functions describe the relationships between information in models at different levels of abstraction. Operating systems, optimizing compilers, and variable delays in memory systems and interconnection networks are some of the complicating factors. Some of these effects will need to be represented as probability distribution functions at the higher levels in the hierarchy.

4.2 Application Representation and Mapping

One part of the execution process clearly involves mapping an application through the various levels of instantiation to whatever level is being used for performance analysis. In the background studies, two problems were encountered:

- the semantics of the various levels of application instantiation were ambiguous, and
- each level of application instantiation is represented, typically, by a different language.

When trying to discuss these issues, the lack of **a** clear definition for the various levels of abstraction became a problem. Without a clear definition for each level, the identification of any added constraints on the inherent parallelism of a problem as the problem was mapped from the original problem domain to the domain of an executable computer program became very difficult.

The following terminology was developed in order to describe the stages in the solution of a problem, from its original statement through creation of an executable algorithm. An example is used to demonstrate the use of this technology. The example is the determination of the temperature distribution of a conducting surface with various constant-temperature and insulation constraints.

- **Problem** Level **0 A** problem level 0 specification is the highest-level specification of a problem. The PLO specification describes the problem, and contains the knowledge about the problem necessary for its solution.

Problem Level 0

What is the steady-state temperature distribution in the following conducting solid? The upper surfaces are maintained at temperature Tu, the lower surface at Tl, and the sides are semi-insulated. (The physical differential equations are implicitly included at this level.

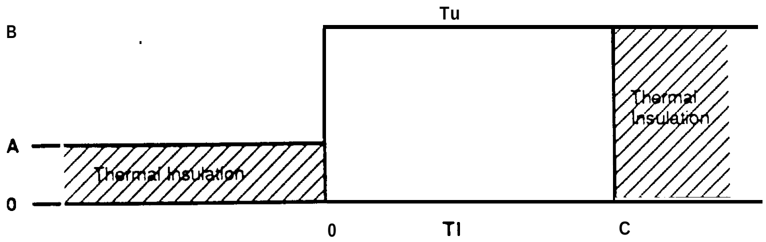


Figure 6: Problem Level 0

For the temperature distribution example (see Figure 6), a PLO specification would consist of a description of the conducting solid and its surfaces, and would pose the question as to what is the steady-state temperature distribution. The knowledge about the physical behavior of the system is included in the form of differential heat equations.

- **Problem Level** 1 A problem level 1 specification is complete enough that an algorithm could be generated from this specification without further knowledge of the problem. This PL1 specification introduces the model used in the solution, and thus is the first level where only an approximation to the actual problem may be found.

  For the temperature distribution example, a problem level 1 specification describes the discrete form of the problem (see Figure 7). This discrete form includes the mesh, the boundary conditions, and the difference equations used at each point of the mesh.

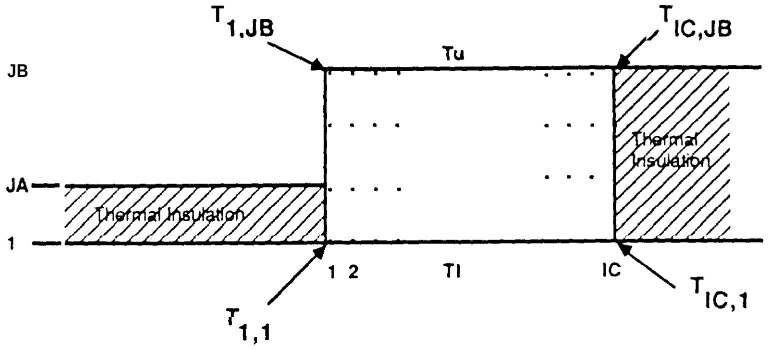
- **Problem Level** 2 **A** problem level 2 specification of the problem includes the PL1 specification, but also includes algorithmic information. The algorithmic information specifies the execution of the model.

  For the temperature distribution example (see Figure 8), this algorithmic information specifies the time-coincident, repetitive application of the difference equations over all points in the mesh, and the termination conditions.

- **Algorithm Level 1** An algorithm level **1** specification is **a** representation of the algorithm to such detail that an execution of the representation would provide the algorithm's solution to the problem. The specification is unconstrained by factors not inherent in the PL2 specification, and represents the detailed algorithm at its most pure level. This AL1 specification is similar in content to the PL2 specification, but typically PL2 will be natural-language, and AL1 will be **a** more rigidly defined form

Problem Level 1

Divide the solid into an array of mesh points. The finite difference forms of the heat equation and boundary conditions relate the temperature at the mesh points for the given problem by the following relationships:



(1) $T_{i,j} = (T_{i,j-1} + T_{i,j+1} + T_{i-1,j} + T_{i+1,j}) / 4$

for $2 \leq i \leq IC-1$ and $2 \leq j \leq JB-1$

(2) $T_{i,j} = (T_{i,j+1} + T_{i,j-1} + 2^*T_{i-1,j}) / 4$

for $2 \leq j \leq JB-1$ and $i = IC$

(3) $T_{i,j} = (T_{i,j-1} + T_{i,j+1} + 2^*T_{i+1,j}) / 4$

for $i = 1$ and $2 \leq j \leq JA$

(4) $T_{i,j} = Tu$

for $1 \leq i \leq IC$ and $j=JB$ OR $i = 1$ and $JA \leq j < JB$

(5) $T_{i,j} = T_L$

for $j = 1$

Figure 7: Problem Level 1

Problem Level 2

Over the mesh given in PL1 , an initial temperature distribution is assigned and is then corrected by repetitive application of equations 1 through 5 from the PL1 description. The program is terminated when either the temperature at each point has been correct N_{max} times or when the absolute value of successive temperature changes is less than ε at every point.

For tests, use $N_{max} = 60$, $\varepsilon = 0.10$, and 10 mesh points in each direction.

Figure 8: Problem Level 2

(e.g. Petri net, Holt diagram, etc.). The execution of any AL1 specification must be, by definition, equivalent to the execution of any other representation.

For the temperature distribution problem, an AL1 specification is a flow chart, or Holt diagram, or Petri net, etc., which specifies a precise execution of the algorithm, but does not constrain the algorithm (see Figure 9).

- **Algorithm Level** 2 An algorithm level 2 specification is a constrained AL1 specification. An AL2 specification has the same result potential as the AL1 specification, but may have artificially introduced constraints in either its representational method (e.g. FORTRAN), or its representational goals (e.g., coding for efficient execution on a specific machine).

  For the temperature distribution **example,** the AL2 specification (see the example in Figure 10) is a FORTRAN, or Pascal, etc., program which has the same algorithmic features of the AL1 specification, but contains extra constraints such as the serialization of the calculations in a FORTRAN program.

The level descriptions above describe the transformation of a problem from its conception to its realization in an executable form. The definition of these levels was valuable in the analysis of where the application-related constraints occur. However, since the objective of the research was to proceed from high-level language source, further definition of these levels was not required at this time.

The relationships between the levels beginning with the high-level language level are important. Given an understanding of how an application maps through the various levels to the detailed register-transfer (RT) level, inverse mappings may be possible to relate performance parameters at higher levels to the RT level. Unfortunately, each of the levels, beginning with the high-level language level (Algorithm Level **2**), are represented in different formal languages. These languages are so varied that important functional relationships between the levels are often obscured by differences in the notation used. As part of the Background Studies, alternative languages and other forms of representation were considered in order to determine whether a common form could be identified for use by all

The following SASL program represents the algorithm.

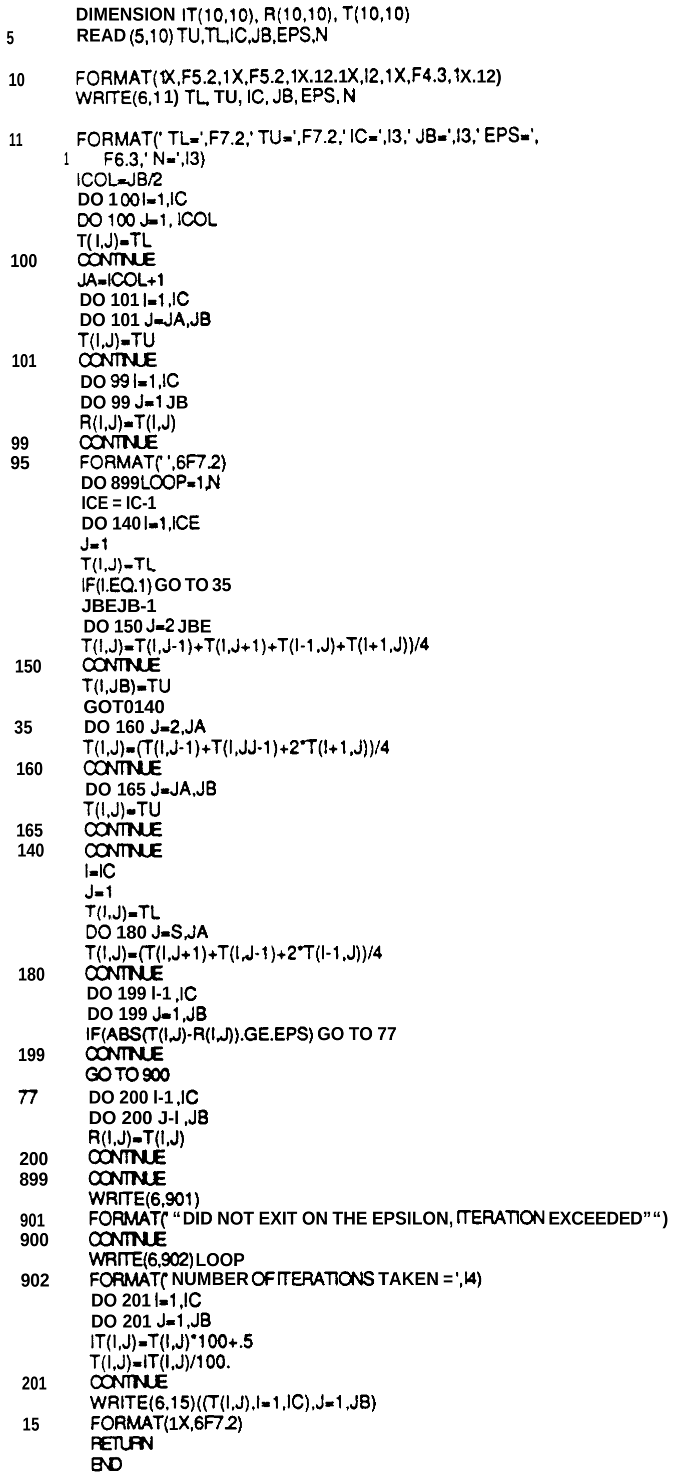
```
temp 0 initial
WHERE

temp time T = time > Nmax -> T : [:(time=",time,%)]
                smallchange T T' -> T' : ["(time=",time+1,%)]
               temp (time + 1) T'
             WHERE
              T' = next T
              smallchange T1 T2
                = all [(T1 i j - T2 i j) c epsilon;
                 j <- 1..JB; i <- 1..IC]

next T = [[f a b; b <- 1..JB]; a <- 1..IC]
        WHERE
         f i j = equation1 i j
                     -> (T i (j-l) + T i (j+1) + T (i-l) j + T (i+l) j)/4
               equation2 i j
                    -> (T i (j+1) + T i (j-l) + 2 * T (i-l) j)/4
               equation3 i j
                    -> (T i (j-l) + T i (j+l) + 2 . T (i+l ) j)/4
               equation4 i j
               -> Tupper
               equation5 i j
               -> Tlower
        equation1 i j
         = 2 <= i and i <= (IC-1) and 2<= j and j <= (JB-1)
        equation2 i j
         = 2 <= j and j <= (JB-1) and i = IC
        equation3 i j
        = i = 1 and 2 <= j and j <= JB
        equation4 i j
        = 1 <= i and i <= IC and j = JB or i = 1 and JA <= j and j < JB
        equation 5 i j ,
        = j = 1

Nmax = 60
epsilon = 0.10
IC = 10
JB = 10
Tupper = 100
Tlower = 0
initial  =  [[  0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0],,,,,,
                [ 0,0,0,0,0,0,0,0,0,0],
                [ 0,0,0,0,0,0,0,0,0,0]]
```

Figure 9: Algorithm Level 1 [in SASL]



```
          DIMENSION IT(10,10), R(10,10), T(10,10)
5         READ (5,10) TU,TL,IC,JB,EPS,N

10        FORMAT(1X,F5.2,1X,F5.2,1X.12.1X,I2,1X,F4.3,1X.12)
          WRITE(6,11) TL, TU, IC, JB, EPS, N

11        FORMAT(' TL=',F7.2,' TU=',F7.2,' IC=',I3,' JB=',I3,' EPS=',
       1     F6.3,' N=',I3)
          ICOL=JB/2
          DO 100I=1,IC
          DO 100 J=1, ICOL
          T(I,J)=TL
100       CONTINUE
          JA=ICOL+1
          DO 101 I=1,IC
          DO 101 J=JA,JB
          T(I,J)=TU
101       CONTINUE
          DO 99 I=1,IC
          DO 99 J=1 JB
          R(I,J)=T(I,J)
99        CONTINUE
95        FORMAT(' ',6F7.2)
          DO 899LOOP=1,N
          ICE = IC-1
          DO 140 I=1,ICE
          J=1
          T(I,J)-TL
          IF(I.EQ.1) GO TO 35
          JBEJB-1
          DO 150 J=2 JBE
          T(I,J)=T(I,J-1)+T(I,J+1)+T(I-1,J)+T(I+1,J))/4
150       CONTINUE
          T(I,JB)=TU
          GOT0140
35        DO 160 J=2,JA
          T(I,J)=(T(I,J-1)+T(I,JJ-1)+2*T(I+1,J))/4
160       CONTINUE
          DO 165 J=JA,JB
          T(I,J)=TU
165       CONTINUE
140       CONTINUE
          I=IC
          J=1
          T(I,J)=TL
          DO 180 J=S,JA
          T(I,J)=(T(I,J+1)+T(I,J-1)+2*T(I-1,J))/4
180       CONTINUE
          DO 199 I-1 ,IC
          DO 199 J=1,JB
          IF(ABS(T(I,J)-R(I,J)).GE.EPS) GO TO 77
199       CONTINUE
          GO TO 900
77        DO 200 I-1 ,IC
          DO 200 J-I ,JB
          R(I,J)=T(I,J)
200       CONTINUE
899       CONTINUE
          WRITE(6,901)
901       FORMAT( "DID NOT EXIT ON THE EPSILON, ITERATION EXCEEDED" ")
900       CONTINUE
          WRITE(6,902) LOOP
902       FORMAT(' NUMBER OF ITERATIONS TAKEN =',I4)
          DO 201 I=1,IC
          DO 201 J=1,JB
          IT(I,J)=T(I,J)*100+.5
          T(I,J)=IT(I,J)/100.
201       CONTINUE
          WRITE(6,15)((T(I,J),I=1,IC),J=1,JB)
15        FORMAT(1X,6F7.2)
          RETURN
          END
```

Figure 10: Algorithm Level 2 [in FORTRAN]

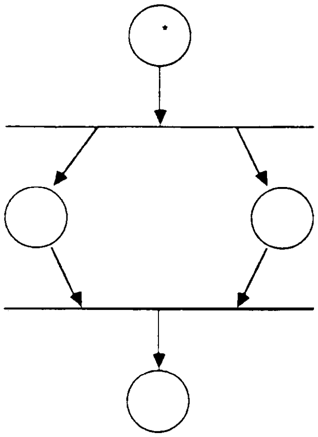


Figure 11: A Simple Petri Net

levels in the hierarchy. Some of the alternatives considered as a common, level-independent representation form were:

- Petri Nets
- Role-Activity Diagrams
- SASL

Petri Nets are graphical representations of execution. A Petri Net has two types of nodes: ***place*** and ***transition.*** Edges in a Petri Net connect only nodes of opposite type. In Figure 11, the ***places*** are represented by circles and the ***transitions*** are represented by horizontal lines.

In addition, a notion of ***activation*** is used to understand the evaluation sequence in a Petri Net. When a ***place*** holds a value, a ***token*** is shown in the ***place*** (see top circle in the figure for example). When every input ***place*** of a ***transition*** has at least one token, the ***transition*** is ***enabled to fire.*** The process of ***firing*** a ***transition,*** removes a token from each input ***place*** and adds a token to each output ***place*** of the firing ***transition.*** Petri Nets are quite general and are a powerful notion in modeling parallel structures and asynchronous activities. However, they become quite complex when representing execution of a complicated application executing on a concurrent system.

Petri Nets can be analyzed with queueing models. In this case, a ***transition*** would represent a service station in the queueing model. A ***place*** would represent a queue. **A** token would represent a task. The number of tokens in a ***place*** would be the queue length. The ***transition*** firing time would be the service time in the queueing model.

Petri Nets could also be analyzed as a Markov process since the next state depends only on the previous state. Because of a property of conservation of tokens at a ***place*** within Petri

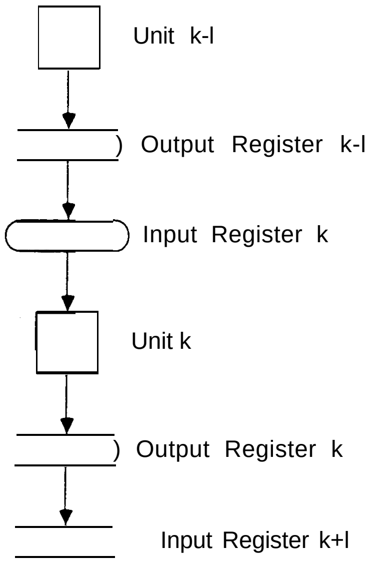


Figure 12: Pipelined Asynchronous Control Unit

Nets, notions such as bottleneck, peakload, max utilization, waiting time, queue length, etc. are straightforward to study.

Petri Nets show both control and data flow, and as such relate well to notions of simulation. The interpretation of token, transition, and place are flexible. In addition, Petri Nets themselves are hierarchical (as an example below demonstrates).

Hardware Modeling As an example, **a** simple asynchronous control unit for a pipelined computer could be represented by a Petri Net Model. Figure 12 shows the general organization of the hardware with various functional units, each of which has an input register and an output register. Figure 13 shows the corresponding Petri Net model of this asynchronous control unit. Notice the tokens residing simultaneously in various ***places*** in the net.

Software Modeling As a second example, consider the temperature distribution example given earlier. Recall that the overall problem was shown in Figure 6 and that a Fortran version of it was shown in Figure 10. Figure 14 shows the basic blocks in that Fortran program. Figure 15 shows the Petri Net representation of that program flow. In order to show more detail, a portion of the representation shown in Figure 15 is expanded in Figure 16, thus demonstrating the potential of hierarchy within this sort of a model.

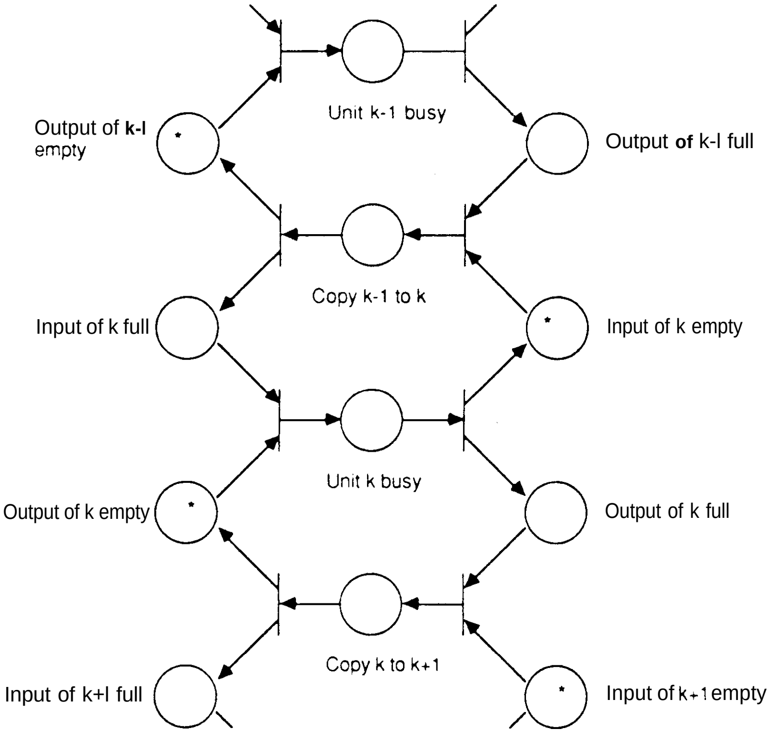


Figure 13: Petri Net Model of Asynchronous Control Unit

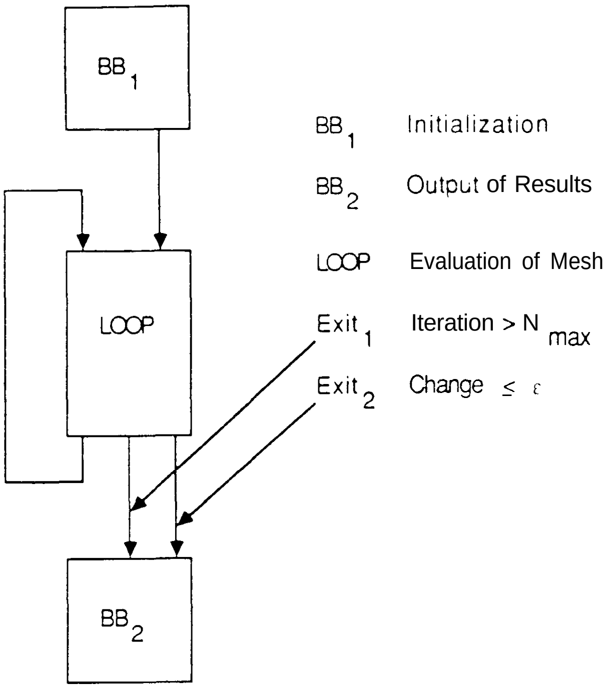


Figure 14: Basic Blocks of Temperature Distribution Problem

4.2.1 Role-Activity Diagrams

Role-Activity diagrams are Petri nets which show the organization of systems. The method used was invented by Anatol Holt. Roles are sequences of activities in which actors participate when they are playing that role. An activity often requires a number of actors in their roles for its realization.

