MIT/LCS/TR-363

EXPLOITING PARALLELISM IN VLSI CAD

Joshua David Marantz

June 1986

This blank page was inserted to preserve pagination.

Exploiting Parallelism In VLSI CAD

by

Joshua David Marantz

© Joshua David Marantz, 1986

June, 1986

The author hereby grants to M.I.T. and Digital Equipment Corporation permission to reproduce and to distribute copies of this thesis document in whole or part.

The thesis research was conducted at Digital Equipment Corporation in Hudson, MA, between February and August of 1985, as part of the MIT VI-A Internship Program.

Exploiting Parallelism In VLSI CAD

by

Joshua David Marantz

Submitted to the Department of Electrical Engineering and Computer Science
on June 3, 1986 in partial fulfillment of the requirements for
the Degrees of Bachelor of Science and Master of Science

Abstract

In the domain of computer science, particularly VLSI CAD, an increasing amount of engineering time is spent running compute-bound programs. Many of these programs have an intrinsic parallelism that is externally accessible. This thesis describes a novel software system that uses a small number of independent computers connected by a network to exploit the parallelism inherent in existing software, and thereby reduce its running time. A dynamic task-scheduling algorithm based on a static acyclic data dependency graph is used to control the parallel execution. A network interprocess multi-client message-passing system is developed and discussed. The system is applied to design rule checking by executing each rule on a separate processor, and the results are analyzed.

Name and Title of Thesis Supervisor:

Christopher J. Terman,
Assistant Professor of Computer Science and Engineering

Key Words and Phrases:

parallel processing, design rule checking, CAD tools

Acknowledgments

I would like to thank my DEC supervisor, Ed McGrath, and my thesis advisor, Chris Terman, for the ideas, inspiration, support, and encouragement which made this thesis possible.

I would like to thank the past and present members of the VLSI Methodology and Advanced Development Group and the VLSI Layout Verification Group at Digital who helped provide a stimulating environment in which to do the thesis research. In particular, Ed Prentice and Sam Levitin were both indispensable as sounding boards and critics for my ideas. I would also like to thank my first supervisor at DEC, Bob Gottlieb, for teaching me system programming.

Finally, I would like to thank my entire family for their love and support over the years.

The thesis research was conducted at Digital Equipment Corporation in Hudson, MA, between February and August of 1985, as part of the MIT VI-A Internship Program.

Contents

1 Introduction 15
- 1.1 Accelerating CAD Tools . . . 15
- 1.2 Parallelism in VLSI CAD . . . 17
- 1.3 A Software Methodology for Multiprocessing . . . 17
- 1.4 Chapter Outline . . . 18

2 Background 21
- 2.1 Previous Work in Parallelism for VLSI CAD . . . 21
- 2.2 Previous Work in Accelerating DRC . . . 22
- 2.3 Motivation: Parallel DRC . . . 24
- 2.4 Scheduling strategies . . . 25

3 *EPIC*: A general method of exploiting parallelism 29
- 3.1 Dividing the job . . . 29
- 3.2 Multiprocessing on a Local Area Network . . . 32
- 3.3 Software Architecture . . . 34
- 3.4 Task Scheduling . . . 36
- 3.5 Communications . . . 40
  - 3.5.1 Control Communication . . . 41
  - 3.5.2 Control Communication Requirements . . . 42
  - 3.5.3 Data Communications . . . 44
- 3.6 Fault Tolerance . . . 46
- 3.7 Error Recovery . . . 49
- 3.8 VAXcluster Support . . . 50
- 3.9 Performance Monitoring . . . 52
- 3.10 Results . . . 54
- 3.11 Future Extensions . . . 54

4 Applications 57
- 4.1 Design Rule Checking . . . 57
  - 4.1.1 Predictions . . . 61
  - 4.1.2 Testing . . . 62
- 4.2 Circuit Extraction . . . 68
- 4.3 Compiling and Linking Programs . . . 68

5 Conclusion 71
5.1 Summary . . . 71
5.2 Directions For Future Research . . . 71
5.2.1 Other Applications . . . 72
5.2.2 Reducing the overhead . . . 73
5.2.3 Lessons Learned about Distributed Programming . . . 73
5.3 Conclusion . . . 74

A $\mathcal{EPIC}$/DRACULA User's Manual 77
A.1 How Parallel DRC Works . . . 77
A.2 Potential Benefits From Running Parallel DRC . . . 78
A.3 Environment For Running Parallel DRC . . . 79
A.3.1 Requirements For $\mathcal{EPIC}$ . . . 79
A.3.2 ECAD DRACULA Requirements . . . 80
A.3.3 Input Requirements . . . 80
A.4 Running A Parallel DRC . . . 80
A.4.1 Preprocessing Steps . . . 80
A.4.2 Running $\mathcal{EPIC}$ . . . 82
A.4.3 Triggering The Parallel DRC . . . 85
A.4.4 Summary Files . . . 85
A.5 Appendix . . . 87
A.5.1 Sample Run Of EPIC:FIXECAD . . . 87
A.5.2 Execution Control File . . . 88
A.5.3 Command File . . . 88

B Data Dependency Graphs 89

C Data from the testing of $\mathcal{EPIC}$ 93
C.1 DRACULA with DEC CMOS rules . . . 94
C.1.1 Serial DRACULA on a VAX 11/780 computer . . . 94
C.1.2 Parallel DRACULA on three VAXclustered VAX 11/780 computers . . . 94
C.1.3 $\mathcal{EPIC}$ using one VAX 11/780 computer . . . 95
C.1.4 $\mathcal{EPIC}$ using two VAXclustered VAX 11/780 computers . . . 97
C.1.5 $\mathcal{EPIC}$ using three VAXclustered VAX 11/780 computers . . . 99
C.1.6 $\mathcal{EPIC}$ using two independent VAX 11/780 computers . . . 101
C.1.7 $\mathcal{EPIC}$ using three independent VAX 11/780 computers . . . 103
C.1.8 $\mathcal{EPIC}$ using four independent VAX 11/780 computers . . . 105
C.1.9 $\mathcal{EPIC}$ using five independent VAX 11/780 computers . . . 107
C.1.10 $\mathcal{EPIC}$ using six independent VAX 11/780 computers . . . 109
C.2 Compiling and Linking $\mathcal{EPIC}$ . . . 111
C.2.1 $\mathcal{EPIC}$ using one VAX 11/780 computer . . . 112
C.2.2 $\mathcal{EPIC}$ using two VAXclustered VAX 11/780 computers . . . 114
C.2.3 $\mathcal{EPIC}$ using three VAXclustered VAX 11/780 computers . . . 116
C.2.4 $\mathcal{EPIC}$ using two independent VAX 11/780 computers . . . 118
C.2.5 $\mathcal{EPIC}$ using three independent VAX 11/780 computers . . . 120

C.2.6 *EPIC* using four independent VAX 11/780 computers 122

D *EPIC* Messages **125**

D.1 Messages sent from user to monitor 125

D.2 Messages sent from monitor to master 126

D.3 Messages sent from master to monitor 126

D.4 Messages sent from master to slave 126

D.5 Messages sent from slave to master 126

List of Figures

2.1 Sectioned Data Dependency Graph 26

3.1 Sample Task Description List and Data Dependency Graph 30
3.2 A More Interesting Data Dependency Graph and its Execution 31
3.3 Star Network Topology 32
3.4 Algorithm for Generating Data Dependency Graph 35
3.5 A Skeleton for a Task Scheduling Algorithm 36
3.6 Algorithm for computing height of all tasks 38
3.7 Algorithm for computing the size of all tasks 39
3.8 A data dependency graph and its execution using *height* and *size* 40
3.9 A task scheduling trial simulation where height/size heuristic is suboptimal 41
3.10 Algorithm for determining which tasks have already been done 48

4.1 MOSIS CMOS DRC rules fragment, ECF fragment, and COM fragment . . 59
4.2 Optimistic analysis of DEC CMOS rules based on data dependency 61
4.3 Optimistic analysis of MOSIS CMOS rules based on data dependency . . . 62
4.4 Results and analysis of DEC CMOS DRC tests 65
4.5 *EPIC* analysis of Makefile simulation based on data dependency 69
4.6 Results and analysis of `make epic` 69

Preface

The text of this thesis was formatted using LaTeX. The more complex figures and drawings were generated automatically by the software described in this thesis, using a graphical description language called Postscript[1]. The two forms of printed media were merged together electronically (rather than photographically or with scissors and glue). The capability of automatically merging text with graphics in this manner has made practical the inclusion of a *flipbook animation.*

The animation is an attempt to show how the behavior of the multiprocessing task scheduling algorithm changes according to the number of processors. The parallel execution is simulated under the assumptions of zero communications overhead and unit execution time for each task. The nth "frame" in the animation is a graphical representation of the execution using n processors. The graph is composed of diamonds, which represent atomic units of computation called *tasks.* Horizontally adjacent diamonds represent tasks that are executed in parallel on different processors. The vertical axis represents time, with the beginning of the computation at the top of the page.

Initially, each frame occupied a single page, and the effect of flipping through the 100 pages was aesthetically pleasing. Unfortunately, in terms of the thesis, it was not justifiable as a 100 page appendix. So each frame was reduced so it would occupy the top and right margins of an existing page. The first frame of the animation is visible at the right edge of this page. It shows how when using one processor, no more than one task can be processed at any given time, so they are all executed one after the other. The next page shows the simulation using two processors. The meaning of the animation will become clearer after reading Chapters 3 and 4, but it was necessary to include a word of

[1]Postscript is a trademark of Adobe Systems

explanation here, since this is where the animation begins.

Chapter 1

Introduction

As the complexity of VLSI circuits increases, so does the running time of the CAD tools we use to build those circuits. At the current state of VLSI technology and CAD tool performance, tasks such as layout verification, simulation, and mask-making have proven to be expensive bottlenecks in the VLSI design process. If the advances in the complexity and functionality of the VLSI chips we build are to keep pace with advances made in VLSI process technology, then we must make substantial improvements to the software tools used to design and manufacture those chips.

1.1 Accelerating CAD Tools

There are several ways to accelerate CAD tools:

1. Developing more efficient software
2. Buying faster general purpose computers
3. Using special-purpose hardware accelerators
4. Exploiting the hierarchy inherent in the represention of VLSI circuits
5. Exploiting the parallelism inherent in many of the existing CAD tools

Developing more efficient software is always an attractive alternative. In industrial design rule checking, ECAD's DRACULA2 offered an order of magnitude speed-up over what was previously available[1]. However, significant runtime improvement through better

[1] DRACULA2 is a trademark of ECAD corporation

software is often limited by the computational complexity of the problem at hand. Observations in industry indicate that further improvements are needed in the time taken by design rule checkers.

Buying faster general purpose computers is perhaps the lowest-risk option listed above. If purchasing new hardware will increase both the processing and the memory speed, then it will certainly increase the speed of the CAD tools that run on it. This strategy has the added advantage that it can be easily combined with any of the other strategies. However, since monetary cost rises faster than computational speed, it is not a cost-effective solution. This is evidenced by a comparison of Digital's VAX 8600 and MicroVAX II computers[2]. They were both introduced in early 1985, so they represent roughly the same level of technology. The VAX 8600 computer has approximately 5 times the processing speed of the Microvax II computer, but costs about 10 times as much. Relying on faster computers is also not likely to be a good long-term solution, because recently the complexity of VLSI circuits has grown much faster than the cost of processing speed has fallen. Digital's VAX 8600 computer has four times the speed and twice the cost of the VAX 11/780 computer (1977), while chips of 1985 have twenty-five times as many transistors as those of 8 years ago [Allen 1983].

Developing special hardware accelerators offers the greatest potential of all the solutions listed above. Runtime improvements of several orders of magnitude are not uncommon. In design rule checking, speedup factors of up to 140 have been predicted using small amounts of custom hardware [Seiler 1985]. Similar improvements have been achieved in circuit simulation using the ZYCAD hardware accelerator[3]. Unfortunately, the cost of these devices, both in money and development time, is often prohibitive. In the event of an algorithmic improvement that decreases the growth rate of a problem, the hardware will lose its edge as the problem increases in size, rendering it obsolete.

[2]VAX and MicroVAX are trademarks of Digital Equipment Corporation

[3]ZYCAD is a trademark of ZYCAD corporation

1.2 Parallelism in VLSI CAD

Exploiting the parallelism inherent in VLSI CAD tools is an attractive way to accelerate them. The nature of VLSI lends itself to a high degree of parallelism. VLSI chips are composed of several layers, which are often examined separately. Each layer is composed of different blocks which exhibit a very high degree of functional locality. Each block is composed of many polygons which exhibit some degree of geometric locality.

We observe that the parallelism inherent in VLSI is manifested in CAD tools in several different ways. Logic simulators possess parallelism based on the locality of activity in a digital circuit. [Arnold 1985] exploits this property in a multiprocessing logic simulator based on RSIM [Terman 1983]. Design rule checking and circuit extraction can be accelerated by taking advantage of the geometric locality of the polygons that constitute the chip. [Levitin 1986] describes a system that uses this approach to accelerate a VLSI circuit extractor called IV [Tarolli, Herman 1983]. Similarly, [Bier, Pleszkun 1985] describes a system that divides a layout into separately checkable partitions, checks each partition, examines the partition boundaries to eliminate false errors and catch missed errors, and merges the resulting error reports together. In design rule checking, there is also parallelism inherent in the set of design rules that guide the checking program. This thesis describes a DRC accelerator that exploits the parallelism inherent in the design rules.

1.3 A Software Methodology for Multiprocessing

If an existing program can be partitioned into tasks that are each sufficiently time-consuming compared to the time it would take to move the task's input and output data between processors, then an existing local area network may be effectively used as a multiprocessor to run that program. This is the case with DRC, and is likely to be the case with Digital's circuit extractor and mask-making software. If several processors share a common file system, such as in VAXclusters, then the input/output size constraint can be removed[4].

[4]VAXcluster is a trademark of Digital Equipment Corporation

Parallelism is a cost-effective strategy for accelerating VLSI CAD tools. No special-purpose hardware is needed. It is possible to use a small number of general purpose computers as a multiprocessor. Thus the utility and the expense of an n-processor system can be shared with those who need more serial processing machines. Parallelism can be combined with higher-speed general purposes computers and with higher-performance software.

Many CAD tools have some parallelism, due to the nature of VLSI. So a hardware investment made toward faster DRCs may also pay off by accelerating simulations, mask preparations, and circuit extractions. Another example of exploitable parallelism is compiling and linking a large software system.

This thesis describes a software system called *EPIC* (Exploiting Parallelism In CAD) that controls the parallel execution of any software system that exhibits a restricted class of parallelism. The necessary characteristics of the computational environment and the program to be accelerated are as follows:

- The program must be partitioned into discrete tasks.
- Each task must be individually callable from the operating system.
- All communication between tasks must be done through disk files.
- Unless different computers can share the same file system, the time it takes to execute an individual task must be greater than the time it takes to transfer the files that it reads and writes.

1.4 Chapter Outline

Chapter 2 describes previous work in accelerating CAD tools. This includes efforts to use parallelism and hardware acceleration to speed up design rule checking and simulation. The primary motivation for this thesis, Parallel ECAD DRC, is described.

Chapter 3 describes the theory and implementation of *EPIC*. The more interesting features, such as task scheduling, are described in detail.

Chapter 4 describes the application of *EPIC* to various problems, such as design rule checking, circuit extraction, and compiling and linking programs. Optimistic predictions are made for the speed-up of each application. The speed-up factors are determined for

several experimental runs of each application. The experimental results are then compared to the optimistic predictions.

Chapter 5 concludes the thesis with a summary of the work reported and suggestions for future research.

Appendix A contains a user's manual for running *EPIC* with ECAD DRC.

Appendix B contains graphical representations for the data dependency graphs for several applications.

Appendix C contains the raw data for the experimental runs, including a table of statistics and a graphical representation of the task assignments for each slave.

Appendix D contains all the messages *EPIC* sends for control communication. They effectively define the architecture of the software behind *EPIC*.

Chapter 2

Background

2.1 Previous Work in Parallelism for VLSI CAD

A substantial amount of research has recently been devoted to the area of parallel simulation. Papers have been published on the parallel acceleration of several classes of simulators, including relaxation based simulators such SPLICE [Newton, Sangiovanni-Vincentelli 1983,Deutsch, Newton 1984], and event based logic simulators such as RSIM [Terman 1983,Arnold 1985].

Until very recently, not much had been published on parallel design rule checking. In the past year, there has been more activity [Bier, Pleszkun 1985,Nielson 1986]. [Bier, Pleszkun 1985] seeks to exploit the geometric locality of VLSI layouts by dividing the layout into vertical slices, checking each slice on a separate processor, and merging the error reports together. This approach could suffer from a large number of missed errors and false errors at the borders of the slices. At some cost in redundant computation, these problems can be eliminated by dividing the chip into slices that overlap by at least one *maximal design rule interaction distance* (DRID). Errors reported within one DRID of the border of a slice are filtered out in the merge phase as potential false errors. If they are real errors, they will be flagged during the check of the neighboring slice.

This strategy was not tested on a real multiprocessor, but based on statistics gathered during serial runs, a speedup of 8:1 was predicted for 14 processors. As communications costs are small, this figure may be realistic. It is not reasonable to expect this

algorithm to offer a linear speed-up factor, since the computational overhead of processing overlapping slices and discarding errors at the borders will grow with the number of processors.

The data partitioning algorithm has the desirable property of having its potential parallelism scale as a function of complexity of the layout. If there is no communications overhead, then we should then be able to use more processors to hold the DRC execution time constant as the circuit grows. Unfortunately, the overhead of checking overlapping regions of the chip and removing false errors from the reports may reduce the potential speedup significantly, and prevent the number of processors from being profitably scaled with the layout. This thesis presents an alternative strategy that has no intrinsic computational overhead. Unfortunately, the parallelism of our technique does not grow with the complexity of the layout, but with the complexity of the rules set. Nevertheless, it promises to allow more efficient use of each processor, and therefore provide better speed-up factors for limited numbers of processors.

2.2 Previous Work in Accelerating DRC

Empirically, the time and space consumed by a design rule check has been observed to be about $O(n^{1.2})$ or $O(n^{1.3})$ where n is the number of transistors. As the number of features on a typical VLSI chip moves into the millions, DRC will become more of a bottleneck in the designers' loop.