Any given role is played by, at most, one actor at a time, although different actors may play the role at different times. Also, an actor, playing a role, may participate in only one activity at a time. For an activity to occur, all roles associated with that activity must have actors taking part. Thus, each role associated with that activity is a resource necessary for the realization of that activity.

Role-Activity diagrams, represented as role-activity nets, are a method of specifying use of resources in a system. For computer algorithms, the resources compute the values in the solution of the problem. Thus, Role-Activity diagrams specify the interaction of values in the algorithm, as well as the interaction between resources.

A vertical strip in a Role-Activity diagram represents the succession of roles taken on by an actor (a resource) over time. Time is assumed to progress down the page. Activities requiring multiple actors are represented by horizontally-connected squares under each actor required. Figure 17 is an abbreviated directory which can be used to interpret the con-

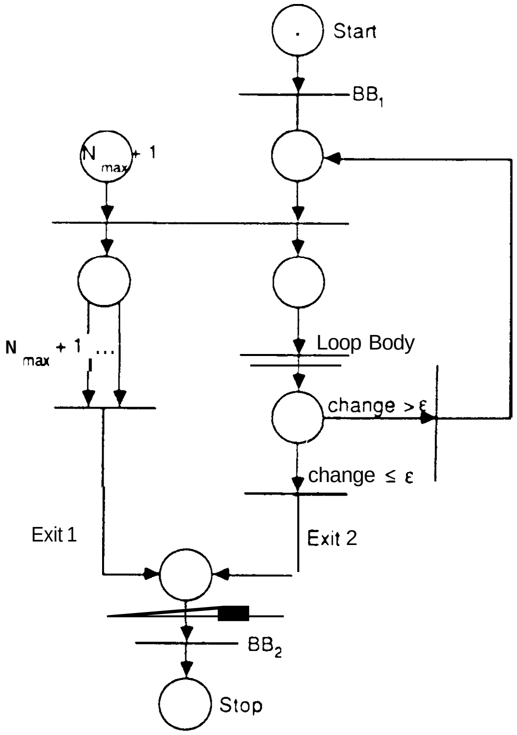


Figure 15: Petri Net Model of Temperature Distribution Problem

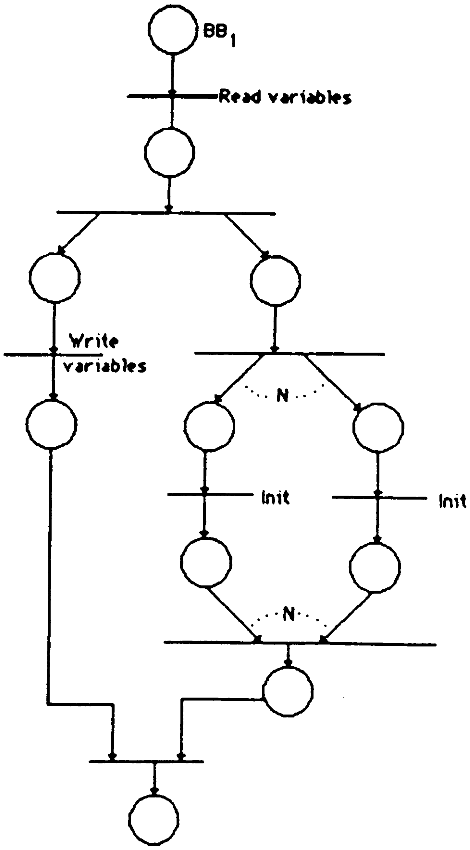


Figure 16: Petri Net Expansion of BB_1

stituents of a Role-Activity diagram. Figure 18 shows a portion of a Role-Activity diagram. The figure shows the update of a cell within the temperature distribution problem.

Role-Activity diagrams are very general, although because of the explicit representation of time, they are more constrained than Petri Nets. The graph structure has few artificial restrictions. However, these diagrams are not a perfect representation for this project. Generalizations such as looping and recursion are not handled directly in Holt diagrams, and are often difficult to create and/or spot. Because the production of each value must be represented by one role, a program with a lot of values (such as large arrays) would take up a lot of diagram space. Not only is this kind of graph difficult to draw, it is also difficult to look at because of its large size. Large graphs are also tedious to change. Many problems, especially in simulation and modeling, have very regular characteristics. In these problems, the Role-Activity diagrams have similar regularity. We have identified the production of Role-Activity diagrams from a higher-level specification, the 'generative problem.' Some study of this has revealed some relatively simple extensions to Role-Activity

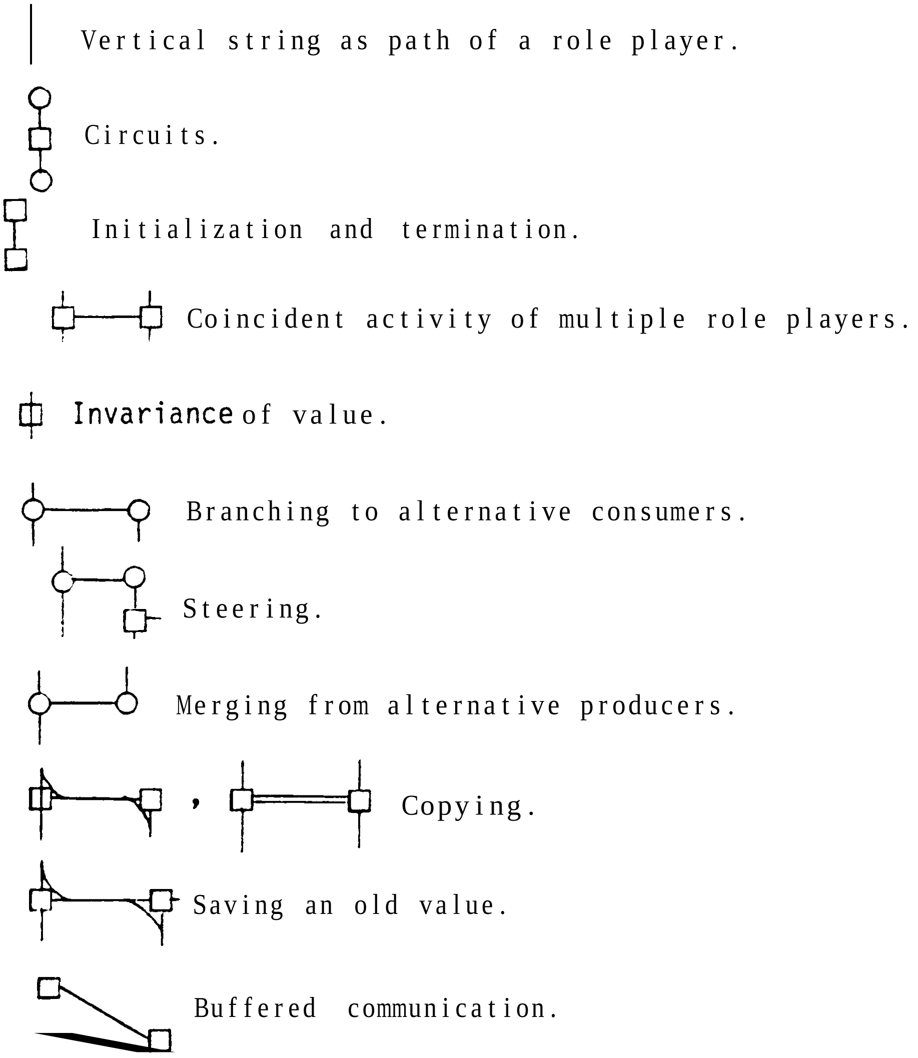

Vertical string as path of a role player.
Circuits.
Initialization and termination.
Coincident activity of multiple role players.
Invariance of value.
Branching to alternative consumers.
Steering.
Merging from alternative producers.
, Copying.
Saving an old value.
Buffered communication.

Figure 17: Abbreviated Directory for Role-Activity Diagram Interpretation

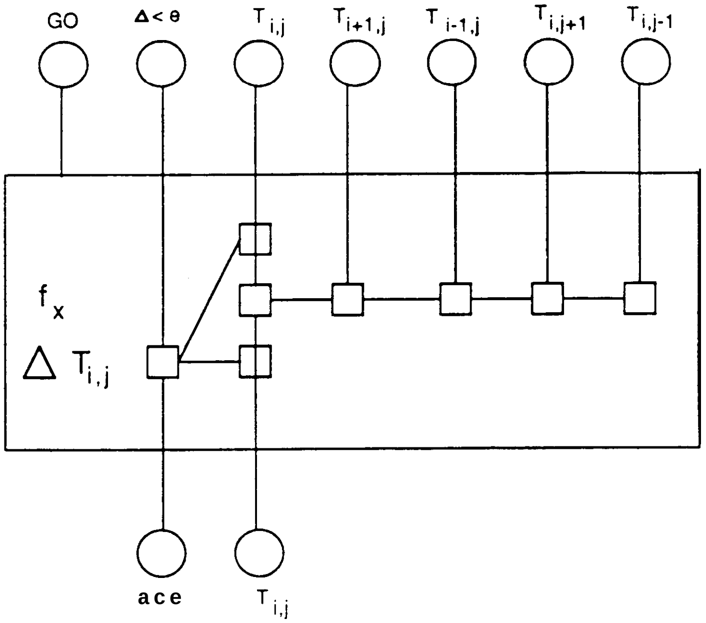


Figure **18:** Portion of Role-Activity Diagram—Temperature Distribution Problem

diagrams which provide the desired characteristics. A major advantage of Role-Activity diagrams for these applications is the ease and visibility with which artificial constraints, such as those imposed by certain compilers, or languages, may have been added to the **AL1** representation. In addition, even constraints added by the machine (such as queueing at memory) can be easily represented with Role-Activity diagrams.

Role-Activity diagrams can easily represent the execution of a program on a concurrent system since the various system execution resources are the ***actors*** and the computations to be performed are ***activities.*** Communications between actors also are explicitly shown. However, these diagrams become quite complex when trying to represent dynamic resource allocation.

4.2.2 SASL Programs

SASL is a language developed by David Turner at the University of St. Andrews (hence SASL: St. Andrews Static Language). **SASL** is an applicative language, where the program is an expression consisting solely of the application of functions to arguments. Assignment is impossible; within a context, a variable's value never varies. Substitutions and reductions in the representation may occur, but the value stays the same. Parallelism in SASL is possible because there are no side effects (there is not assignment). So, 4+1 is always equal to 3+2 in SASL; the objects are different, but the value is the same.

SASL's only data structures are lists, which may be arbitrarily complex. The elements of these lists need not be all of the same type. Lists may also be of infinite length, as long as all of the elements are not evaluated. For example,

```
ones = [1] ++ ones
```

is an infinite list of 1s.

A function is also a value in SASL, and all functions have only one argument. By a procedure known as Currying, a function of one argument returns a function which takes another argument, etc., so that many arguments can be used. For example,

`PLUS 3 4` is really `((PLUS 3) 4)`,

where the function `(PLUS 3)` takes the argument **4.**

Functions in SASL are non-strict. That is, they use lazy evaluation of their arguments so that a function can still have a defined value if not all of its arguments are defined.

SASL programs are expressively complete, but it is sometimes difficult to think of the proper way to express a concept such that no restriction of the **AL1** parallelism occurs. Certain concepts cannot even be stated in SASL. For example, non-deterministic problems such as chaotic-relaxation algorithms cannot be represented. Some extension to SASL would have to be created in order to represent such a concept. Despite these drawbacks, SASL would probably be a good AL1 representation tool, especially if some form of state representation could be added. (The temperature distribution example given earlier used SASL in just this way—see Figure 9.) Problem representation in SASL is lucid and concise. However, there are no inherent means of inserting artificial constraints, so that un-adorned SASL cannot be used as a representation in the testbed at both **AL1** and AL2 levels.

4.3 Survey of Concurrent Architectures

As part of the Background Study, the potential range of machines which might be studied with the planned Testbed and Emulation Tools was studied by surveying a wide range of existing machines. These machines can be characterized from different viewpoints:

- models of computation,
- interconnection network,
- processing element,
- memory system, and
- application

The purpose of this survey was to understand the breadth of each of these viewpoints, at least with current architectures.

4.3.1 Taxonomy of Concurrent Architectures

There are literally hundreds of concurrent architectures proposed and designed in the last decade. These architectures are so different that the usual classifications as SIMD and MIMD [Fly 72] machines are not sufficient.

New taxonomy schemes are developed by many researchers in order to catagorize these different architectures into different classes. One particular taxonomy as reported by Haynes and others in [HLS 82] divided the wide spectrum of concurrent architectures into six classes:

1. Multiple special-purpose functional units
2. Associative processors
3. Array processors
4. Data flow processors
5. Functional programming language processors
6. Multiple CPUs

Machines with multiple special-purpose functional units are usually designed to perform some specific tasks efficiently. One example is the systolic arrays which will be described in more detail in the next section. Computation intensive problems, in which the kernels are based on a number of basic mathematical operations, have found great success in these structures. Matrix multiplication, solution of linear systems, and FFT are some examples.

Associative processors are those machines which utilize an associative memory. In associative memory, one bit of any memory word is available on one access, thus it is possible to search the whole memory simultaneously for specified contents by iteration on bit slices. This organization also allows memory words to be addressed by their contents instead of their addresses. One example of associative processors that we will describe in the next section is STARAN designed and built by Goodyear Aerospace Corporation.

Array processors are machines with multiple arithmetic units operating in lockstep and performing the same operation on different data. This is the most common and popular type of concurrent architecture available on the market. They are particularly suitable to problems that involve a large proportion of array data types. Five examples will be studied in the next section: Illiac IV, BSP, MPP, CHiP, and NON-VON.

Data flow computers are very different from the conventional von Neumann architectures in which a program counter is used to schedule the next instruction to be executed. An instruction in a data flow computer is ready for execution when its operands arrive. As a consequence of this data-activated property, a very high level of concurrency can be exploited. The Data Driven Signal Processor (DDSP) will be given as an example of this type of concurrent architecture.

Functional programming (FP) language machine or reduction machine has gained considerable interest recently. The main advantage of FP is that when algorithms are described in such applicative languages, much parallelism can occur automatically-with no analysis of program structure and without explicit programmer involvement with parallelism. SERFRE will be studied as an example in the next section.

Multiple processors belong to the class normally called MIMD. They are more flexible than the classes described above; however, their control is much more complex. The interconnection network, which connects the processors, usually forms a crucial part both in the design and operation of each machine.

Since the architectures in this class are so diverse that we are going to look at nine different machines in the next section: Cedar, FMP, **S-l,** Cm*, HEP, Empress, MP/C, Ultra, and TRAC.

4.3.2 Architectures Studied

In this section, we will describe **18** different machines, this is by no means a complete survey of all concurrent architectures, we are just trying to cover as wide a spectrum as possible. Using the taxonomy described in the previous section as **a** guideline, we have chosen machines that belong to these six different classes. Since array processors (SIMD) and multiple processors (MIMD) have received most attention, most machines that have been chosen belong to these two classes. In each subsection that follows, we will try to describe the most significant parts of each machine's architecture.

Systolic Array The systolic architectural concept [Kun 82] was developed by Kung and associates at Carnegie-Melon University, and is a general methodology for mapping high-level computations into hardware structure. A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the boundary cells.

In Figure **19,** the hex-connected systolic array can be used to multiply two N x N band matrices of bandwidths W_1 and W2, each of which performs the inner product operation C t C + A $*$ B. The entire multiplication requires only 3N + Min(W_1, W_2) time units. As the matrices shift into the array, they always move in exactly the same direction and

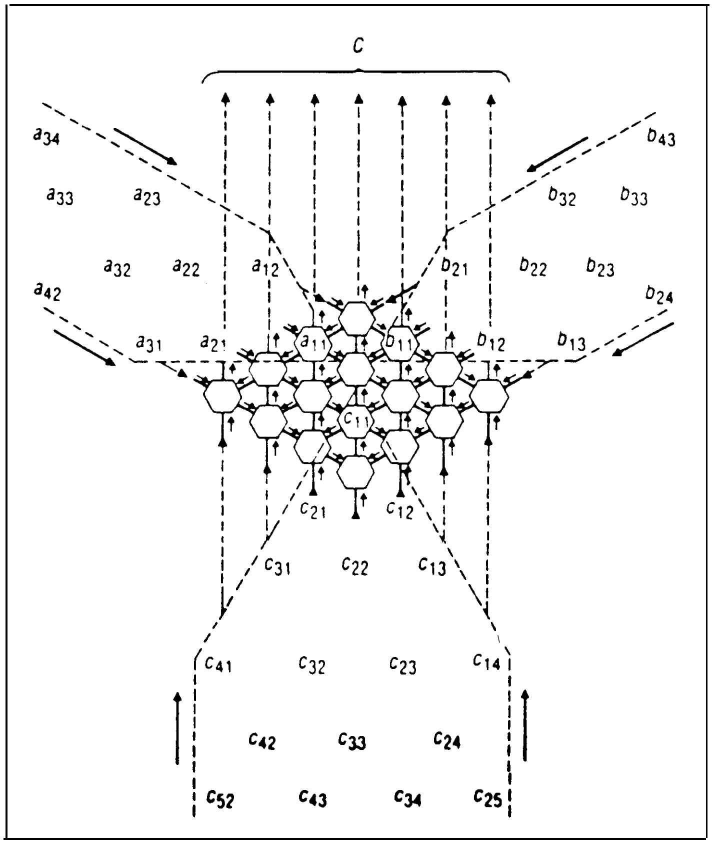


Figure **19:** Hex-connected systolic array

require no control. Each cell performs one computation at each step, and input and output are overlapped with computation. For each I/O access, there are multiple computations performed on the data item, thus execution of compute-bound problems can be speeded up without increasing the I/O requirements. This is a very significant improvement over the classical von Neumann architecture in which the memory access time is associated with each operation of the data item.

For specialized algorithms that can be implemented by the systolic array, they are fast, hardware efficient, and require no software control in communication and synchronization. The major problem with a systolic array is still in its I/O barrier. Implementation of the systolic array on a VLSI chip is limited by the number of pins, or I/O terminal, available on a single chip.

STARAN STARAN [Bat 74], depicted in Figure 20, is the first bit-serial parallel processing system. It was developed by Goodyear Aerospace Corporation in 1972. It consists of up

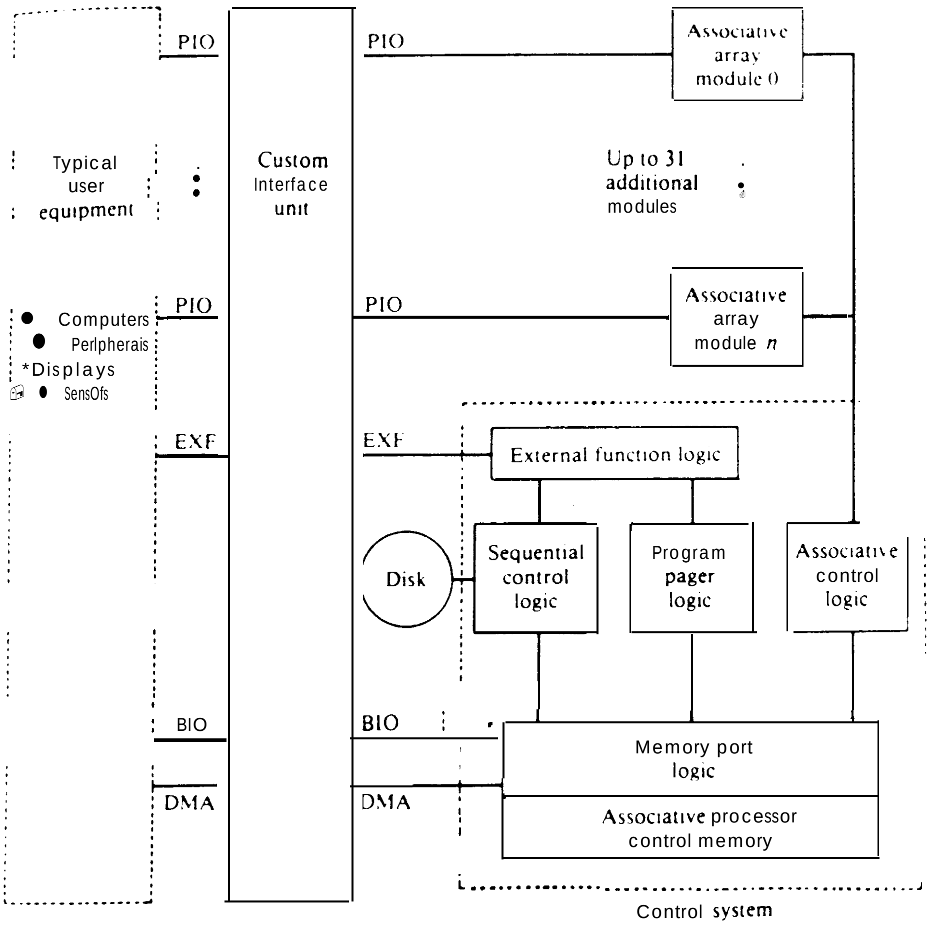


Figure 20: The STARAN System Architecture

to 32 associative array modules, each contains 256 processing elements, a 256-word 256-bit multidimensional access (MDA) memory, a flip network, and a selector. Each processing element operates serially bit by bit on the data in all MDA memory words. The MDA memory can be addressed in either bit-slice (one bit of all 256 words) or word-slice (all bits of one word).