Hierarchical DRC is one possible solution to the DRC problem, and has recently been studied extensively ([McGrath, Whitney 1980], [Whitney 1981], [Newell, Fitzpatrick 1982], [Smith, McDonald, Chang, Jerdonek 1984]). In a normal chip, many cells are defined in terms of other cells, and blocks of cells are repeated (such as in a memory). Hierarchical DRC attempts to exploit this repetition by checking only one instance of a given cell or cell block, regardless of how many times it occurs. This has the added advantage of only generating one error when a repeated cell is faulty, thus reducing the volume of error reports while still conveying the same information.

In practical applications, however, the amount of repetition is limited by various

factors, such as overlapping regions and globally routed conductivity [McGrath 1985]. Often, the advantage to be gained by exploiting the repetition is lost to the overhead of finding and re-checking the cases where a cell's boundaries are violated by other layout. Thus, while hierarchical DRC is profitable for certain chips, it is not yet a sufficiently general solution. When it is does become profitable, it can be combined with the multiprocessing DRC algorithm presented in [Bier, Pleszkun 1985], or the approach presented in this thesis.

At Hewlett-Packard, hierarchical DRC has been successfully used in practice [Hammer 1986]. Using a core of checking routines based on NCA's VDRC[1], a methodology was developed whereby the layout designer DRCs cells as they are initially layed out. The CAD system maintains a central database of cells, keeping track of whether any cell has been modified since it was last checked. When a cell is instantiated, only externally visible geometry is checked in subsequent DRCs. This system is especially effective because the cost of checking each cell is spread throughout the design process, rather than lumped together at the end. The disadvantage is that the designers must completely avoid overlapping cells with other cells and with routing.

It has been suggested that the DRC bottleneck can be eliminated by "correctness by construction" [McGrath, Whitney 1980]. This involves using layout systems that enforce the design rules at the construction phase, making it impossible to violate a design rule. Such layout systems tend to use design rules that are too simplistic, resulting in poor layout density, and thus producing slow chips [McGrath 1985]. Specifically, the corner stitching structures of Magic do not provide for 45° angle geometries [Taylor, Ousterhout 1984]. Modern industrial design efforts require this capability.

Advancements in the algorithms behind design rule checking have improved the overall performance ([Wilcox, Rombeek, Caughey 1978], [Arnold, Ousterhout 1982], [Chapman, Clark 1984]). For example, Chapman and Clark outline a method for improving the performance of IBM's *Unified Shapes Checker* by using scan lines. On chips with more than 50,000 transistors, they realized a CPU-time reduction of more than 50%. This savings is substantial, but they predict that the improvement will not be sufficient to

[1]VDRC is a trademark of NCA corporation

swiftly check chips as transistor counts move into the millions.

Seiler describes a method for doing DRC's in hardware [Seiler 1985]. This method has obvious advantages. Dedicated and custom-designed hardware can do a good job of exploiting "inner-loop" parallelism. However, a working prototype was not produced. Until the introduction of a production quality hardware DRC accelerator, it may be more timely to increase performance by augmenting the existing CAD software.

2.3 Motivation: Parallel DRC

Digital Equipment Corporation's primary motivation for supporting this project was to produce a system that runs parallel ECAD DRCs. The key observation that motivated our strategy is that a design rule check does not entail the execution of a single algorithm, but instead involves the sequential execution of many computationally independent algorithms. More specifically, DRC is a sequence of rules, such as the following:

1. POLY-DIFF SPACING $\geq 1\lambda$
2. POLY-POLY SPACING $\geq 2\lambda$
3. POLY WIDTH $\geq 2\lambda$
4. GATE OVERLAP $\geq 2\lambda$

Conceptually, there is no data dependency between these rules. Therefore, each rule can be executed independently by a separate processor. That is not very efficient, because there are often intermediate computations which contribute to the checking of a rule, and the results of these computations are often used in the checking of more than one rule. We would like to do these computations only once, and share the results among all those processors that need them.

These intermediate computations are explicitly listed in the ECAD *rules file* that is used to control each DRC run. The rules file is essentially a computer program written in a language especially tailored for DRC. The language has statements that do operations on the various layers of the chip, such as polysilicon and diffusion. Some statements do logical operations such as the pixel-wise AND and OR of two layers, producing new layers. Other statements do spacing or width checks on a given layer at a given tolerance, producing

error reports. A side effect of the execution of the program is that all the rules are checked. As a final step, all the error reports are appended into a summary file, and the geometry of the errors is depicted in an "error cell" layout that can be read into the layout editor.

Each statement in the rules file can be mapped directly onto a sequence of operating system commands that cause the statement to be executed. The input and output file names can be extracted from the text of the rules file statement. By comparing the input file names of one statement to the output file names of another statement, we can determine whether there is a data dependency between the execution of those two statements. In this manner, we can build a data dependency graph from the rules file, with the information about how to execute each statement stored at each node.

The data dependency graph has a set of *roots*, or nodes whose input files are part of the input data to the whole task, rather than outputs of another node. The number of roots is generally equal to the number of different VLSI layers for the particular process technology. The computation must begin with the roots. How the computation proceeds depends on the scheduling strategy, and greatly influences the performance of the whole parallel execution.

2.4 Scheduling strategies

The following approach is taken by ECAD in their marketed version of `Parallel DRACULA`[2] [Nielson 1986]. It requires a multiprocessor with a shared filesystem, such a VAXcluster; it won't run on a local area network. This implies that it won't suffer file transfer overhead. It also depends on the scheduling facilities built into the multiprocessor. When submitting a non-interactive (batch mode) job to a VAXcluster, the VAX/VMS operating system[3] determines which processor is most responsive, and assigns the job accordingly.

The first step is to divide the data dependency graph into sections, as shown in Figure 2.1. Each section contains all the nodes in the graph that have a given distance

[2]Parallel DRACULA is a trademark of ECAD corporation

[3]VMS is a trademark of Digital Equipment Corporation

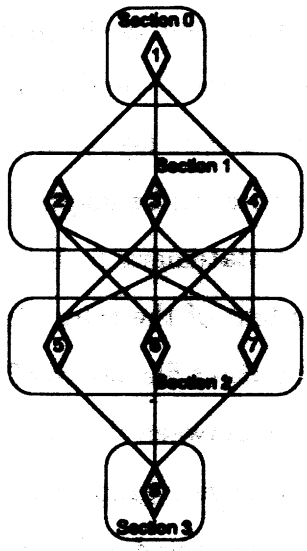


Figure 2.1: Sectioned Data Dependency Graph

from the roots of the graph, where distance is simply the number of nodes one must pass through to arrive at the destination. For example, the roots comprise a section whose distance is zero.

The approach proceeds by executing each section one at a time. Every node in the current section must be completed before any node in the next section can start. This guarantees that the data dependencies will not be violated. It is also very easy to implement. The parallel execution is controlled by a command file.

There are at least two substantial drawbacks to this method. At the end of the execution of each section, the faster processors will remain idle while the slower processors finish up their tasks. At best, this severely limits the number of processors that can be profitably used. At worst, it implies that a processor that becomes severely overloaded or hung (for example, due to another user) after a task has been assigned to it is guaranteed to block the execution of the DRC. Another drawback to ECAD's method is that the requirement that it be run on a VAXcluster is inconvenient; Digital would like to run parallel DRCs on VAX computers that are not VAXclustered together.

By more cleverly using the data dependency graph, we can increase the potential parallelism substantially, keeping each processor busy nearly all the time, thereby enjoying increased performance compared with ECAD's method. To do this, we need to layer a

sophisticated parallel scheduling and execution system around ECAD DRC.

Unfortunately, ECAD DRC represents a true "black box" abstraction: the source code is not for sale. Furthermore, its user interface was not designed to be used as an interface to another program. Though the command interface to any given version of the software may be sufficiently documented, it is not guaranteed to remain stable over time.

A system that is layered around such an inaccessible piece of software must be written to be resilient to change in the interface to that software. Also, it must not depend on specific restrictions that may only apply to the current version of ECAD. One such restriction is that that each line in the ECAD rules file corresponds to a task with no more than two input and output files. It is conceivable that this restriction could disappear at the whim of an ECAD engineer.

The way to achieve this resiliency is to try to choose a model for the computational structure of ECAD's DRC that is general enough to be adaptable to any conceivable change that ECAD might make. The following chapter describes how this is done.

Chapter 3

EPIC: A general method of exploiting parallelism

This chapter describes the implementation of a software system called *EPIC* (Exploiting Parallelism In Cad). *EPIC* provides a mechanism for controlling the parallel execution of existing software that exhibits a specific class of intrinsic parallelism. *EPIC* was written in PL/I for the VAX/VMS operating system, and runs on any number of VAX computers connected by DECnet or in a VAXcluster[1]. No special hardware configurations are required. Between the *EPIC* kernel and the preprocessors provided for running ECAD DRCs and `Makefiles`, 8751 total lines containing 5548 PL/I source statements were written.

3.1 Dividing the job

The system described here provides a mechanism for running Parallel DRC by solving the more general problem of how to control the parallel execution of any program that can be externally divided into a finite set of *tasks*. We define *task* as a unit of computation that can be executed using a sequence of standard operating system commands (such as DCL commands, for the VAX/VMS operating system). Each task has a known, finite set of inputs and outputs, each of which is a disk file. These tasks are explicitly specified in

[1]DECnet is a trademark of Digital Equipment Corporation

the manner of Figure 3.1.

```
task "split"-
    /input=(chip.data)-
    /output=(left.data, right.data)-
    /dcl=("$splitter chip left,right")

task "left"-
     /input=(left.data)-
     /output=(left.errors)-
     /dcl=("$drc left")

task "right"-
     /input=(right.data)-
     /output=(right.errors)-
     /dcl=("$drc right")

task "merge"-
     /input=(left.errors,right.errors)-
     /output=(chip.errors)-
     /dcl=("$merge left,right chip")
```

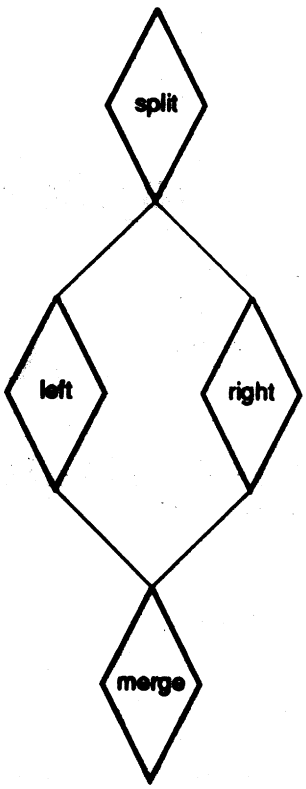


Figure 3.1: Sample Task Description List and Data Dependency Graph

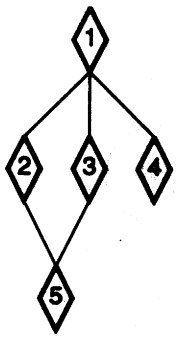
The strategy we will use for Parallel DRC involves distributing the design rules to the various processors. Each processor applies its subset of the rules to the whole chip. But $\mathcal{EPIC}$ is not restricted to this form of parallelism, which is called *instruction partitioning*. As hinted at in Figure 3.1, $\mathcal{EPIC}$ is well suited to *data partitioning*. The multiprocessing DRC scheme proposed by [Bier, Pleszkun 1985] could easily have been implemented with $\mathcal{EPIC}$.

A simple way to determine whether or not we can expect $\mathcal{EPIC}$ to be able to enhance the performance of a given program is by comparing the sizes of the input and output files of each of its tasks with the time it takes to execute those tasks. If the execution time is far greater than the amount of time it takes to transfer the input and output files between the various processors, then the potential exists for substantial throughput improvements using $\mathcal{EPIC}$. Of course, if all of the processors share a single file system,

then data communication becomes less of a bottleneck, and the restriction can be relaxed.

The extent of the parallelism, and hence the potential for throughput enhancement, is further limited by the data dependencies within the task list. By comparing the inputs and outputs of each task, we can generate a data dependency graph, as shown in Figure 3.1.

In Figure 3.1, the potential parallelism is limited to a maximum of two processors. If we assume that each task takes one "tick", then by using two processors we can do the job in 3 ticks, whereas we would need 4 with a single processor. Due to the data dependencies, a third processor couldn't be used at all. So we say the parallelism has a *maximum extent* of 2.



Processors	Two		Three		
Processor	A	B	A	B	C
time = 1	1		1		
time = 2	2	3	2	3	4
time = 3	5	4	5		

Figure 3.2: A More Interesting Data Dependency Graph and its Execution

The most obvious way to try to determine the extent of parallelism is to find the width of the widest row in the graph. This worked in Figure 3.1, and clearly having that many processors would yield the fastest possible execution time. However, by assuming that each task executes in one tick, we can do just as well using fewer processors. Consider the data dependency graph in Figure 3.2. The maximum extent of parallelism is now 3, since we can keep 3 processors busy at *time* = 2. But the minimum extent of its parallelism is 2, because "4" can be executed by the second processor during the third tick, while the first processor is executing "5". *EPIC* tries to optimize task scheduling in this manner so it can get the most performance out of the available processing power.

3.2 Multiprocessing on a Local Area Network

The computational model I have selected for parallel processing is not unlike the dataflow model. Of course, the size of each atomic computation is somewhat smaller in dataflow, so the capacity for incurring overhead from controlling the computation is also smaller. Hence, I use a significantly different approach to controlling the computation in *EPIC*.

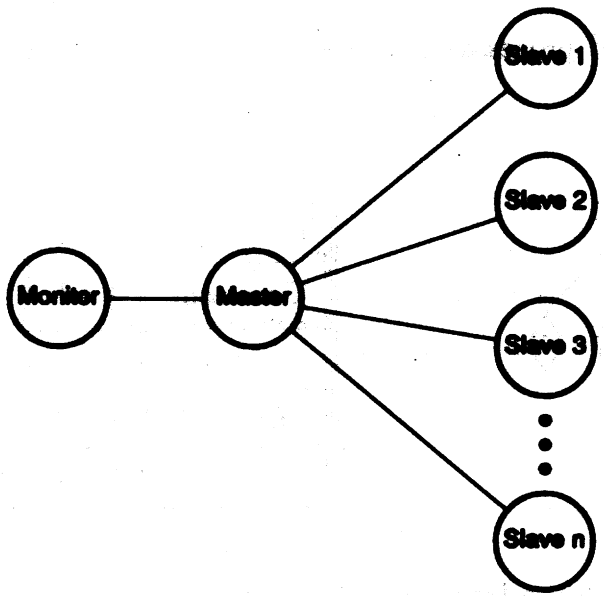


Figure 3.3: Star Network Topology

Ethernet[2] technology is used as the physical layer beneath the DECnet protocol in DEC's local area networks[3]. Ethernet is essentially a coaxial cable that connects each node on the network. A processor sends a message by broadcasting it over the cable. Each processor receives all the messages and scans them for the ones that are addressed to it. Conceptually, an Ethernet can provide the basis for a variety of software network topologies. The topology *EPIC* uses is a *star* network, as shown in Figure 3.3. The processor at the center of the star, called the *master*, is responsible for controlling the whole execution. One of the processors on the points of the star is used to provide a user

[2]Ethernet is a trademark of Xerox Corportion

[3]DEC is a trademark of Digital Equipment Corporation

interface for the master. An interactive program called MONITOR is run on this computer to allow a human to control the execution. The remaining processors at the points of the star, called *slaves*, are responsible for executing whatever tasks the master assigns, and for transferring the appropriate input and output files.

There were several specific engineering factors considered in the decision to use a star network topology. The programs we intend to run in parallel tend to have irregular computational structures. Their data dependency graphs take on arbitrary shapes, forcing us to spend considerable effort trying to keep each processor busy. This is further complicated by the computational environment in which we run. Each processor is a time-sharing computer, and while we expect that *EPIC* would only be run when it wouldn't be competing for cycles, we can't let a loaded processor slow down the rest of the computation. Thus a fragile task scheduling strategy would involve allocating each task a to specific processor before the computation begins. A more robust task scheduling strategy is to dynamically assign computable tasks to available processors, so a relatively slow processor will execute proportionally fewer tasks. Fortunately, since each task takes so much time, we can afford to incur some computational overhead figuring out the best strategy for assigning tasks to processors. A good way to do that is to have one processor running a master program that has total control of the computation.

As it turns out, the master does not take very much CPU time once some initial preprocessing has been done. Most of the time, it's just waiting for a slave to indicate that it is finished with its task. The short burst of CPU time it needs to figure out which task gets allocated to the free slave is small compared to the time it takes the slave to finish the task. Experimentally, I have determined that the master can efficiently share a processor with a slave.

It is enlightening to look at an example which is not conducive to a star network topology. In regular parallel structures, it is easy to predetermine the best way to allocate processors to tasks. Systolic arrays are one way of executing such computations. Central control of each processor in a systolic array is undesirable, since there is typically a large amount of communication between neighboring processors, but very little other communication. It is better to have each processor know precisely how and when to talk to its

neighbors than to have one processor take responsibility for relaying all communication from the sender to the receiver. Due to the extreme volume of information passing through it, that processor would then be a severe bottleneck in the computation.

Another class of applications that are not well suited to the *EPIC* model of computation are those where it is not clear at the start of the program exactly what computation will occur. The task breakdown is done at run time, rather than "compile" time. If this is the case, *EPIC* will not be able to efficiently schedule the tasks.

A good example of this is Parallel RSIM [Arnold 1985]. It uses a master-slave star network configuration as its multiprocessor, but there is no finite set of tasks from which to generate a data dependency graph, since RSIM is an *event based* simulator. A change in the value of a node in the circuit causes a simulation of the surrounding devices. If this simulation causes other node values to be changed, then the devices connected to those nodes are simulated as well. This propagation continues until the network settles. There is no way to predetermine exactly what computation will occur when a given node changes. Instead, before any simulation occurs, Parallel RSIM exploits functional locality in the circuit by partitioning it and sending one section to each processor. The various sections are simulated independently until a value on a shared node changes. The processor that changed the node then sends a message to other processors that share the node indicating the new value and the simulated time when the change occurred. *EPIC* is not equipped to deal with this sort of computation. It needs to know about each task in the problem before it can begin.

3.3 Software Architecture

EPIC is composed of three separate programs, MONITOR, MASTER, and SLAVE. Each is run in a separate process. These processes can be on different computers. Normally, one would run the MONITOR, MASTER, and one SLAVE all on one processor, since MONITOR and MASTER take almost no CPU time during the computation.