Thus, data can be input and output in the usual word by word fashion while processing can be done in bit-serial fashion. The flip network is used for data shifting or manipulation to enable parallel search, arithmetic or logical operations among words of the MDA memory.

STARAN has high-speed input-output capabilities and the ability to interface easily with conventional computers which handle the tasks that must be processed in a single sequential data stream. The main application areas of STARAN are in signal processing and database.

Illiac IV Illiac IV [BDM 72] was developed at the University of Illinois in the 1960s and fabricated by the Burroughs Corporation in 1972. The original design had 4 quadrants of 64 mesh-connected processing elements under the supervision of 4 control units. Due to cost escalation and schedule delays, only 1 quadrant (see Figure 21) was ever built. The speed of the 64-PE quadrant is approximately 200 million operations per second. The control unit controls and decodes the instruction stream and broadcasts instructions and common data to all PEs. It is also a scalar processor by itself besides having the ability to control the PE-array.

Each PE is a powerful computing unit, and has a 64-bit wide routing path to its four neighbors. The main application area is in scientific applications like numerical weather forecasting and nuclear engineering research.

BSP The Burroughs Scientific Processor (BSP) [KS 82] (see Figure 22) was an attempt by Burroughs Corporation to improve on the Illiac IV design. It has 16 arithmetic elements and 17 (prime number) memory modules interconnected by two alignment networks: full cross-bar switch with broadcasting and conflict resolving ability. This permits general-purpose interconnectivity between the arithmetic array and the memory-storage modules. It is the combined function of the memory-storage scheme and the alignment networks that supports the conflict-free capabilities of the parallel memory. The parallel processors perform vector computation with a clock period of 160 ns. The control processor provides the supervisory interface to the system manager in addition to controlling the parallel processor. The scalar processor processes all operating system and user-program instructions which are stored in the control memory. It executes some serial or scalar portions of user programs with a clock rate of 12 MHz and is able to perform up to 1.5 megaflops. The BSP is capable of executing up to 50 megaflops and is used mainly for scientific applications.

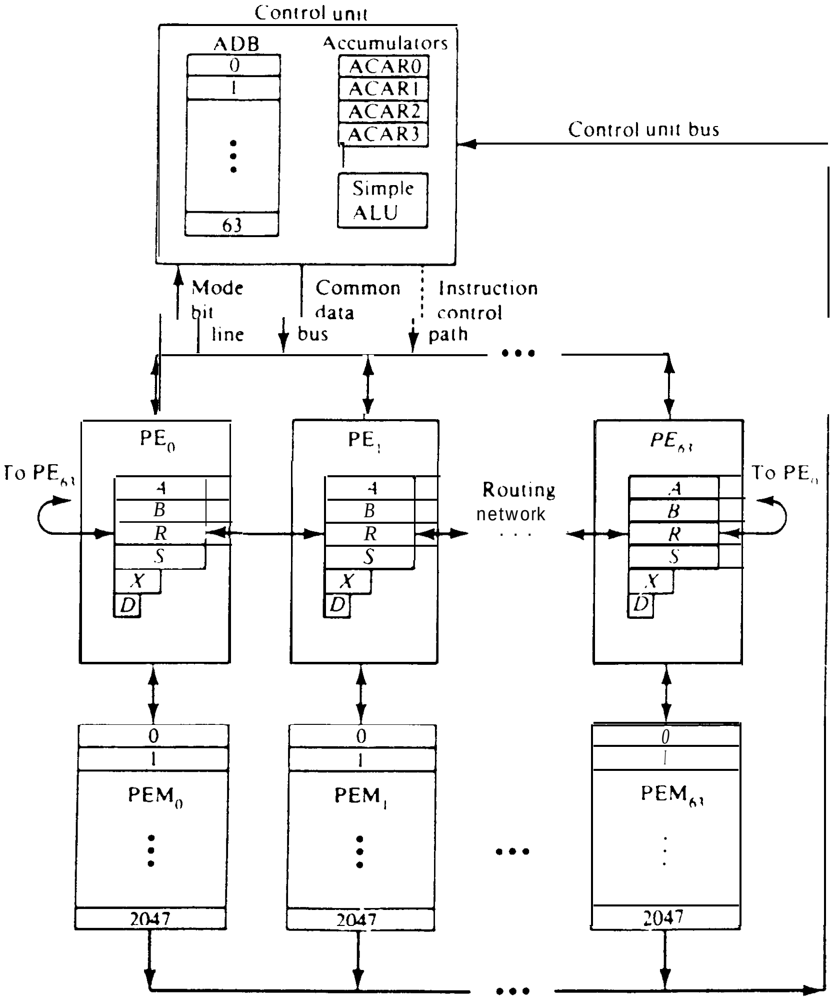


Figure 21: A 64-PE Illiac IV Array

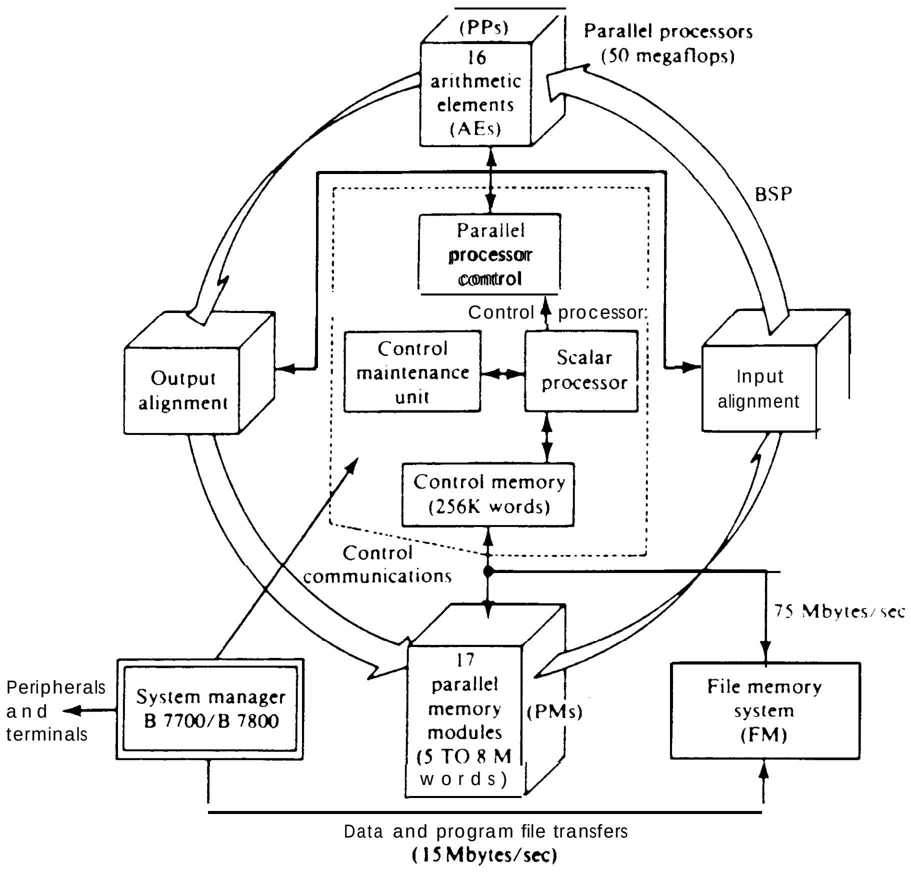


Figure 22: Functional Structure of BSP

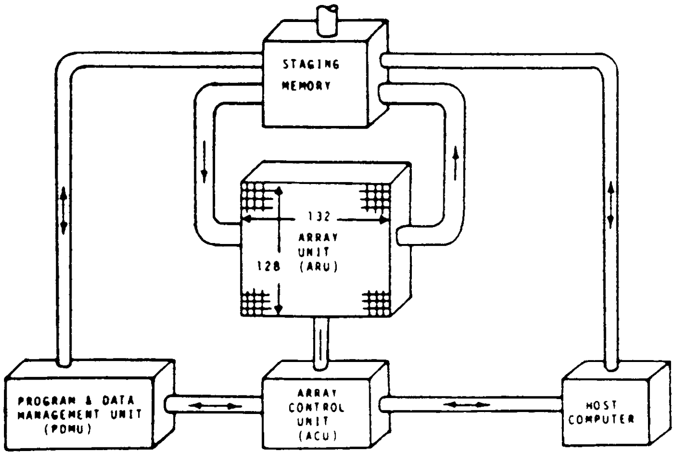


Figure 23: Block Diagram of the MPP

MPP Like STARAN, Massively Parallel Processor (MPP) [Bat 82] was also designed and built by Goodyear Aerospace Corporation starting from 1979 to be a high speed satellite image processing system. The processor, shown in Figure 23, has 16,896 bit-serial processing elements (PE's) arranged in a 128-row by 132-row (4 redundant rows for fault tolerance) rectangular array with strictly nearest-neighbor connections. The edge connection is programmable so that the array may look like a plane, a cylinder, a torus, a spiral, or a linear string. On 32-bit floating-point data, addition occurs at 430 MOPS and multiplication at 216 MOPS. The staging memory in the input-output path of the array unit acts both as a buffer between the array unit and the outside world, and also to reformat data so both the array unit (bit-serial) and the outside world (word-serial) can transfer data in the optimum format. MPP is a SIMD machine and all PE's perform the same instruction on every machine clock cycle. Although built for satellite imagery processing, preliminary application studies indicate that MPP can also support general image processing, weather simulation, aerodynamic studies, radar processing, reactor diffusion analysis, and computer image generation.

CHiP The CHiP computer, shown in Figure 24, [Sny 82] is a family of architectures each constructed from three components: a collection of homogeneous microprocessors (28 to 216), a switch lattice, and a controller. The microprocessors are not directly connected to each other, but rather are connected at regular intervals to the switch lattice. Each switch in the lattice contains local memory capable of storing several configuration settings and thus be changed dynamically during program execution: mesh for dynamic programming; hexagonally connected mesh for LU decomposition; torus for transitive closure; tree for sorting; double tree for searching; etc.. The perimeter switches are connected to external storage devices. The controller is responsible for loading the switch memory. CHiP processing begins with the controller broadcasting a command to all switches to invoke a particular configuration setting. Individual microprocessors then synchronously execute the instructions stored in their local memory.

By integrating programmable switches with the processing elements, the CHiP computer achieves a polymorphism of interconnection structure that also preserves locality, thus allowing algorithms that exploit different interconnection patterns to be used in the same program. CHiP can be viewed as a configurable systolic array: it has all the advantages of the systolic array while it is still general enough to embed different interconnection patterns in its lattice.

NON-VON The **NON-VON** architecture, shown in Figure 25, consists of two parts: primary processing subsystem and secondary processing subsystem. The primary processing subsystem is organized as a binary tree of small processing elements (SPEs) which have no stored program and can only execute instructions sent by its ancestor nodes. The SPEs in the first few levels of the tree are each connected to a large processing element (LPE) which

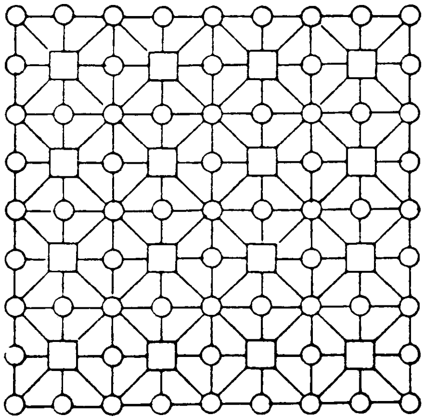


Figure 24: CHiP lattice. PEs shown as squares, switches as circles.

has locally stored program and may operate independently. Thus, NON-VON can act as a single SIMD machine with the node at the root being the ancestor of all the nodes below it, or as a multiple SIMD machine with each subtree controlled by a node connected to an LPE. The LPE connected to the root is called the control processor and is also connected to the host processor.

The secondary processing subsystem consists of 64 to 256 disk-drives each connected via an intelligent head unit to an LPE. These intelligent head units perform certain computationally simple operations (e.g. selection) on the fly, thus added to the processing power of the whole system.

NON-VON is designed to be used mainly in the areas of relational database, sorting and vision.

DDSP The Data Driven Signal Processor (DDSP) [HNI 82] is being developed by ESL Incorporated to be a programmable, modular, high-speed data flow computer primarily for signal processing applications. A block diagram is shown in Figure 26. Its configuration ranges from one to 32 processors with a maximum performance of 71 MFLOPS. DDSP is designed with a high order language (Data Driven Programming Language, or DDPL) capable of generating efficient machine code, and follows the single assignment rule. It implements a dynamic tagged data flow model where tokens are tagged with a label field determined at run-time. The processors in a DDSP system are closely coupled through an interconnection network. A processor consists of an input queue for temporarily saving tokens, a matching store (associative memory) for associating pairs of tokens, and a processing element for performing high speed integer and floating point computations (2.2 MFLOPS). Because of the nature of signal processing computations, the interconnection network is essentially a linear arrangement of processors with wrap-around from the last pair of processors to the first pair, and augmented by a three level tree used for long distance communication. Besides

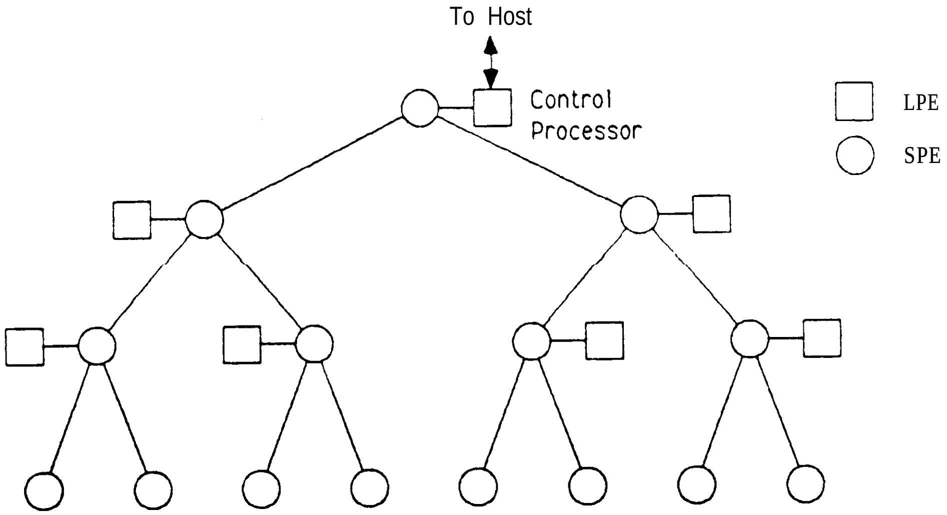
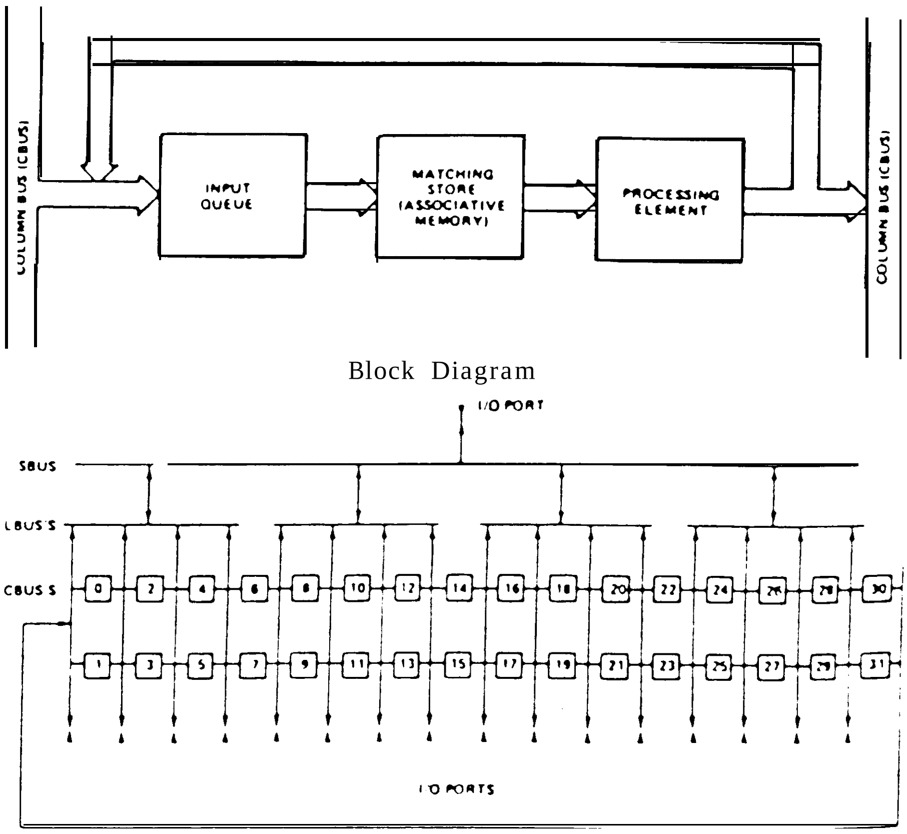


Figure 25: NON-VON Primary Processing Subsystem

signal processing, DDSP can also be used in fields of sonar and image processing.

SERFRE SERFRE [Vil 82] is a multi-processor command-driven (string reduction) machine and can directly executes a FP (Functional Programming) language, trying to have subprograms executed on different processors. It is a dynamic loosely-coupled system using direct communication with storage of messages. Figure 27 shows the architecture of a possible, single-user implementation of SERFRE. Figure 28 shows the structure of a module. The I/O processor controls the memory system as well as the initiation of a program evaluation and returning of the result to the user. The C-processors have their own local memory to store data and function definitions. A C-processor consists of a register for the return address, a stack for the program, registers for the data, and a reduction engine. When asked to evaluate a function involving concurrency, it will try to call for other non-busy C-processors to execute the subprograms, if none is available, it will evaluate them sequentially.

Cedar The objective of the Cedar project [GKL 83] at University of Illinois is to investigate ways to accommodate several thousands of high performance processors to deliver



Block Diagram

Interconnection Network

Figure 26: DDSP

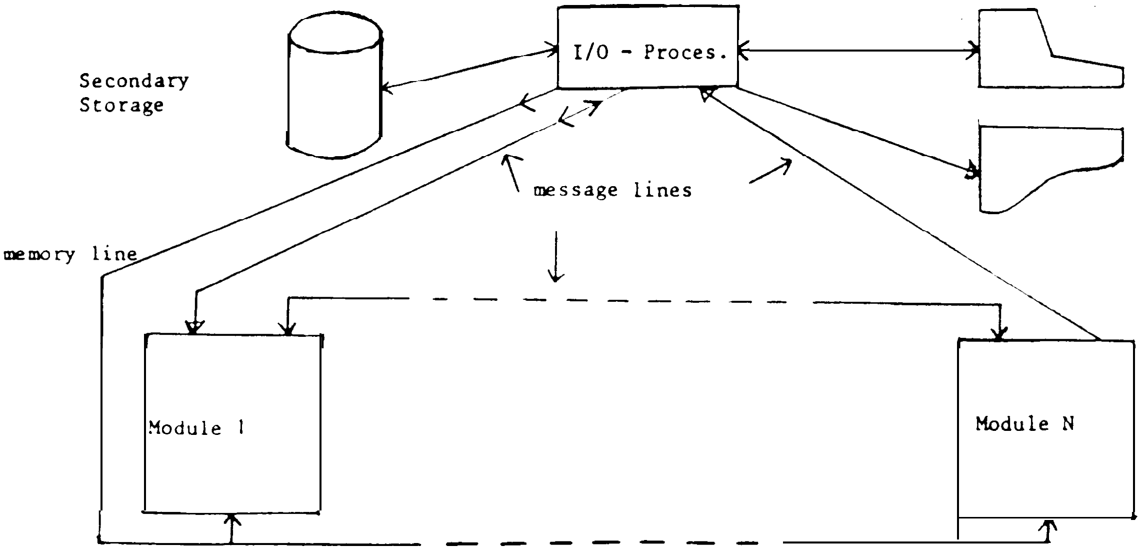


Figure 27: Architecture of the SERFRE

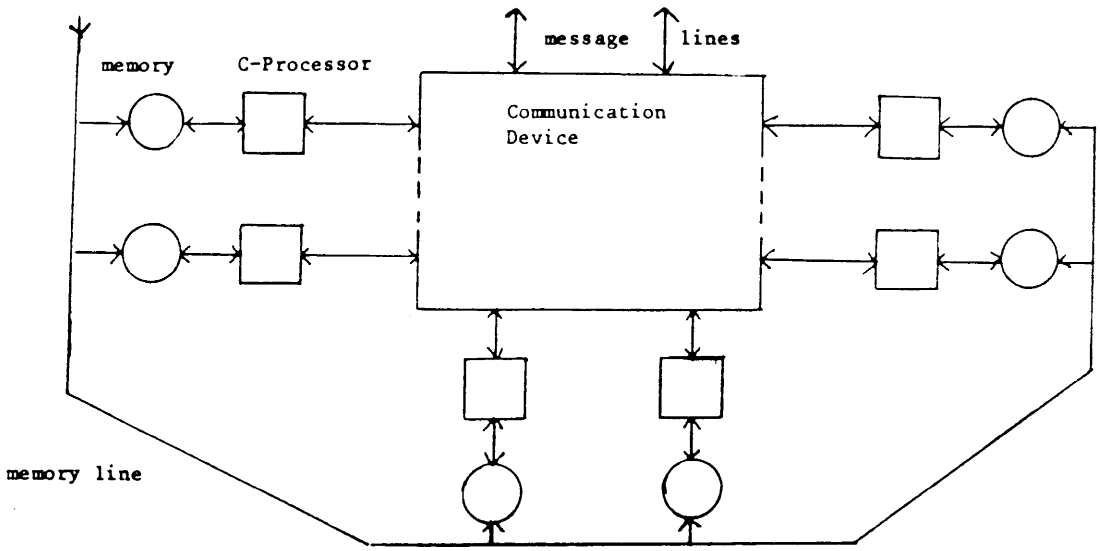


Figure 28: Structure of a Module

several gigaflops. It will make use of the VLSI technology to build powerful VLSI processors, for instance, 32-bit, 2.5 MFLOPS. The uniqueness of this architecture is the concept of Macro Dataflow which combines the control mechanisms of data flow architecture and storage management of the von Neumann machine. A program is viewed as a flow graph of nodes. Each node is either computational (CPF) or control (CTF). The Global Control Unit executes CTF while the processor clusters execute CPF. A processor cluster consists of a number of processors and local memory modules working cooperatively to execute a CPF. When a CPF is finished, the cluster control unit will signal the Global Control Unit so that other nodes depending on this CPF can be scheduled to be executed. Besides local memory, processor clusters can also access the global memory through the global network, an Omega network. Figure 29 shows a block diagram of the structure of Cedar.