The three programs communicate by passing messages. Using VAX/VMS mailboxes and the DECnet interprocess communication protocol, a message passing subsystem was

developed. It provides a uniform procedural interface to allow programs to easily handle a variety of asynchronous events, such as subprocesses, timers, multi-client interprocess communication, and terminal I/O.

The MONITOR is the only program with which the user interacts. It allows the user to initiate and control the parallel execution, and provides a periodically updated display of the status of each SLAVE's process. For more information about the MONITOR, see the *EPIC*/DRACULA User's Manual in the Appendix.

The most interesting program is the MASTER. It is initiated by a user instruction to the monitor. The monitor creates a remote process on the master's processor, and opens up a communication channel to it using the message passing system. From that point on, the monitor is used essentially as a front end for the master.

```
open execution control file
task_list := empty-list()
while not(end-of-file) do
    read task description
    append task description to task_list
end while

for each element "t1" in task_list do
    for each element "t2" after t in task_list do
        if any of t1's outputs match any of t2's inputs then do
            t1 is a predecessor of t2
            t2 is a successor of t1.
        end if
    end for
end for
```

Figure 3.4: Algorithm for Generating Data Dependency Graph

The first thing the master does is read the *execution control file*, which contains all of the task descriptions. This is all the master needs to know about the particular application being run (e.g. DRC or Makefile). Recall that a task description indicates all the input and output files, as well as the sequence of operating system commands that

run that task. Completing each of the tasks in the execution control file is equivalent to running the application. The master is charged with distributing those tasks among all the available processors so as to minimize the total execution time. The strategy it uses requires the generation of a data dependency graph from the execution control file. The algorithm used is presented in Figure 3.4.

The master maintains the database of slaves. A slave is created in response to a request that the user gives to the monitor. The monitor relays the request to the master, and just as the monitor created the master, the master uses the message passing package to create a remote process on the slave's processor, and establish a communication channel with it. The user can request a slave at any time after the master has been created. Each slave has the capacity to execute one task at a time. Hence each slave can be in one of two states: "busy" or "idle". An *idle slaves* list and a *busy slaves* list are maintained throughout the computation.

The computation begins with the *roots* of the graph. A task is a root if it has no predecessors. So initially, the roots are placed on a *ready queue*. A task on the ready queue is said to be computable. When all of a task's predecessors are completed, it is placed on the ready queue.

3.4 Task Scheduling

```
do while there are tasks left to execute
   do while (the ready queue and the free slave list aren't empty)
      assign a slave to a task
   end while
   wait for a slave to finish or a "create slave" message
end while
```

Figure 3.5: A Skeleton for a Task Scheduling Algorithm

With a list of free slaves and a ready queue, the master can begin the computation. The basic structure of the algorithm used to control the execution is presented in Figure

3.5.

Each statement in the algorithm corresponds to a substantial amount of programming. For example, "`wait for a slave to finish`" implies (among other things) checking the finished slave's task's successors to see if they are now computable. One statement which implies a good deal more is "`assign a slave to a task`". If the number of free slaves is greater than or equal to the number of tasks on the ready queue, then we can assign any of the computable tasks to a slave, since each of the tasks will be assigned before the loop falls through to the "`wait...`" statement. Unfortunately, we are not usually provided the luxury of being guaranteed more slaves than tasks on the ready queue. The choice of which task to assign must be made carefully, because it can have fairly profound effects on the speed-up factor of the parallel execution.

A bad algorithm for choosing tasks can result in data dependency bottlenecks. An optimal algorithm for choosing tasks is NP-Complete [Mehrotra, Talukdar 1982]. We present here a heuristic for choosing tasks that has been observed to perform optimally under most conditions. It requires a preprocessing step that has time complexity $O(n^2)$, where n is the number of tasks.

The first step toward discovering this heuristic is to identify the goals of the whole parallel execution system, and how the task scheduling algorithm must try to help achieve these goals at minimal cost. The main objective is to minimize the *real* time (as opposed to the CPU time) needed to execute a set of tasks, given a finite number of processors. To do this, the task scheduling strategy must keep all processors busy as much of the time as possible. Each processor will be always be busy as long as there are computable tasks. So a good subgoal is to keep the ready queue as full as possible. Executing a task that has no successors (called a *leaf*) will clearly make no progress toward replenishing the ready queue. Executing a task that has many successors will clearly make some progress towards that goal, but it's still not clear how one should measure the immediacy of the need to execute a given task. What we do know is that we are interested in the characteristics of the subgraph rooted at that task's node in the data dependency graph.

To help focus our attention on the right characteristics of a task's subgraph, we observe that the limiting factor of a computation is the longest path through the data

```
for each task "T1" in task_roots do
    compute height_of_task(T1)
end do

height_of_task(T1):
    if T1.height is set then return(T1.height)
    sub_height := 0
    for all successors "T2" of task T1 do
        sub_height := max(sub_height,height_of_task(T2))
    end for
    T1.height := estimated_execution_time(T1) + sub_height
    return(T1.height)
end height_of_task
```

Figure 3.6: Algorithm for computing height of all tasks

dependency graph. No matter how many processors are available, the overall execution time will never be less than the sum of the execution times of all the tasks along the critical path. This sum is called the *height* of the graph. As the computation progresses, we seek to chip away at this critical path in support of our meta-goal, which is to minimize the total execution time. So the conclusion of this intuitive argument is that we should give top priority to tasks which lie on the critical path. The appropriate quantitative measure is the height of the task's subgraph. Using the algorithm presented in Figure 3.6, we can compute the height of each of the n tasks in $O(n)$ time.

Using the height as a priority scheme for each task does not provide very much resolution. In the data dependency graph generated from a sample design rule checker's execution control file, the estimated execution time of each task is 1, and the heights of all the tasks are integer values between 1 and 8. But there is more information in a data dependency graph that is intuitively related to how critical each particular task is. In particular, the total number of tasks that directly or indirectly depend on a given task is relevant. In a sense, it is the measure of the total *fanout* of a particular task. It is equal to the *size* of the task's subgraph. The algorithm in Figure 3.7 computes the size of n tasks in $O(n^2)$ time. In practice, this has been an acceptable penalty to pay for the

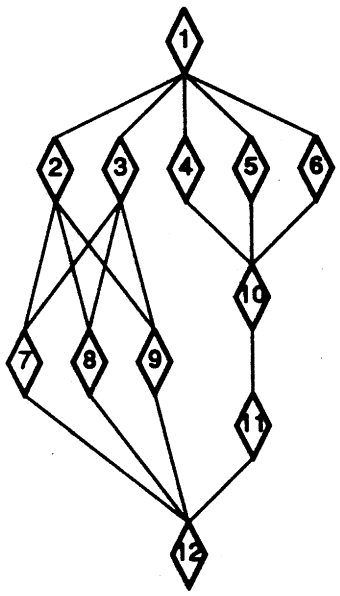
```
for each task "T1" in task_list do
    clear_examined(T1)
    T1.size := find_size(T1)
end for

clear_examined(T1):
    T1.examined := FALSE
    for each successor "T2" of T1 do
        clear_examined(T2)
    end for
end clear_examined

find_size(T1):
    if T1.examined=TRUE then return(0)
    T1.examined := TRUE
    size := 0
    for each successor "T2" of T1 do
        size := size + find_size(T2)
    end for
    return(size + estimated_execution_time(T1))
end find_size
```

Figure 3.7: Algorithm for computing the size of all tasks

more accurate scheduling capability. In the case of design rule checking, the penalty is insignificant compared to the time spent doing the DRC.

Empirically, we verify our suspicion that the height of a task's subgraph is a better measure of its priority than the size of the subgraph. The way to compare the performance of the heuristics is by simulating a parallel execution under the assumptions that each task takes unit time and that there are no communication costs. We then depend on real experiments to back up the results of the simulation. Figure 3.8 shows the parallel execution simulations of a data dependency graph using four processors. While this is only one example, by running the two simulations in your mind, hopefully you will gain intuition that lends support to our empirical observations.

Now we have two numbers associated with each task: a height and a size. We use



Heuristic	Height				Size			
Processor	A	B	C	D	A	B	C	D
time = 1	1				1			
time = 2	2	4	5	6	2	3	4	5
time = 3	3	10			6	7	8	9
time = 4	7	8	9	11	10			
time = 5	12				11			
time = 6					12			

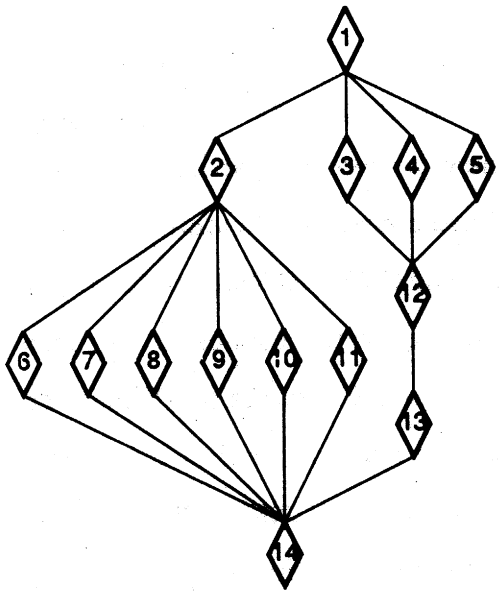
Figure 3.8: A data dependency graph and its execution using *height* and *size*

these as keys to keep the ready queue sorted: first by height and then by size. With the most crucial tasks at the front of the queue, the task scheduling strategy is complete. The $O(n^2)$ operation to find the sizes is run only once before the start of the run. Typically, for design rule checker's data dependency graphs, there are fewer than 200 nodes, and the total time spent on the processing step in the beginning is less than 30 seconds. Once the height and size of each node is computed, they are used to dynamically guide the scheduler in assigning the most urgent task to a slave whenever that slave finishes its previous task.

The strategy performs optimally in most cases. After creating data dependency graphs of various shapes and sizes and simulating each one with a varying number of processors, only one example was found in which the height/size heuristic did not perform optimally: it took seven time units instead of six. This is illustrated in Figure 3.9.

3.5 Communications

There are two major obstacles blocking us in our pursuit of a linear speed-up factor. The first is the challenge of keeping each processor busy as much as possible. For the class of applications that we wish to accelerate, the task scheduling strategy introduced in the previous section does an adequate job. While testing $\mathcal{EPIC}$'s application to an industrial



Heuristic	Optimal			Height/Size		
Processor	A	B	C	A	B	C
time = 1	1			1		
time = 2	2	3	4	3	4	5
time = 3	5	6	7	2	12	
time = 4	8	9	12	6	7	8
time = 5	10	11	13	9	10	11
time = 6	14			13		
time = 7				14		

Figure 3.9: A task scheduling trial simulation where height/size heuristic is suboptimal

design rule checker, the task scheduling behaved well. This is discussed in more detail in the following chapter.

The next challenge is that of minimizing the communications overhead. Since $\mathcal{EPIC}$ was designed to run on a loosely coupled multiprocessor, communications is fairly expensive. In $\mathcal{EPIC}$, there are two flavors of interprocessor communication: *control* and *data*. The mechanism used for these two forms of communication is different.

3.5.1 Control Communication

Control communication is accomplished using the message passing package developed for $\mathcal{EPIC}$. It is based on the VMS/DECnet task-to-task communications protocol [VMS 1985]. From a programmer's point of view, one simply opens a channel using a file specification of the form:

```
node"username password"::"task=commandfile"
```

This causes a message to be sent on the Ethernet to `node`, requesting that a process be created for `username`, and that that process run `commandfile`. The `commandfile` on node should then open a channel (or invoke a program that opens a channel) using a file

specification of the form SYS$NET:. By writing to and reading from these channels, the processes can send messages to each other.

The above mechanism provides the necessary channels of interprocessor communication in the case where one process wants to create a new process on another processor and then talk to it. If two existing processes want to establish a channel of communication, then another strategy is used. When an *EPIC* program (MONITOR, MASTER, or SLAVE) is run, it creates a VAX/VMS *mailbox* [VMS 1985]. A mailbox contains a global buffer into which any process that knows how to find the mailbox can write a message. When the program creates the mailbox, it assigns a *logical name* to the mailbox so that other processes can find it. By convention, MONITOR uses the logical name EPIC$MONITOR, MASTER uses EPIC$MASTER, and SLAVE uses EPIC$slave-name. Therefore, within a single logical name space, there can only be one monitor and one master, and each slave name must be unique. Thus when one program wants to contact another, it opens up a channel to the appropriate mailbox (for example, monitor opens up a channel to node::EPIC$MASTER:) and initiates a conversation. By reading to and writing from that channel, the two existing programs can communicate.

3.5.2 Control Communication Requirements

The MASTER program communicates with any number of slaves, in addition to the monitor. The "wait for a slave to finish or a "create slave" message" line in Figure 3.5 requires the use of an I/O subroutine that is not provided by VAX/VMS or the PL/I run time library. At some level in the code, there must be some statement that reads a record from any of several I/O channels, returning the message and the channel number of the first channel to send a record. In order to provide this functionality, an asynchronous read request is left pending on each channel using the VAX/VMS system service SYS$QIO. When the channel responds, a subroutine specified as a parameter to SYS$QIO is called at the interrupt level. This subroutine is called an *asynchronous system trap* (AST). It is the AST's responsibility to append the message that was received onto a queue of messages, set a *global event flag* that indicates that a message was received, and requeue the SYS$QIO.

When designing a large, complex system such as *EPIC*, the existence of ASTs poses a tough software engineering problem. Since ASTs execute at a higher priority level than mainline code, we cannot generally assume atomicity in a sequence of operations that updates a data structure. For example, if one is in the process of deleting an element from a doubly linked list, and an AST is triggered that modifies that list, the list could be left in an inconsistent state. In short, ASTs are a power tool, and when power tools are used carelessly, they can kill[4] (or at least cause endless hours of debugging).

There are two strategies for ensuring harmony in data structures that are shared between mainline code and AST routines. The first is to disable AST interrupts with a system call wherever synchronous code accesses a data structure that it shares with AST routines. The disadvantage of this approach is that while interrupts are disabled, the user process can't respond to messages it receives from other processes. If the sending process uses asynchronous WRITEs, then it could queue up an arbitrary number of messages while the receiving process remains in "disabled-interrupts" mode. Depending on the buffer size parameters selected by the system manager of the computer facilities, the buffer could overflow. If the sending process uses synchronous WRITEs, meaning the WRITE statement doesn't return until the reader's AST has been triggered, then the sender will be delayed until the reader's interrupts have been re-enabled. In this case, if the reader has interrupts disabled while waiting for the "message-received" event flag to be set, a deadlock could occur.

The other strategy is to carefully code the routines that access shared data structures so that they are *never* in an inconsistent state. It is possible to do this for singly linked lists, but not doubly linked lists. This is a fairly serious restriction, since it is difficult to delete an arbitrary element from the middle of a singly linked list. One way around this is to share only a singly linked list between mainline and AST-level code. The only operation ASTs get to perform is appending to the tail of the list. All that the mainline code does with that list is remove messages from the head of the list and place them in a more versatile data structure that is safe from ASTs.

The message passing facility uses a compromise between these two approaches.

[4] "Power tools can kill" is a maxim credited to Brian Reid of Stanford University

Since *EPIC* requires both the capability of reading a message from the first channel and the capability of reading a message from a specific channel, the shared list has to support the ability to scan through the list and remove the appropriate message. This could have been implemented using the latter strategy, but the following strategy was more convenient to code, and in practice did not suffer noticeable performance penalties. It shares only one structure between AST-level routines and mainline routines: a doubly linked list structure. Interrupts are only disabled for the time it takes to find and remove the appropriate message. In practice, finding the appropriate message in the list was not expensive, since the list generally had less than 10 messages. Removing the message amounts to moving a few pointers. The key to making the "disable-interrupt" strategy work is to avoid doing any I/O calls while interrupts are disabled.

The primary motivation for writing the message passing package was to eliminate all asynchronous code from the rest of *EPIC*. In addition, the message passing package provides a uniform synchronous procedural interface for handling asynchronous communication between a process p_1 and the following *entities*:

- Independent processes that p_1 created on another node
- The process that created p_1 from another node
- An independent, already existing process on another node
- Subprocesses created by p_1
- The terminal attached to p_1
- Timers created by p_1

The single most significant function it provides is that of reading from the first of any of the entities that sends a message.

3.5.3 Data Communications

Recall that *EPIC* is a shell around an existing software system. *EPIC* divides the execution of that software into tasks. Each of these tasks communicates using disk files. While the problems to which we are restricting ourselves do not use extremely large disk files, experience has demonstrated that the performance improvements we reap through parallelism are most severely limited by the speed with which we pass data between master

and slave. The message passing facility described above is not as fast as it could be, since considerable effort is spent providing the functionality required by *EPIC*. Hence, if we were to use the message passing facility for data communication, we would suffer from suboptimal performance. In addition, the data contained in the input and output files may be represented using any of the file record structures available in VAX/VMS. The message passing facility is restricted to dealing with character strings. The standard VAX/VMS interprocessor file copying commands provide the appropriate functionality at the fastest possible speed.

To copy a file from one VAX/VMS system to another, an interactive user would type

```
$ COPY node1"username1 password1"::device1:[directory1]file1.ext1 -
$_      node2"username2 password2"::device2:[directory2]file2.ext2
```

Naturally, if you were typing this on node1, you would omit the accounting information for it. In general, VAX/VMS allows the inclusion of a node specification (with accounting information) in any file specification. Opening a file with an account specification causes a process to be created on the remote node using the supplied username and password. That process efficiently handles the I/O calls made to the channel. The remote process creation is functionally transparent to the user, except for the time overhead involved.

The way *EPIC* executes the VAX/VMS `COPY` command is by using the message passing facility. The facility provides a call that creates a subprocess and keeps it around. Sending a message to the subprocess causes the text of the message to be interpreted as a VAX/VMS command. When the command finishes, a message is "sent" from the subprocess to the main process. This way, the main process can be doing other things while the subprocess is executing the command.

Each slave is responsible for bringing its task's input files from the master's filesystem to its own, and for sending back the output files when a task is completed. Buffering all the data files on the master is obviously less efficient than having each slave transfer its task's input files directly from the slave that generated them. *EPIC*'s approach has as much as twice the file transfer overhead has the optimal approach. The reason

EPIC buffers all data files at the master is so that if a slave's processor crashes, then its work won't be lost. In the class of problems for which *EPIC* was designed, the cost of re-executing one task may be greater than the total cost of all the file transfers for the execution of every task.