FMP The Flow Model Processor (FMP) [Lun 85] was the result of a series of design studies conducted from 1975 through 1982, sponsored both by Burroughs Corporation and by the NASA Ames Research Center. Its objective was to sustain throughput in excess of 1 GFLOP, and was intended to support large scientific problems especially modeling problems in computational aerodynamics. It was designed to support standard FORTRAN, with extended feature like **DOALL,** in which codes within the body of this construct is executed once for each value specified in the definition of the **DOALL** domain.

The conceptual design consists of 128 processor connected through a Connection Network (CN) to the Extended Memory. The Global Code Memory and the Data Base Memory can also be accessed through the CN. The CN, a form of Omega Network, is a circuit-switching network with decentralized control. The Processor Control and Maintenance

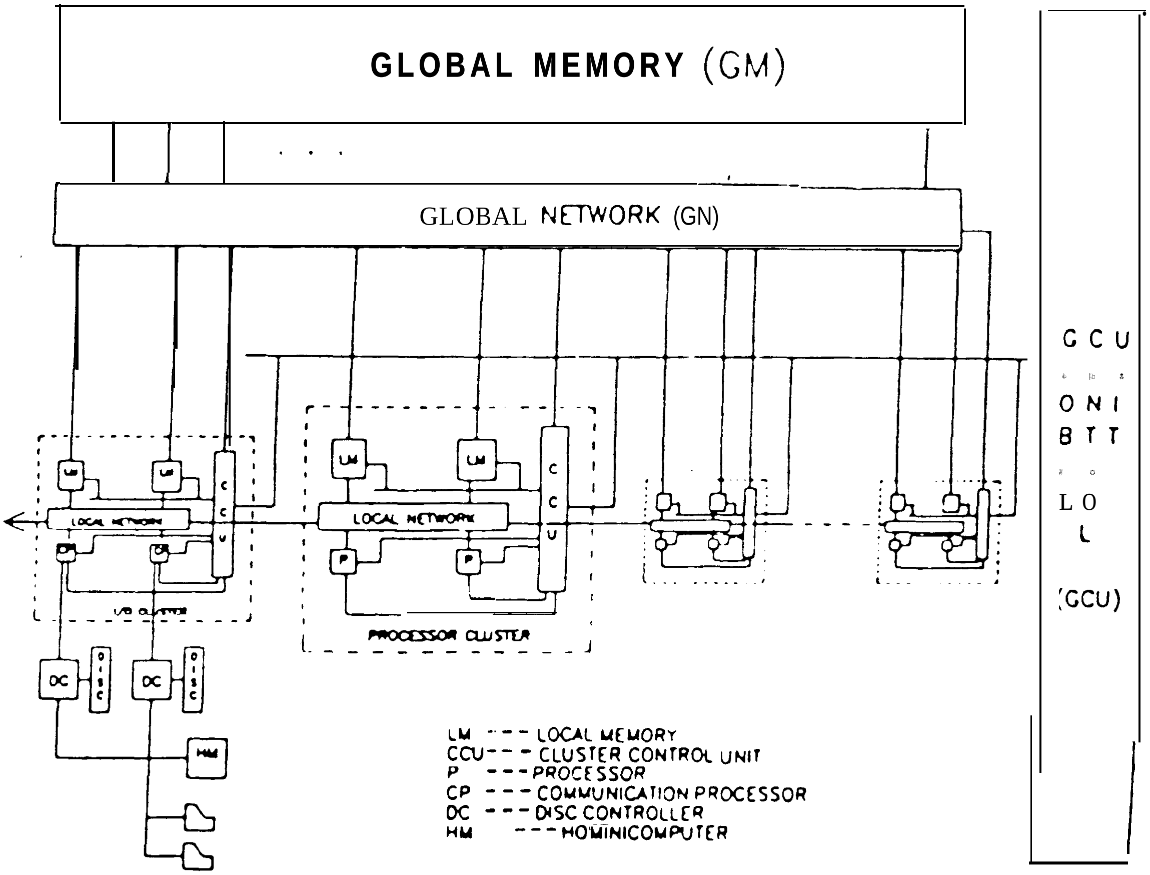


Figure 29: Structure of Cedar

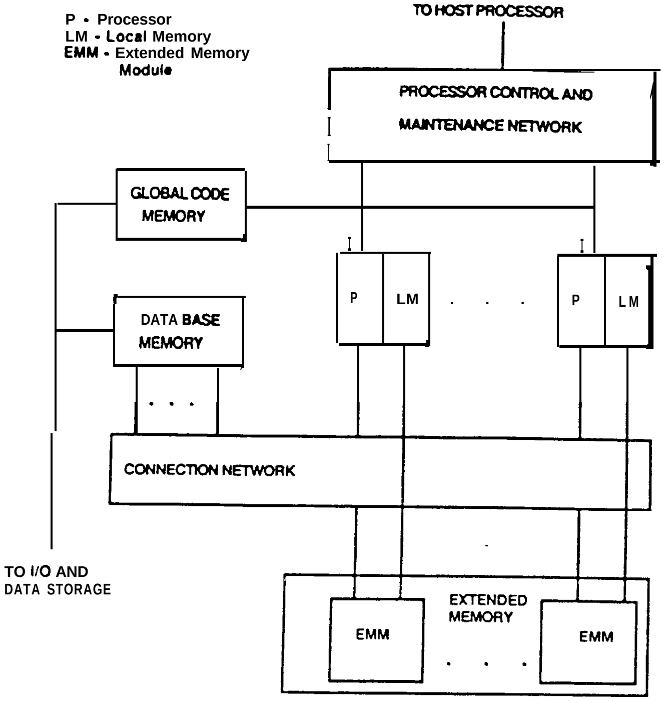


Figure 30: Flow Model Processor Conceptual Design

Network acts as a tree of AND gates to be used to assist in the high speed synchronization at the end of the **DOALL.** Each processor in the FMP is a powerful computing unit. A scheme similar to that of the IBM 360/91 [Tom 67] was used to allow multiple functional units to be used efficiently. A block diagram is shown in Figure 30.

S-l The S-l project [WC 79] has as its general goal the development of advanced digital processing technology for potential application throughout the U.S. Navy. The S-l multiprocessor is designed to be at least 10 times the computing power of the Cray-1. Its architecture (see Figure 31) consists of 16 independent, identical uniprocessors sharing a main memory of 16 modules, each of 1 billion bytes of semiconductor memory. Each uniprocessor is a powerful computing unit with performance comparable to the Cray-1, and can execute instructions independent of others. **A** full Crossbar Switch is used as the interconnection network between the processors and the main memory. A maximum peak bandwidth of more than 10 billion bits per second can be achieved when all 16 channels of the Crossbar Switch are transferring data simultaneously. To further reduce the main memory access time, each member uniprocessor contains private cache memories (data and instructions). As many as eight peripheral processors can be attached to each uniprocessor to handle I/O. The synchronization box is a shared bus connected to each member uniprocessor; one of its major functions is to transmit interrupts and small data packets from one uniprocessor to

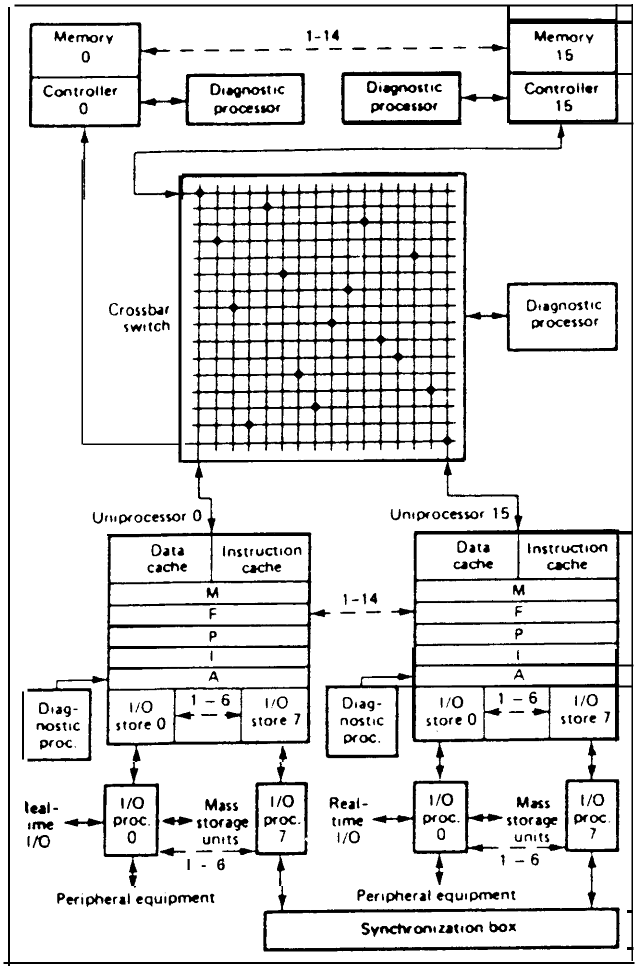


Figure 31: The S-l Multiprocessor

any subset of other uniprocessors in order to coordinate processing streams.

Cm* Cm* [SFS 77] is an experimental computer system designed and built at Carnegie-Mellon University. It is intended to be a testbed for exploring a number of research questions concerning multiprocessor systems. Cm* is a hierarchical and modular system, the basic building block is a processor memory pair called a computer module or Cm. Up to 14 Cm's are connected into a cluster. Each cluster has a shared address mapping and routing processor, Kmap, which allows communication with other clusters through the intercluster buses. A simple S-cluster system is depicted in Figure 32. Communication along the intercluster buses is done in packet switching mode to avoid deadlock over bus allocation. The processor is a DEC LSI-11. All processors share a single segmented virtual memory address space of 2^{28} byte. Each processor has a local memory of 64 Kbyte and is also

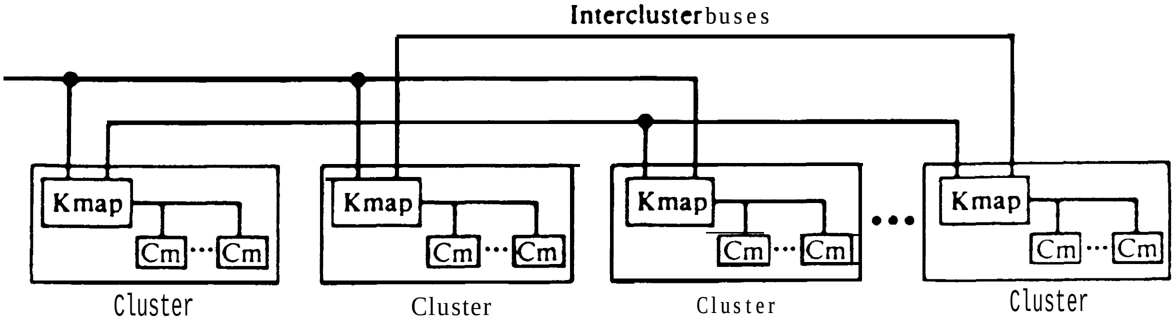


Figure 32: **A** simple S-cluster Cm* system

part of the shared memory in the system. Efficient use of the system depends on ensuring that most of the code and the data references made by a processor are held locally to that processor. Inter-process communication is by message-passing and can be easily built on top of the Cm* architecture.

HEP The HEP computer system [Smi 78] is an MIMD machine of the shared resource type. **A** typical system is shown in Figure 33. In this type of organization, skeleton processors compete for execution resources in either space or time. Two queues are used to time-multiplex the process states. One of these provides input to a pipelined instruction execution unit, which will decode and execute the instruction. For data memory access, the process state enters a second queue. This queue provides input to **a** pipelined switch which interconnects several data memory modules with several processors. Each processor of HEP can support up to 128 processes. Maximum throughput of 10^7 instructions per second per processor occurs when there are at least eight totally independent processes in each processor.

HEP instructions and data words are 64 bits wide. **A** domain of protection in HEP is called a task, and consists of a set of processes which are allowed to communicate with each other. Processes in different tasks or processors may only communicate via data memory if they have an overlapping allocation there. Any register or data memory location can be used to synchronize two processes in a producer-consumer fashion. Three states are provided: reserved, full, and empty. The execution of an instruction tests the states of locations and modifies them in an indivisible manner.

The interconnection switch consists of a number of nodes connected via ports. Messages are sent in packets and routed by the nodes according to their priorities.

Empress The ETH-Multiprocessor Empress [BBB 82] **was** built in order to study the performance of MIMD architectures in general, and particularly in the field of simulation

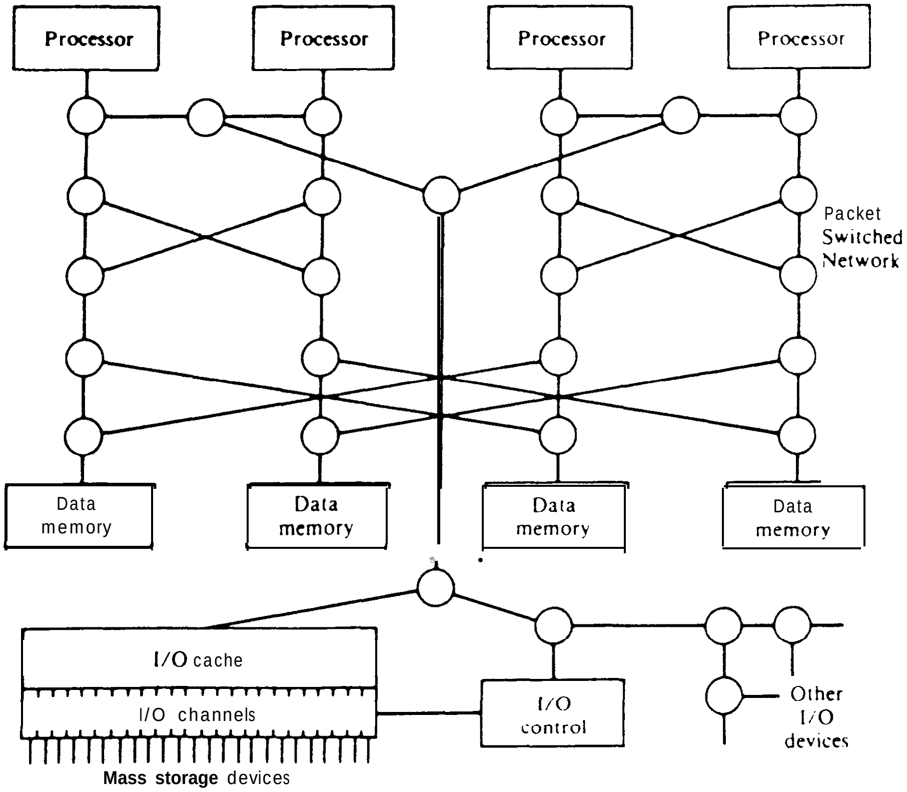


Figure 33: A Typical HEP System

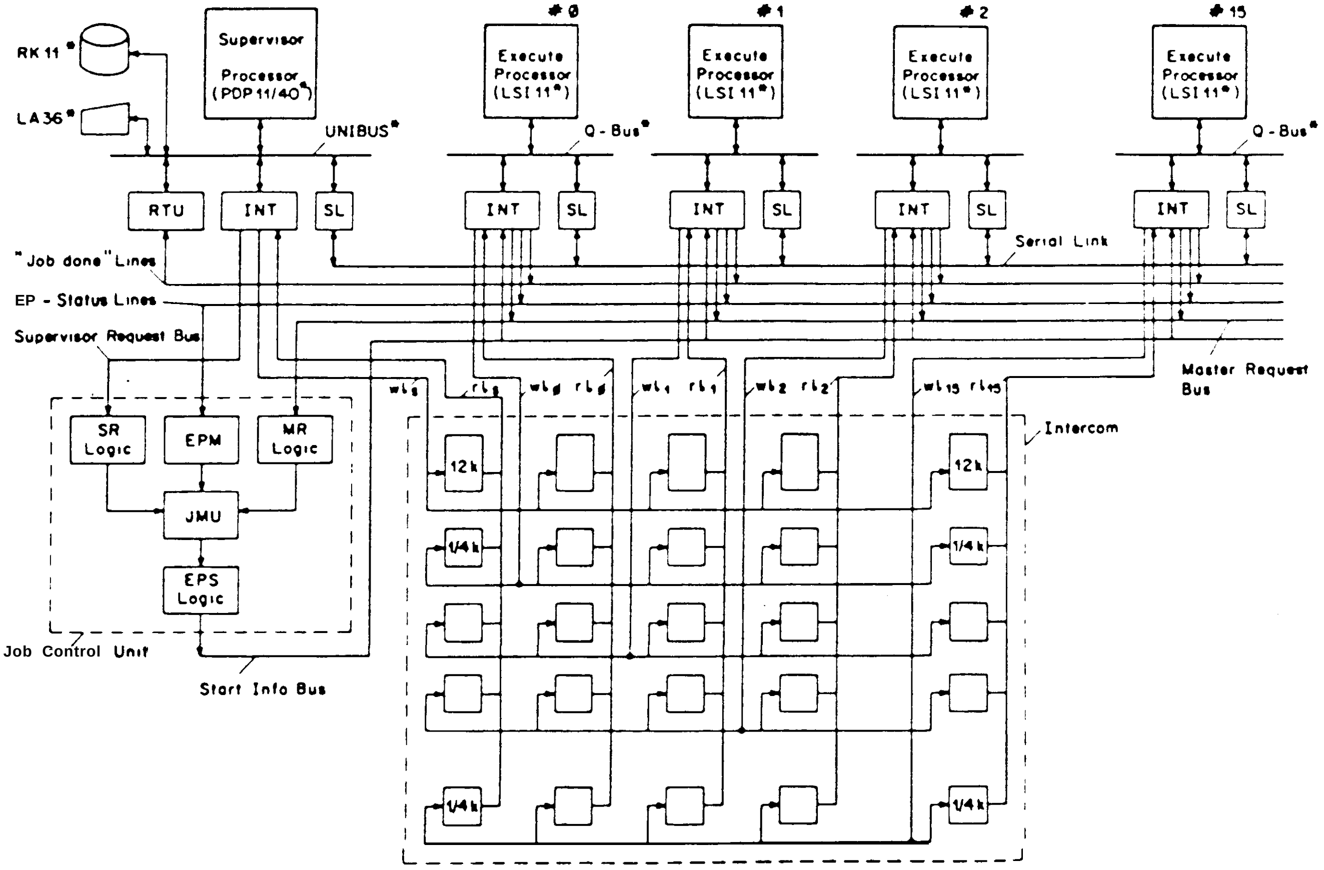
problems. Its architecture consists of a supervisor processor and a number of execute processors, all communicating through an Intercommunication System, Intercom (see Figure 34). The supervisor computer is used to partition a problem into executable jobs which will be dispatched by the job control unit to the execute processors. If an execute processor (master) finds its job exhibit inherent parallelism, it can dynamically request more (slave) processors from the job control unit to form a cooperative group. All I/O, precompilation, optimization and code generation as well as the integration step control are done in the supervisor processor.

The Intercom consists of a quadratic organized memory-matrix in which each processor writes to all blocks in its row and can read blocks from its column. Data duplication within the intercom is only executed into the matrix elements of processors working on the same job. In this way, a result provided by any of the processor is made imrnediately available to all other cooperating processors. Different logical addressing methods are allowed in the Intercom so that cooperating processors may appear to be neighbors although they may be physically apart.

MP/ C The Multiprocessor/computer (MP/C) [AG 82], a dynamically partitioned system, has the shared memory aspect of tightly coupled multiprocessor systems and also the connection simplicity associated with message-connected, loosely-coupled multicomputer systems. It is proposed as a candidate for the effective execution of process-structured algorithms.

Its architecture consists of a number of processor and memory modules, all connected to a system bus. Process fork and join operations are implemented by bus switching as a means of partitioning and recombination of the address space. The bus can be opened between **any** two adjacent processor-memory pairs. Only the leftmost processor in each connected bus segment or partition is active, and can access all memory modules in that partition. All other processors in that partition are inactive. An active processor can activate an inactive processor by splitting the bus segment. Conversely, an active processor may deactivate itself by reconnecting its partition to the one on the left. This ability to partition and reconnect dynamically is best suited to execute tree algorithms, divide-and-conquer algorithms, and database functions. A linear MP/C is shown in Figure 35. Multi-dimensional MP/C machines in which each row or column is a switchable bus, are also proposed.

Ultra The NYU Ultracomputer [GGK 83] is a shared-memory MIMD parallel machine composed of thousands of autonomous processing elements (PE's). By the use of an enhanced message switching network with the geometry of an Omega-network, it can approximate the ideal behavior of Schwartz's paracomputer model of computation which permits every PE to read or write a shared memory cell in one cycle. The Omega-network also implements the fetch-and-add operation used as the synchronization primitive.



*)		product of Digital Equipment Corporation	MR	:	Master Request
RTU	:	Result Transfer Unit	JMU	:	Job Mongement Unit
INT	:	Intercom Interface	EPS	:	Eucutr Processor Start
SL	:	Serial Link Interface	wl	:	write lines
SR	:	Supervisor Request	rl	:	read lines
EPM	:	Execute Processor Monitoring	k	:	1024 words / 16 bits

Figure 34: Empress Hardware

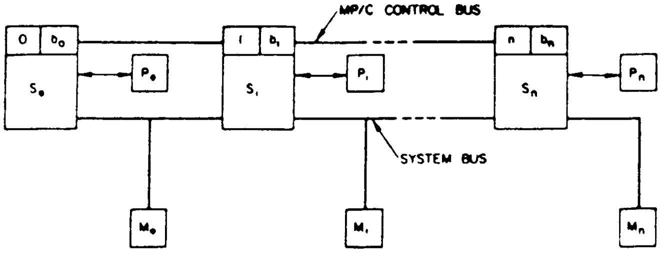


Figure 35: The Linear MP/C

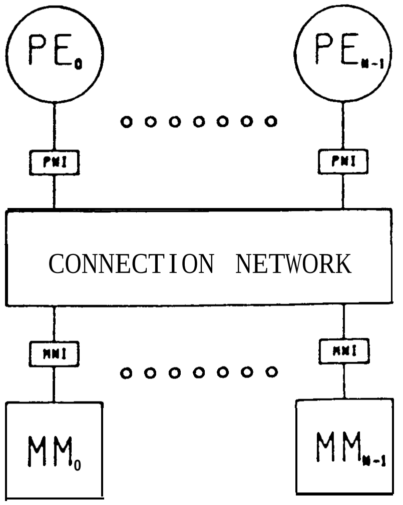


Figure 36: Block Diagram of the NYU Ultracomputer

Its architecture consists of thousands of PE's connected through a connection network to thousands of memory modules (see Figure 36). Each PE is a high-speed VLSI floating point processor. It can also support the fetch-and-add operation: a PE will continue execution of the instruction stream immediately after issuing a request to fetch a value from central memory, the target register would be marked locked until the requested value is returned from memory; an attempt to use a locked register would suspend execution. The connection network is an enhanced message switching Omega-network. Each switch in the network has a queue and an internal adder to support the fetch-and-add operation. Simultaneous accesses to a common memory cell can be detected in the switch and be combined to a single fetch-and-add instruction. The memory unit also has an adder to implement the fetch-and-add instruction.

TRAC The Texas Reconfigurable Array Computer (TRAC) [SUK 80] is an experimental computer system built at the University of Texas at Austin. The uniqueness and the potential capabilities of TRAC arise from its interconnection network; a dynamically reconfigurable banyan network (see Figure 37). The banyan network serves to partition and to configure the processor, memory and I/O resources of the system into different architectural organizations under software control. Within a partition, TRAC is varistructured in that regardless of the data structure requirements for the task, any data width or architecture may be used. Independent or interacting tasks can all be running simultaneously on the same computer. The machine is also virtual in that user programs can be oblivious of the specific set of memory and processor modules used.

Inside the SW-banyan network, the nodes can be configured to form three types of subtree: data trees, instruction trees, and shared memory trees. Besides shared memory,

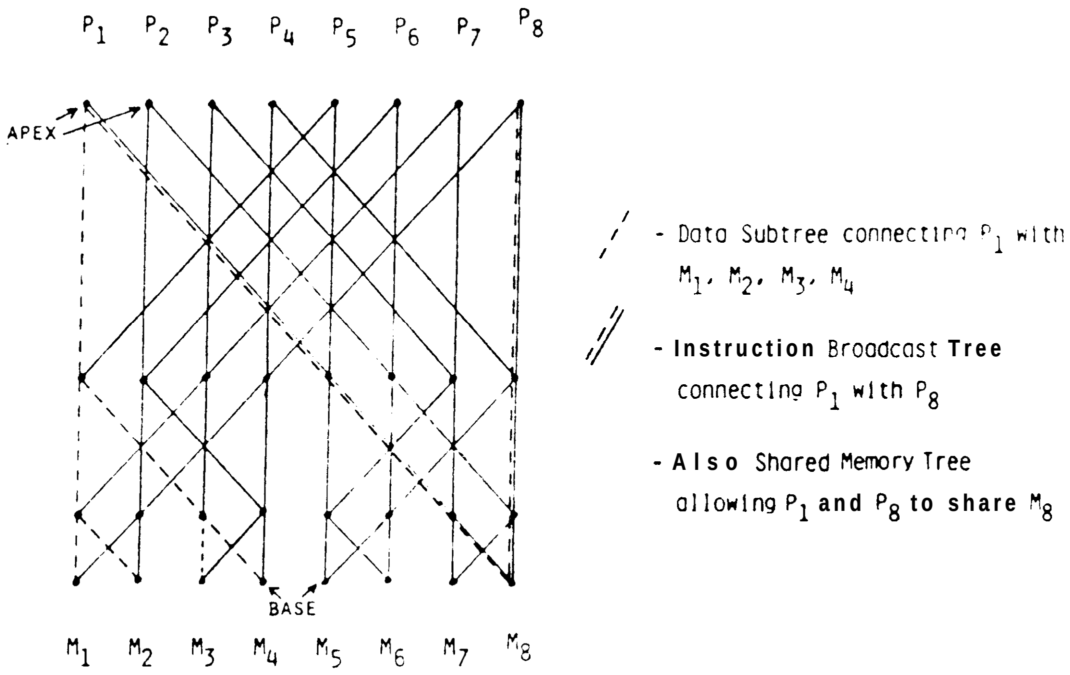

P1 P2 P3 P4 P5 P6 P7 P8
APEX
BASE
M1 M2 M3 M4 M5 M6 M7 M8
- Data Subtree connecting P1 with M1, M2, M3, M4
- Instruction Broadcast Tree connecting P1 with P8
- Also Shared Memory Tree allowing P1 and P8 to share M8

Figure 37: TRAC's Banyan Network

another means of processor-processor communication is packet switching. The packet transmission occur as a background activity so that they do not interfere with other activity.

TRAC subsystems can be architectured to implement multiple models of computation: process forking and joining, task pipelining, data-flow, vector parallelism, and synchronous parallelism.

4.3.3 Different Dimensions of Concurrent Architectures

In the last section, we have described **18** different concurrent architectures. They are so different in structure that it is hard to classify them in any single way. We have already described the different models of computation used by Haynes et. al. in section **4.3.1,** there are four other dimensions that we can identify to describe these machines. They are interconnection network, processing element, memory system, and application. In this section, we will use these five dimensions to classify the **18** architectures described in the previous section.

Models of Computation The six different models of computation described by Haynes et. al. are multiple special-purpose functional units (or pipeline), associative processors,

	Pipeline	Assoc. Memory	SIMD	Dataflow	FP Language	Multiple CPU
Systolic	X					
STARAN		X				
Illiac IV			X			
BSP			X			
MPP			X			
CHiP			X			
NON- VON			X			
ODSP				X		
SERFRE					X	
Cedar						X
FMP						X
S-I						X
Cm*						X
HEP						X
Empress						X
MP/C						X
Ultra						X
TRAC	X		X	X		X

Figure 38: Models of Computation

array processors, data flow computers, functional programming language machines, and multiple processors.

Interconnection Network In the machines that we have discussed, there are many types of interconnection network. Systolic arrays are connected in a pipelined fashion. STARAN has its own FLIP network. MPP and Illiac IV and MPP are mesh-connected. BSP and **S-l** use full crossbar. CHiP uses the switch lattice. NON-VON is tree structured. Cm* and MP/C are bus oriented. Cedar, FMP, and Ultra use the Omega network. HEP uses a pipelined switch. Empress has a quadratic memory matrix. TRAC has a 2-3 SW Banyan Network. Some of these interconnection networks can again be divided into either central or distributed control. Reconfigurability is a feature of some of the networks, which allow them to reconfigure the system resources dynamically to match the need of the problem. For multi-stage networks, three types of switching modes are possible: circuit, message, and packet. The purpose of the interconnection network is for the communication among processors (P-P), or processor to memory (P-M), or both.

Processing Element The number of processing elements used in each architecture varies from 1 for the DDSP to 64K for the CHiP. Most architectures allow variable number of processing elements. Requirements on the processing elements also vary.

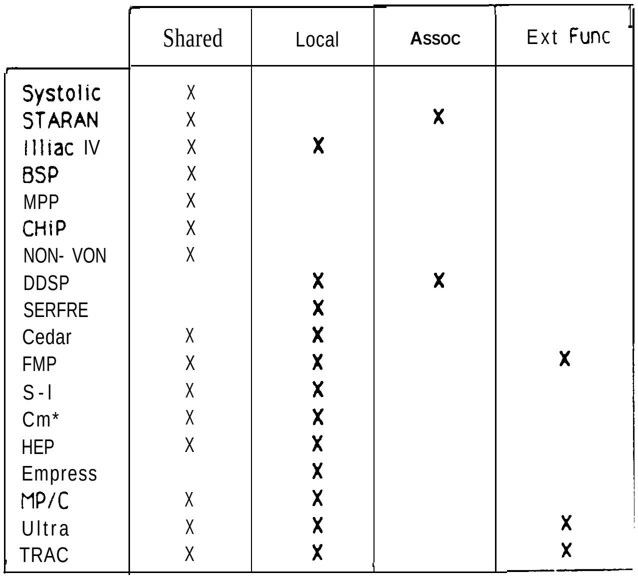
STARAN, **MPP,** and NON-VON use very simple processors. CHiP, Cm*, Empress, and **TRAC** use off-the-shelf microprocessors or LSI-11. Others use powerful custom made

	Type	Control	Reconf ig	Switching	Commun
Systol ic	Pipeline		N		P-P
STARAN	FLIP		N		P-P
Illiac IV	Mesh		N		P-P
BSP	Crossbar	Central	N	Circuit	P-M
MPP	Mesh		N		P-P
CHIP	Sw lattice	Central	Y		P-P
NON-VON	Tree		Y		P-P
DDSP	Bus	Dist	N		P-P
SERFRE	Bus		N		P-P
Cedar	Omega	Dist	N	Circuit	P-M
FMP	Omega	Dist	N	Circuit	P-M
S-l	Crossbar	Central	N	Circuit	P-M
Cm*	Bus	Dist	N	Packet	P-M
HEP	Pipe Switch	Dist	N	Packet	P-M
Empress	Quad Matrix	Central	Y		P-P
MP/C	Bus	Disc	Y		P-M
Ultra	Omega	Dist	N	Message	P-M
TRAC	2-3 Banyan	Cent & Dist	Y	Pack & Cir	P-P,M

Figure 39: Interconnection Network

	Number	Power	Size (bits)
Systol ic	variable		
STARAN	32x256	simple	1
Illiac IV	64	3-5 MFLOPS	64
BSP	16	1-4 MFLOPS	48
MPP	16,384	simple	
CHIP	256-64K	µP	
NON-VON	>16 K	simple	8
DDSP	1-32	2.2 MFLOPS	
SERFRE	variable		
Cedar	128x16	2.5 MFLOPS	32
FMP	128	IO MFLOPS	64
S-l	16	3 MFLOPS	64
Cm*	variable	LSI-11	16
HEP	variable	10 MIPS	64
Empress	variable	LSI-11	16
MP/C	variable		
Ultra	4096	VLSI, fast FP	
TRAC	16	µP	8

Figure 40: Processing Element



	Shared	Local	Assoc	Ext Func
Systolic	X			
STARAN	X		X	
Illiac IV	X	X		
BSP	X			
MPP	X			
CHiP	X			
NON- VON	X			
DDSP		X	X	
SERFRE		X		
Cedar	X	X		
FMP	X	X		X
S-l	X	X		
Cm*	X	X		
HEP	X	X		
Empress		X		
MP/C	X	X		
Ultra	X	X		X
TRAC	X	X		X

Figure 41: Memory System

processors. The word length of the architectures ranges from 1 bit for STARAN and MPP, to 64 bits for Illiac IV, Cedar, FMP, S-l, and HEP.

Memory System Memory system can be either shared or local. In shared memory, different processors can access the same memory cell for communication and synchronization. Data stored in local memory can only be accessed by the processor attached to it. Some local memory are also used as instruction cache to reduce the traffic in the interconnection network. STARAN and DDSP have associative memory for content-addressable memory and matching store respectively. Ultra has adders in the memory system to support the fetch-and-add operation. TRAC has index registers residing in the memory modules so that a shorter 8-bit macro-instruction can be sent by the processor instead of a longer 16-bit full address. References to locations in memory modules are made by specifying one of the index registers.

Application These 18 machines are designed for different applications. In here, we have listed a few important and representative areas: general purpose, scientific, data base, image or signal processing, simulation, testbed, and divide-and-conquer.

	Genera l Purpose	Scien	Data Base	Image/ Signal	SImulat	Testbed	Divrde Conquer
Systolic		X					
STARAN				X	X		
Illiac IV		X					
BSP		X					
MPP				X			
CHIP		X					
NON- VON	X	X	X	X			X
DDSP				X			
SERFRE	X						
Cedar	X						
FMP		X			X		
s- 1	X	X		X			
Cm*					X	X	
HEP	X	X					
Empress		X			X		
MP/C			X				X
Ul tra	X						
TRAC		X	X		X	X	

Figure 42: Application

4.3.4 Conclusions

Although the survey is not complete, it does cover a wide spectrum of both commercial and research machines. It is not the purpose of this research to give a detailed summary of each machine, but to give a general idea of the current status of the research in concurrent architecture. Different architectures are specified using five dimensions: models of computation, interconnection network, processing element, memory system, and applications. Other dimensions, especially software aspects, can also be used, for instance, languages, operating system, scheduling method, communication and synchronization method. With the use of this specification method, we can specify the range of machines that could be modeled by our concurrent system testbed.

4.4 Available Modeling Tools

Another concern during the Background Study phase was to review the available tools which might be available to assist in the performance projection process. These included Emmy, a hardware-based computer system emulator, and Adlib-Sable, a simulation system developed originally at Stanford and now available commerically.

4.4.1 Emmy System Analysis

An emulation system known as Emmy was chosen as the subject of this study because it had been heavily used (at Stanford) in the past. Simulators for IBM 360, DELtran, ADEPT, PDP11, HP1000, Intel's 8080, Data General's Nova, MIPS, and others existed. In addition, local experts on the system were available.

Emmy is a user-microprogrammable machine, with a **4K** microstore. The fast microstore is available for storage use by the microprogram to simulate target machine resources, such as registers. The control store can be dynamically loaded. This dynamicism also allows for some cacheing of microinstructions which can be moved in from main memory as needed. All data paths are 32 bits wide, and address size is 24 bits.

Emmy was intended to be a pipelined machine, with three pipe stages. These stages were: the I-machine, which controlled instruction fetch and decode; the T-machine, which controlled register/register operations and branching; and the A-machine, which controlled non-register storage. The present implementation of Emmy uses ECL and NMOS technology, with **a** PDP 11/07 running Mini-Unix as a front end. Several LSI versions were also built. None of the Emmy implementations were ever actually pipelined: the I, T, and A stages were executed sequentially.

The primary software support for Emmy is Emmyxl, a cross assembler for the Stanford Emmy. This assembler allowed for micro-programming Emmy on the front end system in a mnemonic assembly language. Other support software for driving devices, loading programs, etc., **was** available.

The main data extracted from Emmy simulations was counts. Counts of virtually anything, such as instructions executed or memory access patterns, could be acquired. These values, however, had to be counted by the emulation microcode, as there was no built-in, unobtrusive measurement mechanism.

Several good and bad things were clear about the Emmy System. When first built, even though not pipelined, Emmy was 10-100 times faster at emulation than a conventional machine. Although Emmy is now no faster than **a** Vax for emulation, if reimplemented into a pipelined machine with newer faster components, Emmy would probably regain its speed factor advantage of **10-100.** However, the addition of the instrumentation code for counting often slowed emulations by **a** factor of two.

The opinion of Emmy's users is that it was simply too difficult to develop an emulator. Insufficient tools were available, and developing and debugging at the level of microcode assembly language was too difficult and time consuming.

Emmy was not developed to support emulation of large or concurrent systems, and thus has no special facilities for doing so, nor is especially suited to multi-processors emulations.

Finally, because Emmy was constructed from special-purpose hardware, the maintainability of the system reduced as the original project participants left Stanford. This observation is important when considering options in the development of the Testbed and Emulation Tools.

4.4.2 Adlib-Sable System Analysis

Adlib-Sable is a hierarchical simulation system originally developed at Stanford University. Since its original development, the system has been developed commercially and now is available in the general CAD marketplace. Adlib-Sable is capable of representing systems of different levels of abstraction. The signals between functional units are not constrained with lots of limitations. Adlib-Sable requires that the topology definition be separated from the functional definition of a system, a feature often found to be advantageous.

In order to test the potential use of this system, an interconnection network simulator generator was developed which could be parameterized to specify the type of node, the type of interconnection, etc.

After spending **a** considerable amount of time with this system, we discovered that most of the time was spent on the following areas:

- presentation of results
- specification of test case generation alternatives
- debugging

Presentation of Results Adlib-Sable supported the definition of the functionality of the units. However, problems arose when beginning to decide how to 'instrument' the simulation in order to observe what was happening during simulation. Initially, the experimenter does not have **a** firm **idea** of what the key factor of system performance will be. Monitors of average delay time, total number of packets sent, etc. are inserted as a matter of course at this initial point. In addition, a few debugging features, such as system-status snapshots, special events reporting (such as queue full), and special packet tracing, were initially installed. Then system debugging began. At that point, the initially installed 'instrumentation' didn't seem realistic at all. For example, the report of an average time delay must be understood. What contributes to the **average** delay? Without an understanding of the factors contributing to the overall factor, the overall factor is not understood or appreciated. However, in order to understand the contributing factors, the trace of more detailed information leads immediately to **a** screen full of a very large amount of disorganized data. Printing out the data doesn't help much either, except to make sure that a physical record exists of what occurred during that experimental run.

Special output packages had to be developed, first just to organize the outputs of the built-in instrumentation and put it in tabular forms. Later experiments were concerned with graphical presentations such as panel meters on the display screen in order to present the data in a manner easier for the human observer to understand and analyze. While the specification of the instrumentation could be done in Adlib-Sable, it was supported more from the point of view of additional functionality within the system, rather than from the point of view of how to observe something of interest within the system.

Test **Case Generation Alternatives** As mentioned before, Adlib-Sable was very supportive in the specification of the functionality of the interconnection network functionality. The purpose of creating these simulators was to be able to test them under widely varying loads, especially where these loads were, themselves, derived from real applications. Unfortunately, Adlib-Sable did not have facilities to produce test cases.

In the case of the interconnection network simulation system, a very flexible packet generator would be an extremeely valuable tool for the test bed, both for debugging and for exploring the behavior of the system being simulated. A powerful and flexible testcase generator could be used to setup experiments to single out key factors of performance or to force some particular event to happen in order to understand how a particular system event might occur.

Some of the functions such a test case generator might provide are:

- Generate packets from a fixed random function. The user would specify the random function and the average rate of generation.
- Generate packets of the size and distribution specified by the user. The size could be expressed by some computational function. For example, short (250 byte) fixed-sized packets might be generated 70% of the time and exponentially distributed packet sizes with a mean around 1K bytes might be generated 30% of the time.
- Generate packets with specified attributes. Packets can carry attributes such as trace bit, flush bit, broadcast bit, priority bit, and many others as a tool in debugging and probing the performance of the system being simulated.
- Generate packets from an external file definition. External file formats then become an additional issue. Specifically:
  - Should the external file name out every packet, or could it be a generator, or could it specify the behavior of a generator pending further consultation with the external file. (i.e. definition of a packet generation process)
  - Should command sequences be provided so that more complex testcases involving a combined generation approach could be utilized.