EPIC spends some effort trying to minimise the number of file transfers. The master keeps a database of all the files that reside on each slave's filesystem. Whenever a task is assigned to a slave, it is told which of the input files it already has, so the slave can suppress the `COPY` command. The effectiveness of this strategy is further enhanced by modifying the task scheduling algorithm to take into account what input files for each computable task are already resident on a free slave's filesystem. Specifically, the ready queue is composed of a list of *task groups*. Each task in a given task group has the same height, but varying sizes. The groups are arranged in decreasing order of height, and the tasks within each group are sorted in decreasing order of size. When a slave becomes free, the first task group is scanned to find the task that will require the fewest file transfers to execute. Thus the task scheduling strategy is based on ordering the ready queue by three different characteristics of each task:

1. The height of the task's subgraph (computed once)
2. The number of input files that the slave already has (computed on the fly)
3. The size of the task's subgraph (computed once)

3.6 Fault Tolerance

When the word "timesharing" is mentioned to someone who has recently survived an undergraduate Computer Science curriculum, the image that first enters his mind is that of an overloaded CPU. *EPIC*'s dynamic task scheduling algorithm insures that a relatively heavily loaded processor will be assigned proportionally fewer tasks. Another "timesharing" flashback is that of the downed computer. In those days, when the CPU was down, it was of course no longer possible to get any useful work done (except maybe a trip to the vending machine). With distributed computation, if one processor goes down, the execution should gracefully continue with degraded performance. By outlining a typical

scenario, the need for this requirement will gain more substance. Assume *EPIC* is being used to accelerate the DRC of a chip that might ordinarily take several days on a single VAX 11/780 computer. Ten VAX computers are being used to (hopefully) finish the DRC overnight. If one of them crashes (or is brought down for preventive maintenance), *EPIC* ought to continue the computation at 90% of its former speed. If *EPIC* gives up its unmanned computation, the layout designer may fall behind a whole day, assuming the ten VAX computers will be far too loaded for long non-interactive jobs during working hours.

Giving *EPIC* the capability to handle crashed slaves is fairly straightforward. The scheduler doesn't statically prepartition the set of tasks, it just assigns priorities to them so they can be easily assigned to slaves on the fly. If the message passing facility detects that a slave crashes while it is running a task, that task is placed back in the ready queue according to its priorities. If the slave completed any tasks before crashing, the output files are buffered in the master's file space, so the work won't have to be redone.

At any time during the course of a parallel computation, the user can go into `MONITOR` and create another slave. Again the dynamic task scheduling algorithm makes it easy. The new slave is added to the master's slave database, and (recall Figure 3.5) is immediately assigned a new task. Thus if the user is watching when a slave crashes, then when the machine is brought back up, the user can restart the slave process.

A predecessor to *EPIC* called `PDRC` (Parallel Design Rule Checker) experimented with a mechanism to periodically probe a crashed slave's processor to see if it had come back up [Marantz 1984]. When the processor responded, `PDRC` would automatically regenerate the slave. This worked well most of the time, but became very frustrating while debugging. If a slave was misbehaving for any reason, terminating the process would be futile, since `PDRC` would immediately sense that the processor was still up, and would create the slave again. Nevertheless, this functionality should eventually be brought into *EPIC*.

Currently, *EPIC* is not capable of continuing a computation if the master's processor crashes. It is, however, capable of restarting the parallel execution where it left off. After the master first reads the execution control file, it goes through a process of

```
check_tasks := task_roots

while "check_tasks" is not empty
    T1 := first element of check_tasks
    remove T1 from check_tasks
    if all of T1's output files exist then
        if all of T1's output files were last revised
           after each of T1's input files then
               call task_finished(T1)
end while

task_finished(T1):
    for each successor "T2" of task T1 do
        T2.predecessors_completed := 1 + T2.predecessors_completed
        if T2.predecessors_completed = T2.predecessors then
            append T2 to check_tasks
    end for
end task_finished
```

Figure 3.10: Algorithm for determining which tasks have already been done

eliminating tasks in a manner very similar to that of Unix[5] `Makefiles` (and VAX/VMS MMS[6]). The algorithm used is presented in Figure 3.10.

For most applications, it would be sufficient to merely check for the *existence* of a task's output files in order to mark it as complete. But since it was not hard to compare the revision dates of the input and output files, and since doing so gives $\mathcal{EPIC}$ the basic functionality of make, it was implemented. Thus giving $\mathcal{EPIC}$ the functionality of make was as easy as converting the syntax of the `Makefile` to that of the execution control file.

[5]Unix is a trademark of AT&T Bell Laboratories

[6]MMS is a trademark of Digital Equipment Corporation

3.7 Error Recovery

EPIC tries to address the problem of how to proceed when a slave's subprocess fails to properly execute the task it is given. A failure of this nature is detected in one of two ways. The message passing system will return the VAX/VMS error code if a problem was detected by the program run in the subprocess. If the program is not a VAX/VMS layered product, the error code may not say very much, but hopefully even an independently written program will abort by signalling an error rather than terminating normally. ECAD DRC, for example, behaves in this manner while remaining portable to other operating systems by dividing by zero whenever a problem is detected. The other way an error is detected is by checking for the absense of any of the task's output files when the task's DCL commands are finished.

In the past, the cause of an unsuccessful task execution has stemmed from a variety of sources. Sometimes the error is a reflection of the state of the computational environment of the slave's node. Specifically, a library file or executable image could be missing from a system directory. Sometimes the error is due to a possibly transient condition on the slave's node, such as the lack of a resource needed to execute the task. Often, when one slave failed to execute a task, another was found to be capable of completing it.

The strategy implemented by *EPIC* is to put a failed task back on the ready queue, and keep track of how many times it has failed. When this number reaches a certain threshold, currently defined to be 3, the task is deemed uncomputable, and is removed from the data dependency graph, along with all the tasks in its subgraph.

For certain potential applications of *EPIC*, the cause of failure for any task is be more likely to be illegal or erroneous input files. This is most likely the case when the application is to compile and link software. If *EPIC* detects a failure in a source code compilation, it is a waste of time to try it again three times before deeming it uncomputable. The right solution is then to reduce the task failure threshold to 1. The first time a task fails, it will be removed from the data dependency graph, and the rest of the tasks will be executed normally.

Each slave also gets a counter, which is incremented whenever it fails its task and

decremented whenever it completes its task. If this counter crosses a threshold, currently defined as 2, then *EPIC* destroys the slaves on the grounds that it is a waste of time to be assigning tasks to it if its going to fail more tasks than it completes.

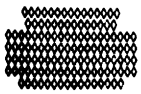
This computer resource management strategy is analogous to human resource management. A manager will assign the most responsibility to his most productive employees. *EPIC*'s strategy could be extended to use more resolution in an attempt to imitate human managers. Currently, each slave is essentially treated as an equal. Slaves are picked from the "idle slaves" list to execute the highest priority task. If there is more than one slave in this list, then the slave that has cached the greatest percentage of the highest priority task's input files gets the job. It would be interesting to implement a scheme where the slaves were ordered according to their past productivity. When selecting a slave, weights would be placed on the number of files it already has, the number of tasks it has completed so far, and the number of tasks it has failed so far.

3.8 VAXcluster Support

A VAXcluster is a group of up to sixteen VAX computers connected to a single file-system. Thus the file system looks exactly the same when you are logged into any VAXcluster member. *EPIC* supports the use of VAXclusters. By issuing a command to `MONITOR`, a user can specify a list of node names to define a VAXcluster. A database is maintained to keep track of where all the relevant data files are in the network. The structure of the database reflects the file sharing between VAXclustered nodes, and provides for any number of discrete VAXclusters:

```
network-database = list of VAXcluster-databases
VAXcluster-database = list of file-specifications
```

The `file-specification` in the `VAXcluster-database` cannot include a "`node::`" specification, but can include a device or directory. Computers that are not VAXcluster members are represented in the database as single-node VAXclusters. Thus an arbitrary environment of VAXclustered and independent nodes is supported.



Each slave entry in the master's database contains a pointer to the slave's node's VAXcluster. So if the master and a slave are on the same VAXcluster and are connected to the same device and directory, the master will know that the slave will never have to copy an input or output file. If they are on the same VAXcluster but connected to different devices or directories, the master will know to instruct the slave to use a local file transfer, and thereby save the overhead of creating the foreign process and moving the file over the Ethernet. If two slaves on the same VAXcluster share the same device and directory, the master will understand that they share the file space, and that one slave will never have to copy a file that was created or copied by the other. As of now, no advantage will be gained from two slaves on the VAXcluster with different devices or directories, unless the master is also on their VAXcluster.

Thus it is highly advantageous to have each slave on a VAXcluster running out of the same directory. If the master also uses that directory, then *there will be no data transfer overhead for those slaves.* This eliminates the single most significant bottleneck in the parallel execution.