The user needs the flexibility to call on any of the testcase generation alternatives dynamically while studying the system being simulated. Recompilation, linking, and loading would be too cumbersome to expect efficient production of new results. A logical approach is to have an integrated user command language, where both the batch file and the shell of the interactive user interface share the same command language. However, in order to make the most out of the command language, it should be able to incorporate debugging commands.

Debugging Debugging the generated interconnection network simulators was probably the most difficult task during this part of the study. The difficulties were in part, because parallel systems are difficult to debug. The lack of debugging tools in Adlib-Sable contributed to the difficulties immeasurably.

Adlib-Sable itself is a compiler which compiles the simulation specifications into Lpascal. Therefore, Adlib-Sable is an Lpascal preprocessor. Debugging in Adlib-Sable is accomplished using the Lpascal debugger, which is the only means that a user has to access/identify/examine/ or alter the variables in the program. The Lpascal debugger is primitive and supports a limited number of functions. In addition, after Adlib-Sable converts a simulation specification to Lpascal code, keeping track of the relationships between variable names and functional structures in the original specification and the executable forms produced seems almost impossible. Since the debugger did not relate back to the simulation specifications, debugging was extremely difficult.

Many systems are made up of homogeneous components. In this case, a single component in the source code may have many copies in the final program. This impression is fostered by **a** system which has different registers for the internal state copies, but a common set of functional code. When executing this sort of system, many pointers and array indices must be analyzed, just to locate the object of interest.

Debugging parallel programs represents another form of difficulty. **A** sequential program can be followed step, by step, by step-with time implicitly following. This implicit time stepping is **lost in a** parallel system. Now many functions are executing 'simultaneously' in different parts of **a** system. To understand the interactions between these different parts of a system, careful management of global system time is required. Adlib-Sable does not provide this capability. For example, consider the Adlib Sable statement:

UPON BooleanExpression CHECK event DO statement

Adlib-Sable is not clear what the detailed action will be during simulation. Will the event be queued up and the statement be made ready for execution upon completion of the event check? Or will the event check be lost because the statement will already be executing, or what? Even a simple statement like

ASSIGN expression TO y SYNC sysclock PHASE1 can have many possible interpretations. Would the value of the expression be evaluated while executing that statement, or would the execution be delayed until y is really ready to be assigned to?

During debugging, a structured report of what is going on within the system is important. If a particular node is found to behave abnormally, the internal states of that node AND the surrounding neighbors may be of specific interest. All of this information needs to be observed simultaneously. However, all internal states of all nodes are not of interest all of the time. Rather, the right information from a selective set of nodes at a particular time is all that is needed or desired. What the right information is cannot be predicted until it is needed. Therefore, debugging tools must be interactive and integrated with the built-in instrumentation.

Often interesting anomalies appear in the middle of the simulation. In order to observe and understand them, the entire start of the simulation must be completed before the 'interesting' work starts. Check-point facilities, to allow restart from an intermediate state without having to resimulate from the start can significantly speed this sort of work. In addition, facilities to insert break-points, or traps, in a program which activate only when the condition occurs are very useful. For example, we might write:

break on condition.

This facility would interrupt the normal flow of execution when certain conditions are met. The insertion, enabling, and disabling of these break conditions must also be done dynamically during the simulation rather the during the original design. Otherwise an unacceptable cycle involving program editing, recompilation, relinking, debugger loading, and debugging environment setup would be resumed.

In Adlib-Sable, an artificial execution environment had to be created manually for debugging purposes. Dummy data structures were created with special internal states and many program variable changes. Initial setup effort for the simulator was, therefore, exceptionally high. The cost of this initial setup effort was an impediment to future implementations of a new debugging environment when major changes were made in the simulation system.

Other Drawbacks of ADLIB-SABLE In addition to the above major missing capabilities in ADLIB-SABLE, the following are some additional drawbacks:

- Static connections: The connections between portions of a system are fixed at compile/link time and cannot be changed during program execution. No generalization is provided to describe repetitious, related connections. Therefore, the user has to list out every connection explicitly. Adlib-Sable does support structured data passing between two units, however, the entire data structure must be changed at the same time using a single ASSIGN statement. This constraint prevents the use of a single connection to carry all the signals between two units. The result of these constraints is that the topology file becomes very large for some applications, in particular, for network simulation. Simulation of networks is almost impossible when all connections must be written by hand. When the connections are generated automatically, such as in the network simulator generator, which was the experiment here, no means was provided to check the correctness of the productions.

- ADLIB-SABLE requires concern about connections inside an abstraction, even when already defined. For example, if a number of gates are connected to an MSI component and if several of these specific MSI's are to be connected together, the connections inside all the MSI must be specified.
- If the components of a group of identical elements are to be identified, other techniques must be used to generate the component ID's rather than definition within the element itself.
- While the version of ADLIB-SABLE we used was a number of years old, it suffered the pangs of much 'university' code. The language itself is not very stable. Very little support is available (consisting primarily of other graduate students who may have used the system earlier). Many times long debugging sessions discovered that what was thought to be a simulation system error was a bug in the language.
- The productivity of a user of ADLIB-SABLE in this environment is very low. The reason is that 'turn-around' time and cost is very high. If a change is made to an Adlib-Sable program, it must be recompiled (using Adlib-Sable, compiled again using Lpascal (which gives a listing of compiler output for use during debugging since the listing gives the relationships between compiler input and output variable names), relink the program with other libraries, run it with the debugger, and setup the debugging environment. This high cost tends to discourage the user from experimentation with different settings, with alternatives, and with other test patterns.

5 Testbed and Emulator Tool Alternative Studies

Three alternatives to the design of testbed and emulation tools were considered. These were:

- IF1 Simulation
- Special Hardware
- Intermediate Level Modeling

5.1 IF1 Simulation

IF1 is an intermediate form of the SISAL dataflow language. This intermediate form is the common target for various SISAL front-ends and serves as a virtual machine definition for SISAL implementation experiments on the Denelcor HEP, on DEC VAX 11/780 computers, on the Cray 1, and on the Manchester Dataflow machine.

SISAL is a single assignment dataflow language, and did not seem to be a good choice to use to represent the various levels of application instantiation. Since the SASL experiments showed the possible power of a purely functional approach, feasibility of mapping SASL to IF1 was conducted.

5.1.1 SASL to IF1 Compiler

The goal of building this compiler was to examine the effect of different source languages on the resulting executable code. IF1 is an intermediate graph for the data flow single assignment language SISAL, whose basic philosophy is very different from SASL applicative philosophy. This particular combination of source and destination language was chosen because of previous familiarity with SASL, and the large amount of work being done on IF1. Code generators are being built from IF1 to several machines, including HEP and CRAY **1.** If the compiler here is successful, and the code-generators elsewhere are successful, the compiler would become a useful tool for testing applicative languages on supercomputers.

The compiler was written with Yacc, C, and Lex. Some problems were encountered, the major ones being that parts of SASL were not LALR. The lack of accurate and precise documentation, especially for IF1, was also a severe handicap. However a working compiler was produced. A simulator exists for IF1 which is available to us (from LLNL). However, contrary to the specification of IF1, which specifically states that type nodes are optional, the version of the simulator we had **requires** type nodes in its IF1 program. This constraint is impossible to meet when compiling from SASL, as the type of some expressions cannot be determined until run time. So, the IF1 output from the compiler has not yet been actually executed.

However, the compiler was still a success in that it showed that the applicative language could be compiled to a data-flow single-assignment intermediate form. The main difficulties encountered in compiling resulted not from the basic differences between applicative and single-assignment philosophies, but rather from slight differences in the semantics of the two languages such as the handling of undefined values, the use of infinite data structures, etc.

The SASL to IF1 compilation process could be made operational (assuming some work on the IF1 simulation system). However, the level of detail of modeling in IF1 (instruction level modeling) was felt to be too low level for a software-based testbed and emulation-based execution environment. This level of simulation detail would not allow the study of large enough models.

5.2 Custom Hardware Testbed

5.2.1 Requirements of the testbed

The testbed should be an effective tool for studying the behavior of any concurrent system of interest. It should be able to model the concurrent system in arbitrary details (according to the input specifications), to simulate execution of any target program at a reasonable speed, to pinpoint any noteworthy system behavior during the simulation, to gather statistics from the program executions, and to extract system characteristics and performance measures from these statistics. Because behavior of concurrent systems is not yet well-understood, while actually building a concurrent system is very expensive, the testbed should be able to aid efficiently both the design and the evaluation of concurrent systems.

For the design efforts considered during this study, the following baseline requirements were established for a custom hardware testbed.

1. It should be capable of simulating a multiprocessor system of one thousand processors.
2. The simulation time should be less than two days for one hour of actual program execution (on the same technology scale). In other words, the simulation should not be more than fifty times slower.
3. The testbed design should be modular and expandable.

In order to provide versatility and better utilizability, the following desirable features are also required.

1. The user should be able to interact with the testbed during simulation. The simulation process should be monitored with built-in instrumentation such that the user could display any selected result dynamically. The testbed should be always under user's control.
2. Necessary monitoring facilities should be provided. These include insertion of system check points, provision of various types of stepping modes, and capability to alter internal process states and data during simulation.
3. The system should be able to run simulations of different system representations. For any two representations between which explicit correspondence is established, capability to switch from one representation simulation to the other representation simulation without program re-execution should be provided.

4. Larger concurrent systems should be able to be simulated, but with slower simulation speed.

5.2.2 Conceptualization of the system under simulation

In the testbed design, the system under simulation is conceptualized as an interconnection of functional units. Functional units are entities with some specified properties, while interconnection links between functional units are merely abstraction of the relationships between some attributes of the connected functional units. For example, if it is a physical system, then **a** functional unit can be a processor, a memory unit, a interconnection block (a switch or a network), or a subsystem, and an interconnection link represents an physical interconnection without any delay; if it is a software system, then a functional unit can be a process, a module or task, a procedure or subroutine, or simply a block of codes, and an interconnection link represents either a data dependence or a precedence relationship.

Accordingly, any representation of the system under simulation is formulated as a (directed or undirected) graph with nodes and edges representing the functional units and interconnection links respectively.

5.2.3 Organization of the testbed

As shown in Figure 43, the testbed in design is functionally decomposed into three components: the testbed component, the statistics collection and analysis component, and the control and user interface component.

The testbed component performs the modeling and simulation functions. It consists of a number of functional unit emulators, a configuration network and some memory extension units. The functional unit emulators are processors with local memories. They can emulate any designated functional units of the concurrent system under study. The configuration network interconnects all the functional unit emulators and is configurable to support the interconnection links of the concurrent system under study. The memory extension units are global memories or external storage subsystems. They are accessible by any functional unit emulator to serve as the extension of the corresponding local memory.

The statistics collection and analysis component collects and stores all the simulation statistics and system informations. All during-simulation and after-simulation data analyses are performed by this component. It consists of a number of instrumentation processors, a instrumentation network and a mass storage subsystem. All instrumentation processors have their own local memories. Some of them are tightly-coupled with the functional unit emulators in the testbed component to provide efficient primitive data collection. The other processors perform the data routing, filtering, formating and the subsequent analysis. The instrumentation network interconnects the instrumentation processors and the mass storage

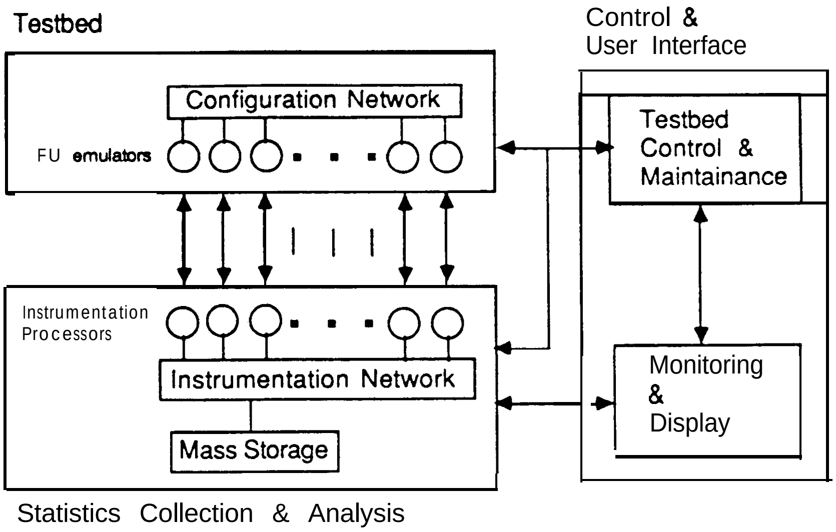


Figure 43: Organization of the testbed

subsystem, and is configurable to aid efficient statistics collection and analysis. The mass storage subsystem stores all simulation and system statistics for future references.

The control and user interface component consists of two subcomponents. The control subcomponent directly controls the operations of the entire testbed, and maintains the testbed. The user interface subcomponent interacts with the user. It allows users to monitor the simulation process and to display results dynamically.

5.2.4 Design of the configuration network

The design of the configuration network in the testbed component is the most essential part of the entire testbed design. It directly impacts the overall performance of the testbed. In the following, the general requirements of the configuration network are first stated. Then it is shown that the design problem can be formulated as three interrelated subproblems. Finally, some results related to these subproblems, especially the mapping problem, are reported.

I. General requirements of the configuration network

For the testbed component to model a given concurrent system, it is essential to map the system representation graph into the testbed structure. That is, every functional unit in the representation graph is designated to be emulated by an emulator in the testbed, and every interconnection link between two functional units is realized by some communication paths between the corresponding emulators. The basic requirement of the configuration network is to provide the necessary connectivity among the emulators in the above mapping.

In order to achieve acceptable simulation performance, it is required that for every interconnection link in the representation graph, the resulting communication delay is short and within some tolerable limit. Besides, any network configuration and functional unit designation should be accomplishable within a reasonable time period. The algorithm to determine the above mapping should be of complexity of polynomial running-time. High utilization of the available emulators is desirable.

The design of the configuration network should be modular and incremental expandable. It should be of low interconnection complexity and should be .compatible with the current device and packaging technology.

II. Formulation of the network design problem The configuration network design problem can be formally formulated into three interrelated subproblems.

Assumption: Any given system representation can be formulated as a graph in which the degree of each node is constrained to a small integer d. This assumption is justified on the basis that in reality, both the hardware building components and the software elementary functional units have limited input/output capabilities due to either physical or logical constraints.

The three subproblems are:

1. The interconnection structure problem

   Find an interconnection structure of low interconnection complexity such that the other two subproblems can be solved in polynomial time.

   Let $\mathbf{G}_{test}= (\mathbf{N}_{test},\mathbf{E}_{test})$ be the graph denoted such structure.

2. The mapping problem

   For an arbitrary graph, $\mathbf{G}_{obj}=(\mathbf{N}_{obj},\mathbf{E}_{obj})$, of maximum degree of d and $|\mathbf{N}_{obj}| \leq k\,|\mathbf{N}_{test}|$, find a mapping f: $\mathbf{N}_{obj} \rightarrow \mathbf{N}_{test}$ such that:

   a. for any u,v in $\mathbf{N}_{obj}$, $f(u)=f(v)$ iff u=v ;

   b. for any e=(u v) in $\mathbf{E}_{obj}$, it is mapped to a path $(f(u)\ .\ .\ f(v))$ in G $_{test}$ and the length of the path is at most D ;

   c. for any edge in $\mathbf{E}_{test}$, at most c edges in $\mathbf{E}_{obj}$ are mapped to paths via the edge.

   The ratio k bounds the size of the system (in terms of the number of functional units) to be mapped. The parameters D and c characterize the maximum intrinsic communication delay (i.e. delay without any traffic congestion and contention) and the maximum communication traffic allowed in the resulting configuration, respectively.

3. The graph reduction problem

   For any graph $\mathbf{G}_{obj}$ of maximum degree of d and of size greater than $k|\mathbf{N}_{test}|$, find a graph reduction g: $\mathbf{N}_{obj} \rightarrow \mathbf{2}^{N_{obj}}$ such that :

a. $|\{g(u):$ u is in $N_{obj}\}| \leq k|N_{test}|$;

b. the reduced graph $G_{red}=(N_{red},E_{red})$ has maximum degree of d,
where $N_{red}=\{$ g(u): u is in $N_{obj}\}$ and $E_{red}=\{(g(u)$ g(v)) : (u v) in $E_{obj}\}$;

c. for any two distinct edges (u v) and (u v') in G_{obj}, g(v) $\neq g(v')$; and
for any two distinct edges (u v) and (u' v) in G $_{obj}$, g(u) $\neq g(u')$.

The motivation of the first two conditions is to transform the problem into the mapping problem of the reduced graph. The third condition is necessary for maximum concurrency of the resulting system.

III. The mapping problem The mapping problem is difficult in general. It bears some resemblance to the subgraph isomorphism problem which is known to be *NP* complete. At the time of this study, no satisfactory solution to this essential problem had been found. In the following, some preliminary studies are presented.

bf Work by Rosenberg [RS 78]

Arnold L. Rosenberg has extensively studied a similar problem (the data encoding problem). The following two results are of interest:

1. Let $G=(N,E)$ be a connected graph. For any elementary cycle or line of order $|N|$, there is a mapping to G such that

   (a) the longest path mapped by an edge is of length at most 3 ;

   (b) the average length of the path mapped by an edge is at most $2 - 2|N|$

   Moreover, the mapping can be found in time 0($|$ El), and is optimal.

2. For any mapping of a graph G of maximum degree d_G to a connected graph H of maximum degree d_H, the longest path mapped by an edge is of length at least $log(d_G)/log(d_H)$.

A graph expansion technique

The motivation of the introduced technique is to transform the object graph into simple graphs of known structures. Therefore the subsequent mapping may become simpler.

By this technique, the node set of the object graph is duplicated into many copies. The edges of the object graph are distributed among these duplicated node sets to form a set of simple graphs. Each resulting simple graph is then mapped into a line graph. Each line graph can be further mapped into any connected graphs. Figure 44 shows an example where the object graph is expanded into a set of complete binary trees.

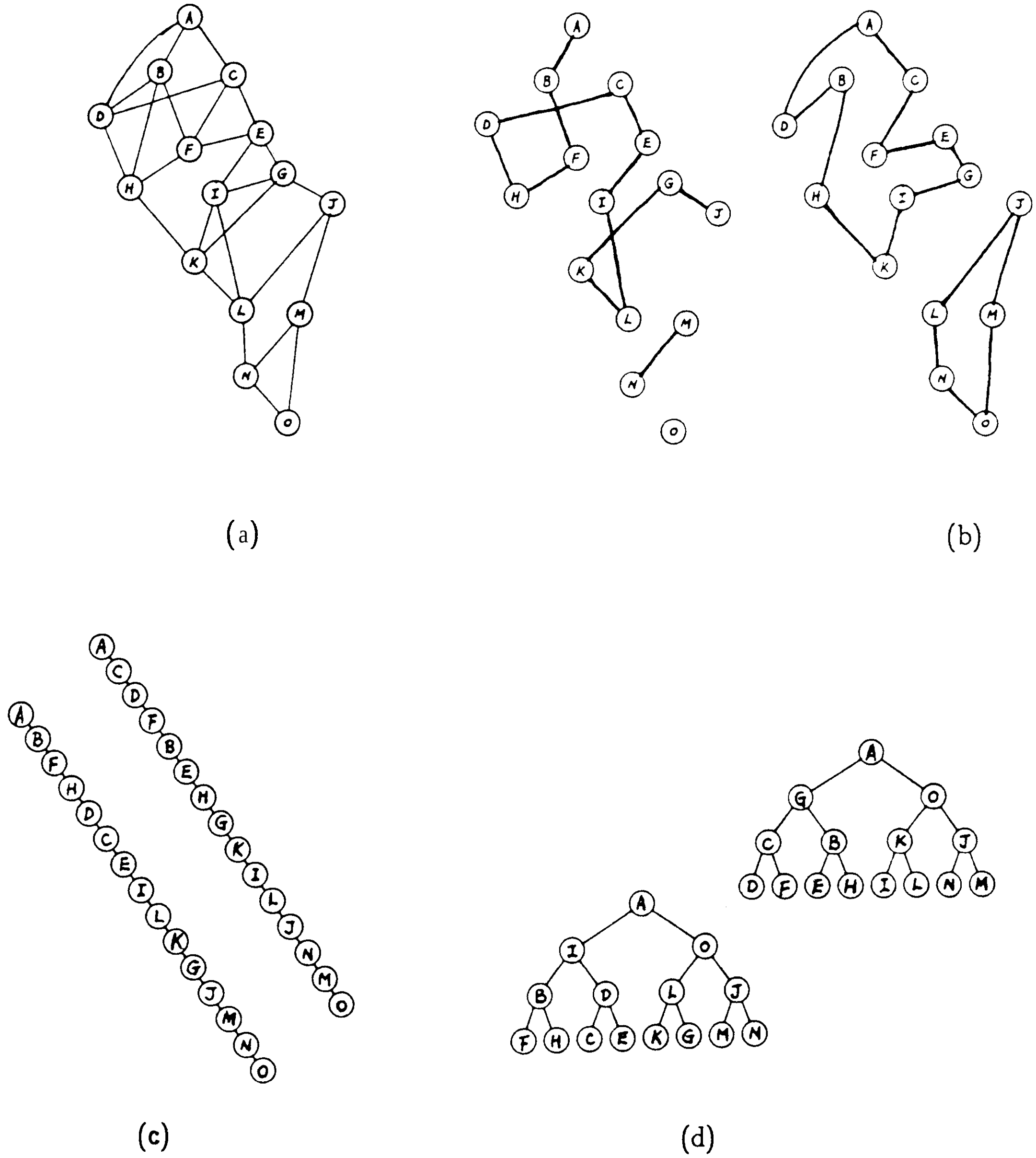


Figure 44: The graph expansion into complete binary trees: (a) the object graph, (b) after edge partition, (c) after mapped into line graphs, (d) after further mapped into binary trees.

In general, the following theorem is established.

Theorem: any graph $G=(N,E)$ of max. degree $2n$ can have its edge set E partitioned into n sets, i.e. E_i, $i=1,..,n$, such that each graph (N,E_i) can be mapped into a line with each edge mapped into a line path of length at most 2. Furthermore, each graph (N,E_i) can be mapped into any connected graph of the same size with each edge mapped into a path of length at most 5.

Proof:

First the graph is augmented to be $2n$ regular[1]. By the Petersen Theorem[2] in graph theory, the edges in the augmented graph can be decomposed into n disjoint factors. (This can be done in time at most of order of $(nN)^2$, where N is the size of the graph.) Then each factor is mapped into a line by transforming each cycle $(u_1,u_2,u_3,..,u_x,u_y,u_z,u_1)$ into a line $(u_1,u_2,u_z,u_3,u_y,...)$ and joining the resultant lines. It is easily seen that each edge is mapped to a line path of length at most 2. Furthermore, by Rosenberg's algorithm, any line (or elementary cycle) can be mapped into any connected graph of the same size with each edge mapped to a path of length at most 3. It can be shown by the same mapping, every line path of length 2 is mapped to a path of length at most 5. This concludes the proof. Q.E.D.

The above graph expansion technique leads to the following observations:

1. Since each node is duplicated into n nodes, the connection of a node to its n images becomes a new problem.

2. If no subsequent graph reduction is performed, the resulting utilization of the available emulators, i.e. $k \leq 1/n$, may be acceptable only for small n. When $n=2$, or $d=4$, the utilization is no more than fifty percent.

3. If permutation networks are used to connect the n images, the resulting path for each edge becomes of length of $O(logN)$, where N is the size of the original graph.

4. If the permutation networks are used to setup permanent hardware links between a node and it's n images, then the delay can be maintained to be constant (i.e. 4 for lines, 7 for connected graphs) during simulation time.

General interconnection network considerations The part of the study is partly motivated by the above observations. First, it is meaningful to know what is the theoretical bound on the diameter of any degree-constrained graph. In other words, how short a communication delay can be in a network of restricted interconnection complexity. Related

[1] A graph is regular if each node in the graph has the same degree.

[2] Petersen Theorem [1891]: If G=(N,E) is a regular multigraph of even degree d=2n, then G has n edge-disjoint factors. [A factor of a graph is a set of node-disjoint elementary cycles such that each node is contained in exactly one cycle.]

questions are : what tradeoffs can be made and how effective they are. The result is summarized below. Then, a class of interconnection networks is defined. It is shown that this class of networks processes many desirable properties. It may be one of the potential interconnection structure to be adopted.

1. **The Moore bound** [AL **82**]

For any graph of size N and maximum degree d, it's diameter D (i.e. the maximum distance between any two nodes) must be at least

$$\frac{\log(N(d-2)+2)-\log(d)}{\log(d-1)}$$

Furthermore, $D.$ log(d) is at least of order of log(N).

This gives readily the following implications.

- For constant d, D is at least of order of log(N).
- Even d grows as log(N), D is at least of order of log(N)/ log(log(N)).
- The Moore bound is difficult to attain (if not unattainable). It is also difficult to decrease D without increasing d.
- At the Moore bound, d increases much faster than D decreases.

2. **A Class of Interconnection Network**

Before the class of interconnection network is defined, it is necessary to mention the product of two graphs.

The ***product*** of two graphs $G_1=(N_1,E_1)$ and $G_2=(N_2,E_2)$, denoted as $G_1 \times G_2$, is the graph (N,E) that

(a) N = $N_1 \times N_2$; and

(b) ((u x) (v y)) in **E** if and only if either (u v) in E_1 and x=y
or (x y) in E_2 and u=v.

Figure 45 gives **an** example.

A Recursive Definition** of a **Class** of **Interconnection Networks:

(a) R_1 is a connected graph, designated as the elementary building graph.

(b) $R_{i+1} = R_i \times R_1$

***Properties** Of* R_i :

(a) Size of $R_i : N_i = N_1^i$

(b) Maximum degree of R_i : $di = i.d_1$, i.e. $O(log\ N_i)$

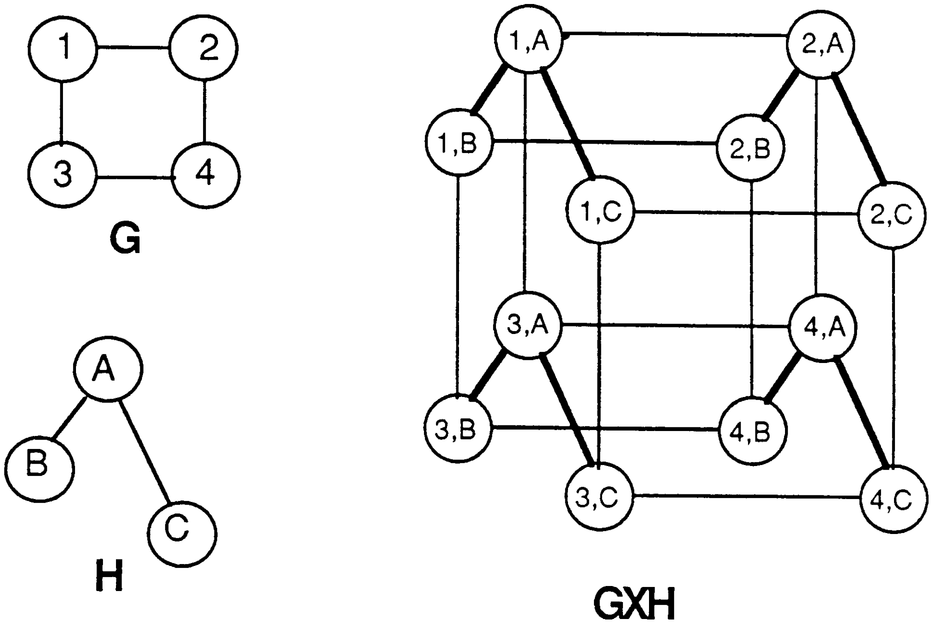


Figure *45:* The product of two graphs

(c) Diameter of $\mathbf{R}_i$: $Di = i.D_1$, i.e. $O(log\ N_i)$

(d) The routing control scheme is distributed and simple. In $\mathbf{R}_i$, every node can be identified uniquely by an i-digit NJ-based number. Any two nodes can communicate with each other within i routing steps. The routing is based on the difference of the identity numbers of the source and destination nodes. In each routing step, the difference number is reduced in a way that one of its non-zero digits is changed to zero and the other digits remain unchanged. This is easily accomplished by routing through the corresponding $\mathbf{R}_1$ connections. The order of the routing steps is immaterial.

(e) Since multi-path communication is available between any pair of nodes, the network is fault-tolerant. It is also dynamically adaptable to the network traffic to maintain low communication delay.

(f) Since for each expansion of $\mathbf{R}_i$, i.e. from $\mathbf{R}_i$ to $\mathbf{R}_{i+1}$, it is always N_1 modules of $\mathbf{R}_i$ to be connected in a predefined way (the $\mathbf{R}_1$ interconnection), the implementation of $\mathbf{R}_i$ is regular and incremental expandable.

This class of interconnection network is general. It defines the hypercubes and includes

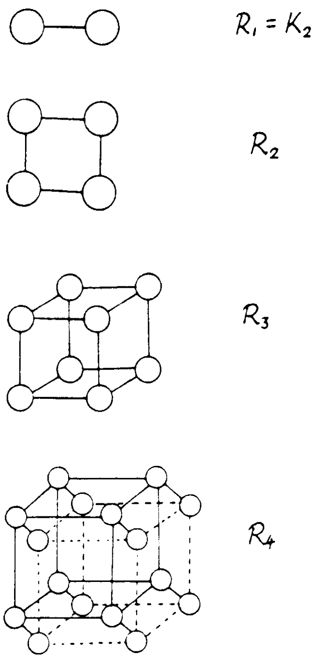


Figure **46:** The class of network with R_1=K_2, the n-cubes

topologically many multi-stage networks. When $\mathbf{R_1}$=$\mathbf{K_2}$ (i.e. complete graph of two nodes), $\mathbf{R_i}$ defines the ***hypercubes*** or ***n-cubes***.(Figure **46)**

The class of networks with $\mathbf{R_1}$=$\mathbf{K_4}$ is considered in particular (Figure **47).** For this class of networks, d_i=$3.log(N_i)/2$ and D_i=$log(N_i)/2$. Table **1** gives some figures comparing this class with the n-cubes. It is found that for the range of size of interest, the diameter is reasonably small while the maximum degree is still physical maintainable. Furthermore, the implementation of the $\mathbf{K_4}$ is compatible to current VLSI technology. All these make this particular class appealing. It may be one of the potential interconnection structure to be adopted in the testbed design.

Table 1: Comparisons between two classes of networks

	R_1=K_4		**n-cubes**	
N	***d***	***D***	***d***	***D***
256	**12**	**4**	**8**	**8**
4096	**18**	**6**	**12**	**12**
65536	**24**	**8**	**16**	**16**

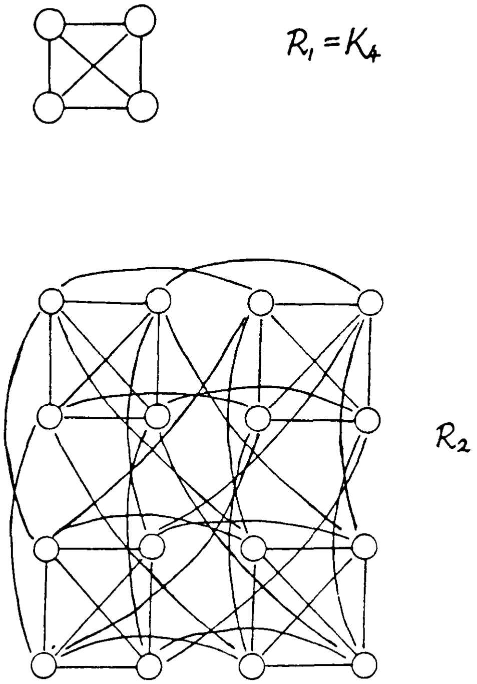


Figure 47: The class of network with $R_1 = K_4$

5.2.5 Summary

The success of the custom approach requires a satisfactory solution to map an arbitrary graph onto the proposed interconnection structure. Although the proposed interconnection structure has a number of desirable properties, the mapping problem remains to be solved.

The custom approach has the additional drawbacks of high-cost of original development, questionable maintainability, and substantial software cost in run-time environment and support tools.

5.3 Intermediate Level Analysis

A final alternative considered was to conduct analysis of the execution of an application from the point of view of an intermediate language and virtual machine structure. In this case, the intermediate language hypothesized is a higher level than that represented by the IF **1** option considered earlier. The proposed structure of the concurrent system testbed would consist of three parts:

- software analysis,

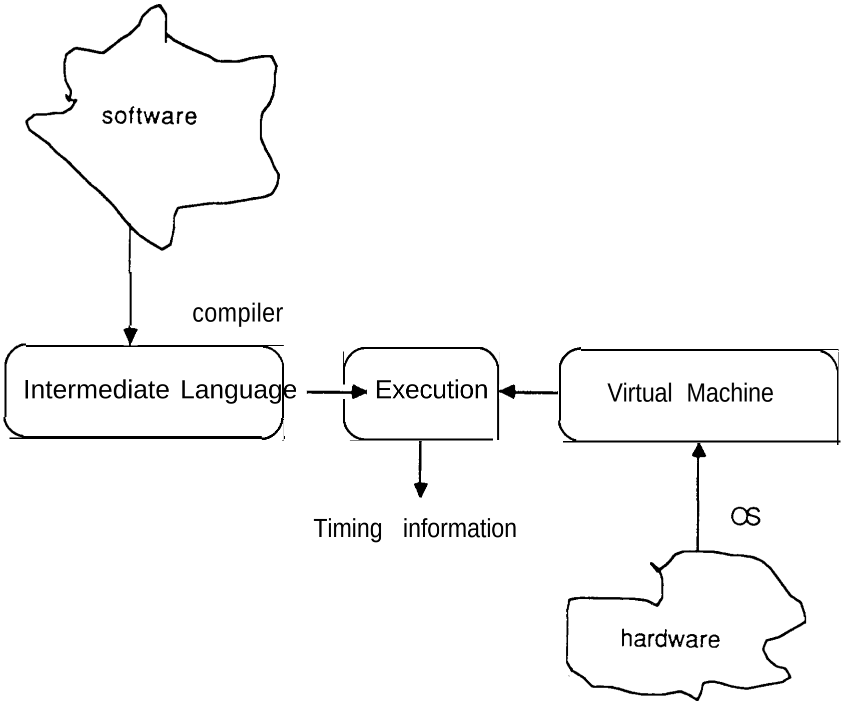


Figure 48: Block Diagram of Intermediate Level Virtual Machine Model

- hardware modeling, and
- execution modeling.

Figure 48 shows a block diagram of this structure.

In software analysis, the effects of the compiler in translating the source code to the intermediate language would be modeled. Compilers for different languages, or even different compilers for the same language, might produce different intermediate code for the same application. The differences are even more significant when optimization is taken into account.

The form of the intermediate language would be some form of dependency graph like a Directed Acyclic Graph (DAG), showing the precedence relationship among instructions. The system proposed assumes that either the compiler is smart enough to detect all possible concurrency inherent in the software, or that the concurrency was identified explicitly in the source code so that the compiler could detect it. No automatic parallelization is assumed.

In hardware modeling, a concurrent machine is modeled as a virtual machine consisting of a set of primitives, or components, that can be used to execute the intermediate language used in the software analysis portion of the system. Thus, different machines can be modeled as the same virtual machine, but with different timing characteristics from each other. By hiding the lower level implementation details, the amount of computation needed to study execution should be significantly reduced.

The execution unit accepts the DAG, which defines the software model, and the virtual machine representation of the underlying hardware system, and produces approximate performance projections. These projections may include total execution time, resource utilization, throughput, response time, maximum degree of concurrency utilized, location of system bottlenecks, etc.

Notice that the modular structure of the testbed allows different combinations of software and hardware to be modeled in the same way. Execution of the same software model by various virtual machines could be used to determine which machine organization would be the most effective for actual execution of that application. Similarly, once an underlying machine design is complete, a virtual machine model could be created and exercised with various applications in order to determine the sensitivity of the machine design to various application characteristics. This modularity ensures that once a piece of software or hardware is modeled, it could be saved in a library and reused later without the need for initial preprocessing.

5.3.1 Software Analysis

Imperative languages like FORTRAN, Pascal, and C, are still the most popular in today's software. A large fraction of important application software is written in these languages. However, in future concurrent systems, other types of languages, such as dataflow languages, functional programming languages, or logic programming languages, may be dominant. Thus, the design of our testbed should be independent of the source language so that software written in different source languages can still be modeled. A graphical form of intermediate language is chosen as a language-independent form to represent the various software to be studied.

An algorithm, when written in different source languages, may map to different machine code, even on the same machine. These variations are due, in part, to the semantic gap between the language and the underlying machine. In addition, compiler optimization aggravates the problem. Different optimizing schemes produce different code sequences. Thus, in software analysis, we have to make sure that the DAG obtained from the source language does represent the best possible code sequence regardless of which compiler is used in the process.

Another problem specific to concurrent systems is the expression of concurrency. The maximum inherent degree of concurrency of programs written in high level languages are usually not explicitly shown. The use of the DAG as the intermediate language also comes to aid in this case. All precedence relationships are displayed graphically so that all processes can be executed concurrently can be easily identified.

In summary, we need a compiler system to translate the source code into the graphical intermediate language form. This compiler system should have a set of frontends, one for each possible source language. Different optimization schemes should also be supported in such a manner that they could be turned on and off individually in order to experiment with their effects on performance individually. The software modeling subsystem must be able to extract and represent all the inherent concurrency in the software. This requirement may lead to the addition of notation to be manually inserted in the source programs since many existing languages do not have sufficient notation to represent concurrency.

5.3.2 Hardware Modeling

The hardware is modeled as a virtual machine consisting of a set of primitives. These primitives are common to all machines so that they form a uniform interface to the execution unit of the testbed. The use of the virtual machine primitives models the effect of the operating system in hiding hardware details from the software. This approach allows very different concurrent machines to appear as the same virtual machine with differing timing characteristics among the virtual components.

The set of primitives uniquely represents the underlying modeled machine by preserving the timing characteristics and resource usage information. They are used by the testbed to interpret the nodes in the DAG into multi-resource, multi-stage tasks. Each task has a service demand on each of the physical resources for a certain amount of time. For example, a typical task may demand 3 milliseconds of service from the CPU unit in the first stage, in the second stage, 100 microseconds from the interconnection network and 20 microseconds from the memory system, then in the third stage, another 10 milliseconds from the CPU.

The choice of the set of primitives is very important. The choice must be made in such a way that the DAG's of all application software that will be modeled in the testbed can be interpreted. An example of the set of primitives is given below. Of course, they are not the only possible set—other sets may also be considered.

1. **Expression Evaluator:** When given an expression, this primitive will evaluate the expression and give the result back. The expression may contain any operators that the source language supports.

2. **Data Structure Manager:** This primitive takes care of all memory accesses and memory management. This primitive is also responsible for setting up and updating the data structures used in the software.

3. **Repetitive Structure Controller:** This primitive controls loops in imperative languages, and streams or set generators in functional languages. For loops in Pascal, say, given the loop count, exit condition, and the timing of the loop body, it can give the total execution time of the loop. The repetitive structure controller will try to

find the best way to execute the structure, especially when a lot of parallelism may be specified.

4. **Conditional Selector:** This primitive takes care of all conditional statements in the source language, like if-then-else and case statements. This primitive contains the timing characteristics of the machine in choosing alternatives.

5. **Communication Manager:** This primitive is responsible for all the communication among processes, whether they are by message passing or through monitoring.

6. **I/O Manager:** This primitive is responsible for all input and output activities of the machine.

7. **Procedure Calling Manager:** This primitive controls all procedure calls, parameter and result passing, and context switching in the software.

8. **Process Scheduler:** Schedule all ready processes to be executed subject to resource constraints. Different operating systems might use different algorithms in scheduling their processes. These scheduling differences would be reflected in this primitive.

In order to model the hardware as a set of virtual machine primitives, we have to have a hardware simulation system that will allow us to model and measure the timing characteristics of the modeled machine. This system should allow the user to build his model of the hardware component easily and generate test cases for measurement purposes. The system should also have an automated data collecting feature so that data generated in the measurement can be used automatically to generate the set of virtual machine primitives necessary for the modeling process.

5.3.3 Execution Modeling

In projecting the performance of concurrent systems, the functional results are not required. Rather, performance **measures** such as execution time, resource usage patterns, response times, throughput, etc. are of interest. Therefore, the software does not have to be executed instruction by instruction. The system must only do enough to keep track of resource usage and the impact of the usage on elapsed time.

The execution unit of the testbed takes in two pieces of information:

- a DAG from the software modeling subsystem with all precedence relationships among the nodes, and

- the set of virtual machine primitives as given by the hardware modeling subsystem

The nodes in the DAG are then interpreted by the virtual machine primitives into multi-resource, multi-stage tasks. Now, the execution can be viewed as a scheduling problem: schedule the set of tasks subject to the precedence relationship given by the DAG and also the physical resource constraints of the modeled machine. The execution unit has to take into account the possible resource conflicts among tasks. A Gannt chart can be used to display the schedule as shown in Figure 49.

Task scheduling, taking precedence relationships and physical resource constraints into account, can proceed in many possible ways. However, conventional scheduling methods for multi-resource, multi-stage tasks subject to resource constraint are usually NP-complete. Therefore, we have to pursue other approximations which can provide fast turnaround time in modeling the execution.

Recently, queueing network models have been used successfully in modeling computer systems. Due to the computational efficient mean value analysis of the queueing network model, performance measures for computer systems can be obtained easily using this model. However, queueing network models, by themselves, cannot be used to model concurrency. Thus, these models cannot be used directly to model the execution of the task systems. In order to implement this approach to performance projection, new ways to apply queueing network models to analyze DAG's are needed to obtain approximations to the desired performance estimations.

5.3.4 Summary

Three important future research areas have been identified in order to implement an intermediate-language based modeling and analysis system.

- software analysis of concurrent software,
- modeling of concurrent machines as a set of virtual machine primitives, and
- queueing network models applied to model the execution of a set of tasks with precedence constraints.

In software analysis, a compiler system that is able to translate software written in different source languages into the graphical intermediate form is required. It should be able to model the effects of optimizing compilers and manage explicitly identified concurrency in the software.

In hardware modeling, a hardware simulation system is required for the user to model components of the concurrent system and to obtain timing characteristics. An automated data collecting feature is also desired to analyze the measurement data collected and to generate the virtual machine primitives automatically.

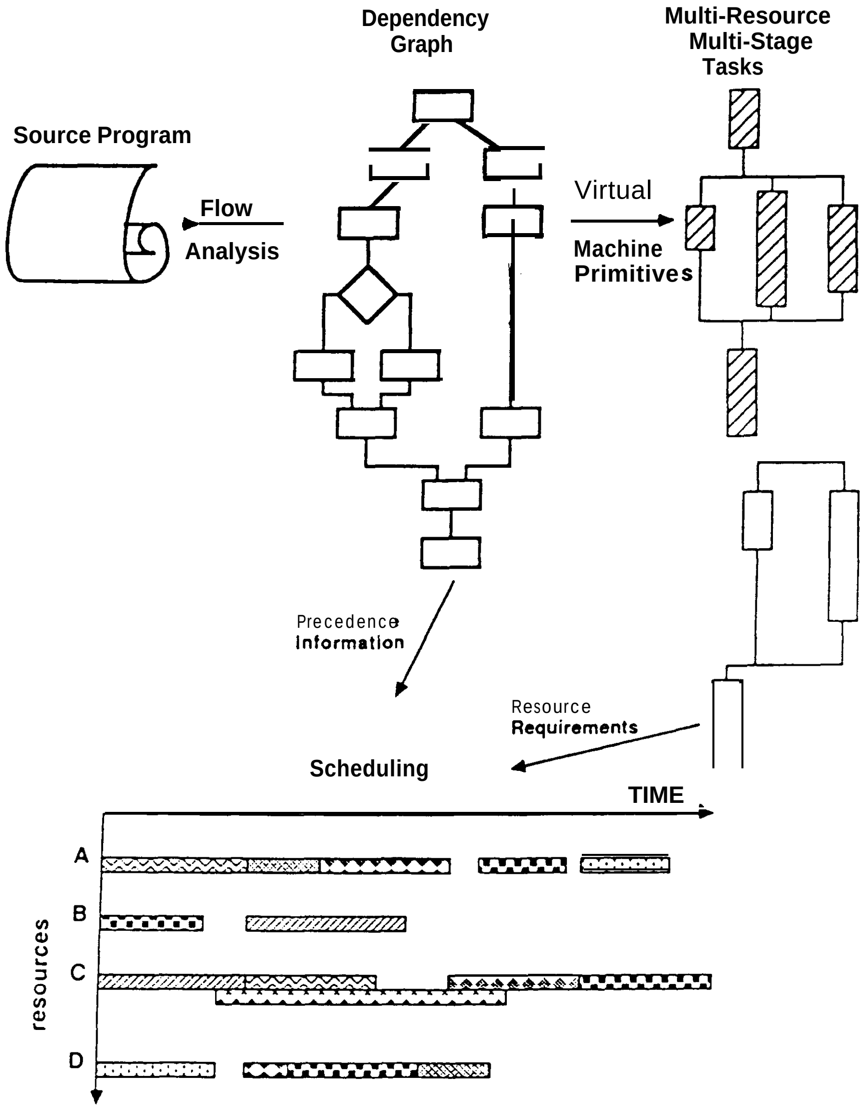


Figure 49: Components of Intermediate Level Performance Analysis

In execution modeling, we are investigating the possibility of using queueing network models to model the execution of task systems with precedence constraints. This method allows us to have a computationally efficient system to obtain approximate performance measures of concurrent systems in a fast turnaround environment.

6 Conclusions and Plans

Based on the work done in the background studies and review of the strengths and weaknesses of the various potential approaches to the Testbed and Emulation Tools identified in the last chapter, the third approach, Intermediate Level Analysis, is recommended as the potentially the most cost-effective approach for the development of Testbed and Emulation Tools. The Intermediate Level Analysis approach has the benefit of being software based, thus being easily ported to various execution environments (custom hardware does not have this capability). Intermediate Level Analysis represents a balance between very high-level approximations, which cannot easily be verified, and very low-level simulation, which cannot easily be conducted on very large problems. By working at an intermediate level, the application needs only be mapped part way in terms of detail. Similarly, the hardware only needs to be generalized part way from its most detailed representations. In both cases, we expect that a better understanding of the hardware and software models will be possible at the Intermediate Level.

Figure 50 shows another version of the major components of such a system. The program to be studied must undergo some sort of software analysis. The Directed Acyclic Graph produced is typical of a high-level graph representation of the program which might be produced (at least at a machine level) within a typical compiler. The nodes in the graph represent that portion of a program that can be executed sequentially (using only one portion of the hardware resource at a time). This form of software representation should allow various compiler optimizations to be represented within the same form so that various representations of the same software might be able to be considered.

A queueing network model form of hardware model is recommended. Here the various components of the hardware system are represented as the higher-level functional units, such as the virtual functional units described in section 5.3.2, where each of the functional units may have a queue of work pending. This network would represent the hardware with some operating system influence, especially at the resource management level. The Performance Prediction system then must simulate the execution of the high-level Directed Acyclic Graph on the high-level, virtual-component based hardware queueing network model.

The most immediate problem in the implementation of this approach is the Performance Prediction portion of this system. Queueing network models cannot handle concurrency. The analysis of Markov Chains is extremely complex (combinatorial complexity) for large

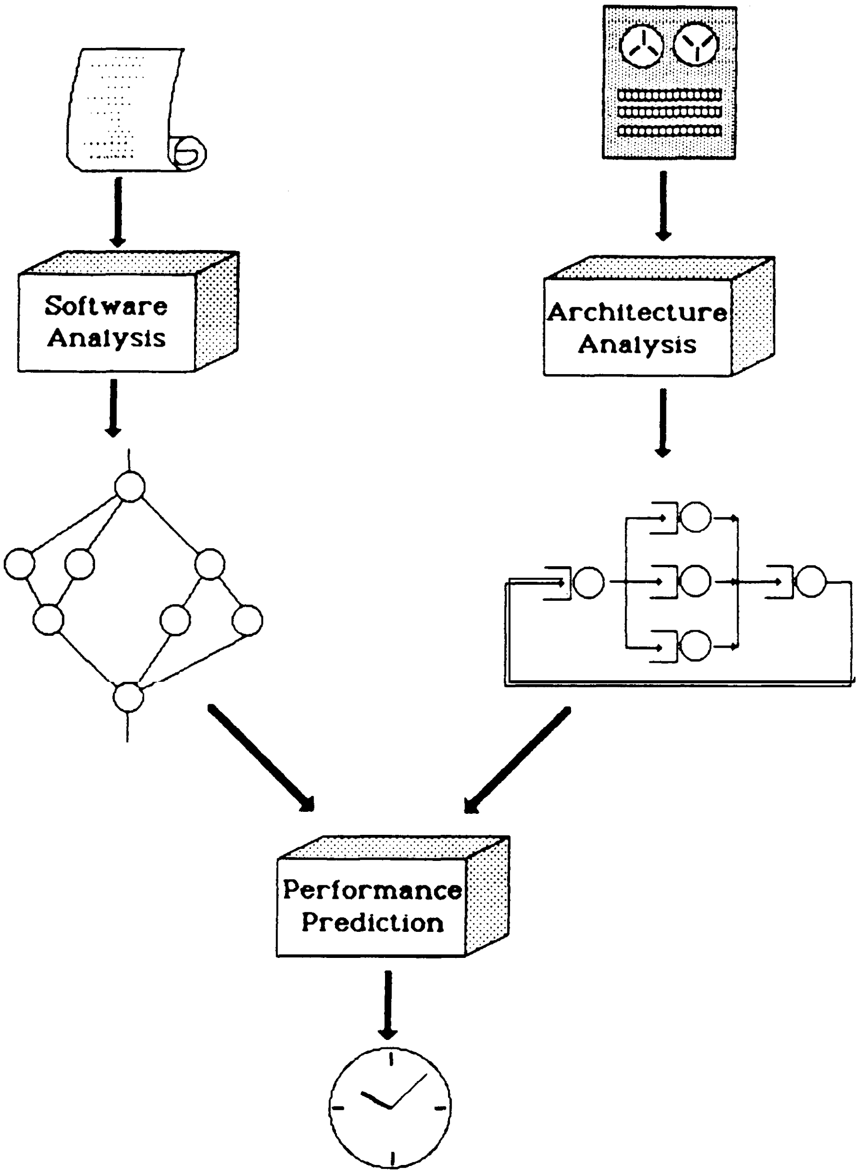


Figure 50: Testbed and Emulation Tool Components

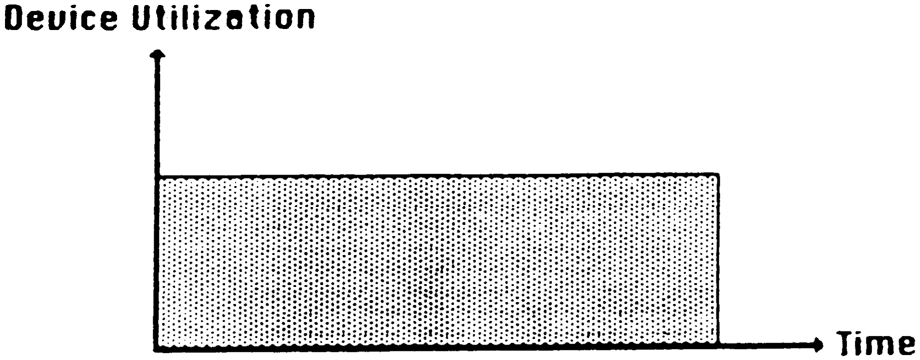


Figure **51:** Constant Device Utilization Assumption

problems, and would be computationally intractable, even for an Intermediate Level Analysis, such as this. Therefore, a new approach to Performance Prediction is needed.

6.1 New Approach to Performance Prediction

We have proposed a new approach to the Performance Prediction. This approach involves an ideal algorithm and then application of that basic ideal algorithm in more practical settings.

6.1.1 **Ideal Algorithm**

First assume that the utilization of each service center in the hardware model is constant throughout the entire execution of the Directed Acyclic Graph. This assumption is shown graphically in Figure **51.**

If this assumption holds, then the time required to complete execution of the task system represented by the DAG can be determined by the following algorithm:

1. Guess the initial utilization, U_0, for each device.
2. Compute the task residence times using mean value analysis.

   $$Ti = f(U_{i-1})$$
3. Find the task system completion time, C_i.
4. Compute a new set of estimates of Utilization for each device.

   $$U_i = g(C_i)$$
5. Iterate steps 2, 3, and 4 until all Ui's converge.

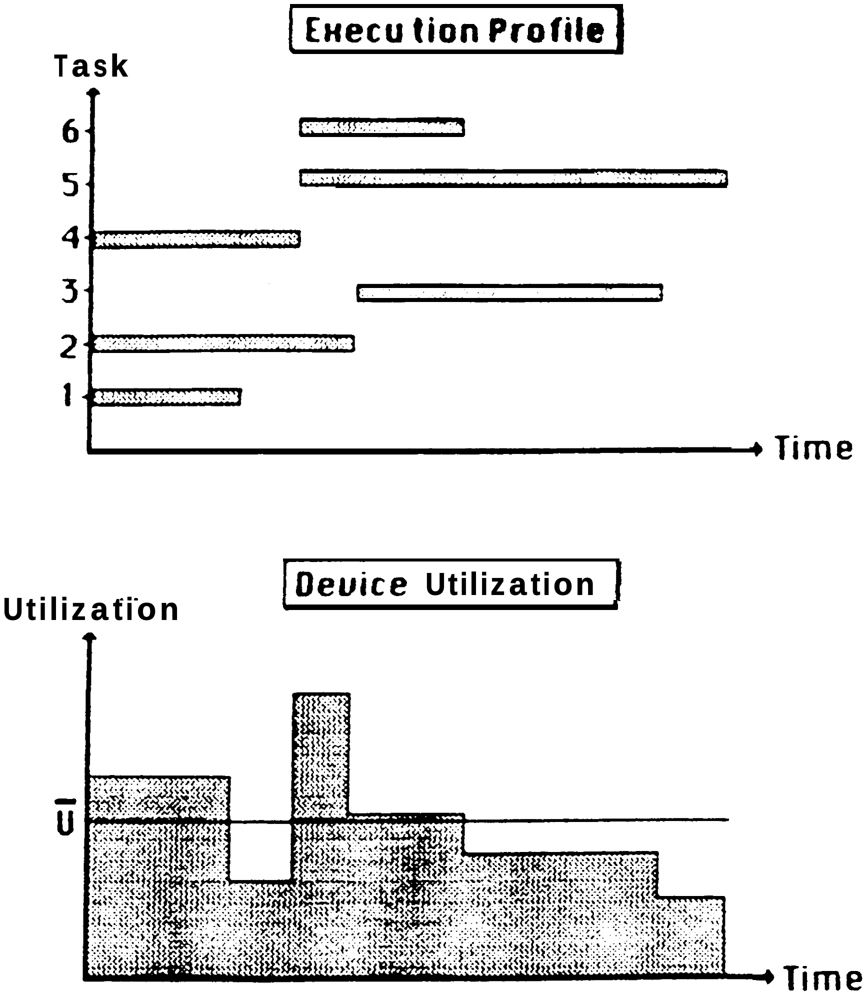


Figure 52: Realistic Device Utilization

6.1.2 **Realistic Algorithm**

As should be obvious, the basic assumption above is not realistic in an actual system. For example, **as** Figure **52** shows, device utilization is not uniform at all over the time of execution of **a** DAG. However, the assumption can hold incrementally. That is, we can choose regions (sometimes arbitrarily small) such that within the elapsed time of the evaluation of that portion of a DAG, the assumption is quite valid. Therefore, a general execution profile of a DAG can be used to estimate device utilizations. The entire execution is then divided into intervals of ***constant*** utilization. Task Residence Times can then be computed using utilization values in the execution intervals utilizing the ideal algorithm already described. The overall Task Completion Time is an accumulation of the times in the various intervals.

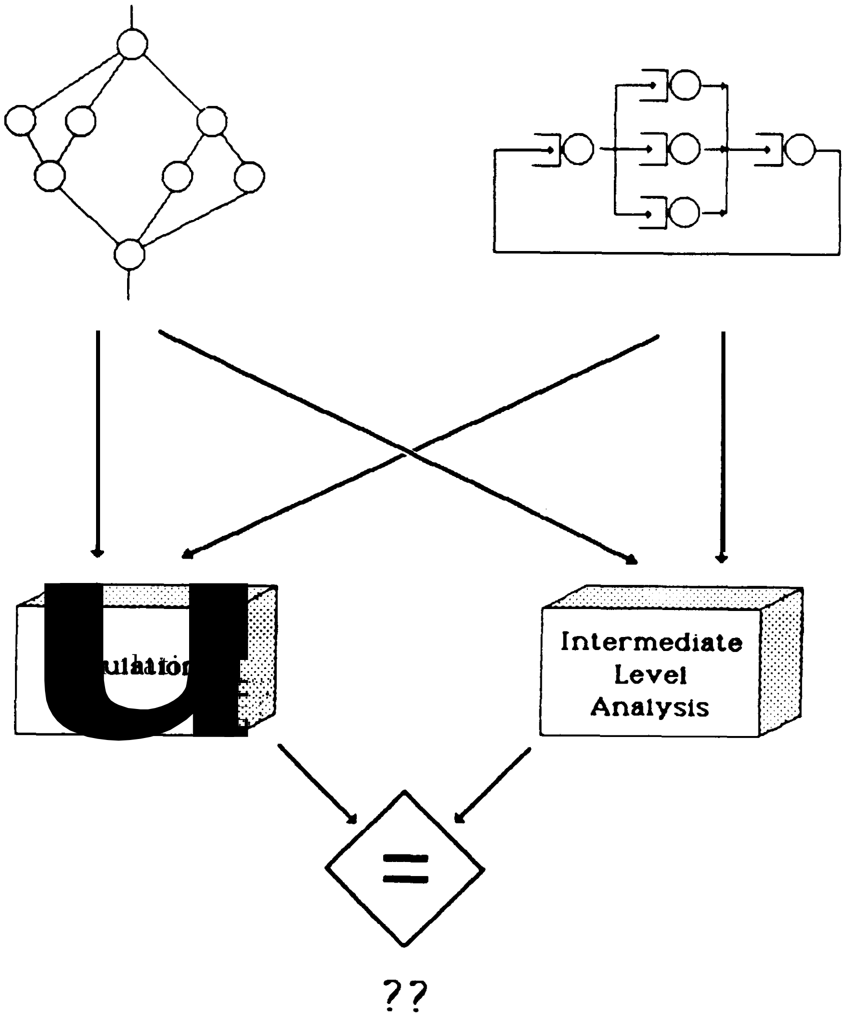


Figure 53: Research Approach to Validate New Method

The complexity of this algorithm is projected to be $O(N^3)$, considerably better than the combinatorial complexity of Markov Analyses. This approach still needs a considerable amount of research before it can be applied. The exact constraints under which it can be applied must be determined. These constraints must then be mapped back to determine the types of software applications and hardware structures that can make use of this form of Intermediate Level Analysis. A general experimental method to the validation of this approach to analysis is proposed in Figure 53. Specifically, establish the capability to both simulate and conduct Intermediate Level Analysis from the same DAG and queueing network model. Then the results of this new approach can be compared and contrasted with the specific operations actually performed during a simulation, leading to a much better understanding of the potential power of this proposed approach.

8 References

[AG 82] B.W. Arden, and R. Ginosar, "MP/C: A Multiprocessor/Computer Architecture,,, IEEE Trans. on Computers, Vol. C-31, No. 5, May 1982, pp. 455-473.

[AL 82] B.W. Arden, and H. Lee, "A Regular Network for Multicomputer Systems,,, IEEE Trans. on Computers, Vol. C-31, No. 1, Jan 1982, pp. 60-69.

[Bat 74] K.E. Batcher, "STARAN Parallel Processor System Hardware," AFIPS Conf. Proc., Vol. 43, 1974 NCC, pp. 405-410.

[Bat 82] K.E. Batcher, "MPP: a supersystem for satellite image processing," AFIPS Conf. Proc., Vol. 51, 1982 NCC, pp. 185-191.

[BBB 82] R.E. Buehrer, H.J. Brundiers, H. Benz, B. Bron, H. Friess, W. Haelg, H.J. Halin, A. Isacson, and M. Tadian, "The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System,,, Trans. on Computers, Vol. C-31, No. 11, Nov 1982, pp. 1035-1044.

[BDM 72] W. J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick, "The Illiac IV System,,, Proc. IEEE, Vol. 60, NO. 4, Apr. 1972, pp. 369-379.

[Fly 72] M.J. Flynn, "Some Computer Organizations and Their Effectiveness,,, IEEE Trans. on Computer, Vol. C-21, No. 9, Sept. 1972, pp. 948-960.

[GGK 83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer,,, IEEE Trans. on Computers, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.

[GKL 83] D.Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar – A Large Scale Multiprocessor," Proc. of the 1983 International Conf. on Parallel Processing, pp. 524-529.

[HLS 82] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell, "A Survey of Highly Parallel Computing," Computer, Jan. 1982, pp. 9-24.

[HNI 82] E.B. HOgenauer, R.F. Newbold, and Y.J. Inn, "DDSP-A Data Flow Computer for Signal Processing," Proc. of the 1982 International Conf. on Parallel Processing, pp. 126-133.

[KS 82] D.J. Kuck, and R.A. Stokes, "The Burroughs Scientific Processor (BSP)," IEEE Trans. on Computers, Vol. C-31, No. 5, May 1982, pp. 363-376.

[Kun 82] H.T. Kung, "Why Systolic Architectures?,, Computer, Jan 1982, pp. 37-46.

[Lun 85] S.F. Lundstrom, "A Decentralized Control, Highly Concurrent Multiprocessor," IEEE Proceedings of the 12th Annual International Symposium on Computer Architecture, June 17-19, 1985, pp. 145-151.

[RS 78] A.L. Rosenberg, and L. Snyder, "Bounds on the Costs of Data Encodings," Mathematical Systems Theory, Vol. 12, 1978, pp. 9-39.

[SFS 77] F.J. Swan, S.H. Fuller, and D.P. Siewiorek, "Cm*- A modular, multi-microprocessor,,, AFIPS Conf. Proc., Vol. 46, 1977 NCC, pp. 637-644.

[Smi 78] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer," Proc. of the 1978 International Conf. on Parallel Processing, 1978, pp. 6-8.

[Sny 82] L. Snyder, 'Introduction to the Configurable Highly Parallel Computer," Computer, Jan. 1982, pp. 47-56.

[SUK 80] M.C. Sejnowski, E.T. Upchurch, R.N. Kapur, D.P.S. Charlu, and G.J. Lipovski, "An Overview of the Texas Reconfigurable Array Computer,,, AFIPS Conf. Proc., Vol. 49, 1980 NCC, pp. 631-641.

[Tom 67] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units,,, IBM Journal, Vol. 11, Jan. 1967.

[Vil 82] F.Y. Villemin, "SERFRE : A General-Purpose Multi-processor Reduction Machine,,, Proc. of the 1982 International Conf. on Parallel Processing, pp. 140-141.

[WC 79] L.C. Widdoes, Jr., and S. Correll, "The S-l Project: Developing High-Performance Digital Computers,,, Energy and Technology Review, Sept.8 1979.

Contents

List of Figures