The only legitimate motivation for running VAXclustered slaves out of different directories is if the application software has naming conflicts with temporary files it uses. Two processes running the same application program may both be trying to read and write a temporary file of the same name. By running the two processes out of different default directories, the naming problem is resolved, and $\mathcal{EPIC}$ will still run, albeit with more data transfer overhead. Another motivation is as a workaround to a bug that may exist in the application software. If a single input file is used by two tasks, and both those tasks are executed at the same time by different CPUs in the same filespace, then the second process to open the file is subject to a file locking error. In VAX/VMS, any number of processes can open a file for *read access.* But if one process opens a file for *read/write* access, any other process attempting to access that file will get a "file locked" error. The problem occurs when a program that is only interested in reading the file erroneously opens it for read/write access.

3.9 Performance Monitoring

In order to support the claims made about the effectiveness of the task scheduling and file transfer optimizations, it was necessary to generate statistics for each *EPIC* run concerning the breakdown of where each slave's time was spent. For the purposes of performance monitoring, each slave is always in one of four states, as described below:

EXEC: Executing a task.

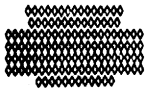
FILE: Transferring an input or output file.

IDLE: Waiting for a task to become computable, but not FREE.

FREE:
1. The execution is in its first stages, and the data dependency graph hasn't widened enough to allow all slaves to begin doing useful work.
2. The execution is in its last stages, and there are no more tasks left to execute. The execution will be finished as soon as the last slave that is executing now finishes its current task. Free slaves are not killed because if an executing slave's processor crashes, a free slave should be available to take over the task.

The distinction between "FREE" and "IDLE" is motivated out of fairness to the task scheduling algorithm. We are interested in identifying those times when a slave remains idle due to an unwise task scheduling decision. Typically, data dependency graphs have a small number of roots, but widen out quite a bit to reveal more parallelism. There is nothing a task scheduling algorithm can do to keep all the slaves busy during the execution of the roots. Additionally, at the end of the computation, it is impossible to keep each slave busy if there are no more tasks to execute. Thus the slave is classified as "FREE" if the cause of its inactivity is not a scheduling decision. "IDLE" time is what we want to keep track of to judge the task scheduling performance.

Each slave is responsible for keeping track of its own performance statistics. A performance monitoring subroutine package was built using VAX/VMS system services for keeping track of the various counters for CPU time and elapsed time. The slave uses the message passing facility to spawn a subprocess to do the file transfers and execute the VAX/VMS commands used to execute each task. Thus the SLAVE program runs in a separate process from the slave's task, and is free to spend whatever time it needs to keep track of the subprocess.



Periodically, the slave sends the master a one line summary of its progress. The master then relays this information to the monitor, which displays the information on the user's screen. The user can control the period at which each slave sends the information by issuing a command to the monitor.

At the end of the computation, each slave sends a detailed summary of its statistics, including:

- The total CPU time and elapsed time it spent in each of the four states.
- The number of tasks it executed.
- The number of tasks it failed to execute.
- The number of files it transferred.
- The number of files it avoided transferring due to file transfer optimization.
- VAX/VMS Statistics such as virtual memory usage and page faults.

The master takes each summary that the slave provides and formats it into a table. In addition, the master makes its own contribution to performance monitoring. Whenever a task is started or finished, the master notes the current time and the name of the task's slave. At the end of the run, it generates a graphical journal of how the run progressed. The graph is organized by assigning a vertical column to each slave. Each column contains a series of diamonds which represent the tasks executed by each machine. The height of each diamond is directly proportional to the time it took to execute the corresponding task. Arcs are drawn between diamonds wherever a data dependency exists between the diamond's tasks. The left edge of the graph is scored with labels indicating the elapsed time at that vertical point on the page.

There are two useful pieces of data to be gleaned from that graph. It gives us an intuitive feel for how the execution was distributed among the available processors. In addition, vertical space between the diamonds in any column indicates that that column's slave was either idle or transferring files during that time. The slope of the arcs ending at the lower diamond gives us intuition about the reason for the space in between the diamonds. A nearly horizontal line indicates that the slave was sitting idle waiting for a task to become computable. A line with a greater slope indicates that the slave was waiting for the input files to the task to be shipped over the network.

Appendix C contains examples of summary tables and graphs for several runs of *EPIC*.

3.10 Results

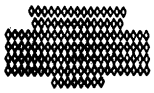
No conclusions can be drawn about the overall performance of *EPIC* without reference to a specific application. The following chapter discusses the application of *EPIC* to VLSI design rule checking, circuit extraction, and `Makefiles`.

3.11 Future Extensions

In this section, several extensions to *EPIC* are contemplated. A fairly straightforward extension is to delete intermediate files as soon as they are not needed. This is not difficult to implement, except when it is combined with the another straightforward extension, which is to avoid buffering intermediate files at the master. The buffering provides a redundancy that is needed to avoid repeating work that is lost due to a crashed slave. If both these extensions are implemented, and if a slave crashes, we may find that we have "burned our bridges behind us": the files needed to redo the slaves work may not exist anymore, possibly forcing us to pop back to the roots of the data dependency graph and effectively start over. The motivation for these extensions is discussed in the following chapter.

Another extension is to bring more intelligence into the choice of which slave to assign to the highest priority task. Most of the time, there are plenty of tasks to execute, and the master is waiting for a slave to finish its current task. But data dependency graphs that have narrow sections, such as the initial separation stage of a "divide and conquer" application, may be run more efficiently if the most powerful computer is used for the bottleneck task.

One flashy feature that would be relatively easy to add is the ability to revive old slaves whose processors crashed and were then brought back up. As mentioned before, *EPIC*'s predecessor, `PDRC`, had this capability.



A more substantial extension addresses the problem of continuing the computation even if the master's processor crashes. It involves the use of *shadows*. A shadow runs on a different processor from the master, though it could share a processor with a slave. It maintains a database of slaves and tasks. Using the message passing facility, it monitors events as they happen on the master and updates its database accordingly. If the message passing facility detects that the master has crashed, the shadow contacts the slaves and takes over control of the computation, thus becoming the new master. If the master and shadow are VAXclustered together, then the transition is conceptually straightforward, since the master's buffered files are still accessible. If they do not share a VAXcluster, then the shadow must actively copy the master's buffered files as they are created.

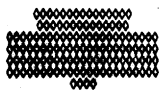
Shadows were not implemented in $\mathcal{EPIC}$ due to lack of time. However, it is unclear whether they would actually be used in practice if they existed. They help make $\mathcal{EPIC}$ fault-tolerant by adding redundancy, but in the case of VAX computers that are not VAXcluster members, they do this at a considerable cost of disk space.

Another substantial extension attempts to reduce the penalty of data communication. The concept is analogous to that of *instruction prefetch*. Based on the observation that network file transfers are more I/O bound than compute bound, $\mathcal{EPIC}$ would attempt to predict what task a slave would execute before the slave finished its current task. The slave would then retrieve the next task's input files in a separate process. Presumably, the slave's execution process and file transfer processes would not detrimentally compete for cycles within the slave's processor, because they use different resources.

Another related technique is *delayed reporting*. Currently, when a slave completes the execution of a task, it immediately proceeds to transfer the output files back to the master. Only when the transfer is complete does the slave notify the master that it is ready to execute another task. By notifying the master as soon as it is finished with the execution of its current task, the slave can be assigned a new task while it is still transferring the old output files. This approach is most effective if the slave already has the files it needs to execute the next task. Hence it is an ideal companion to *data prefetch*.

Data prefetch is difficult to implement because it involves predicting the best task to give to a machine when the execution is in some future state. The use of this technique

would most likely require altering the task scheduling strategy. While these enhancements are interesting topics for future research, the potential gains will diminish as VAXclusters become a more popular vehicle for coarse multiprocessing.



Chapter 4

Applications

This chapter discusses several applications of *EPIC*. Methodologies are presented for automatically generating an execution control file for each application. Results are given for various cases of each application run on several different multiprocessor configurations.

A comparison is made between *EPIC* with ECAD's DRACULA serial DRC program and ECAD's Parallel DRACULA.

4.1 Design Rule Checking

The challenge of adapting DRACULA to be distributed over a network of VAX computers using *EPIC* lies in generating the execution control file from the ECAD rules file. In order to do this, we have to understand the mechanics of how DRACULA is normally run on a single VAX computer. The VLSI process engineer defines the geometric design rules. The VLSI layout designer lays out the chip according to the design rules, thus generating a file in some standard layout description language, such as CIF [Mead, Conway 1980] or GDSII[1]. A programmer must then specify the process engineer's design rules in the language defined for that purpose by ECAD. These rules are fed to ECAD's preprocessor, PDRACULA, which generates the VAX/VMS command file which runs all the VAX/VMS executables that implement the statements in the rules file, hence running the DRC. Typically, the command file is submitted as a batch job.

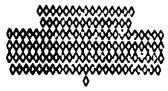
[1]GDSII is a trademark of G.E. Calma Corporation

To maximize efficiency, ECAD rearranges the statements in the rules program. If any individual DRC program is called by more than one rules statement, then ECAD's preprocessor tries to execute those statements together with one call to the program (while obeying data dependency constraints) and thereby minimize image activations. Depending on the value of a switch set in the rules file, the preprocessor may attempt to rearrange the order of execution of the rules program statements and delete temporary files to minimize peak disk space usage.

Unfortunately, all these optimizations deplete the extent of the parallelism by introducing new data dependencies. By deleting intermediate disk files after they are used, the preprocessor introduces a new constraint restricting the order of the execution of the rules statements. But the philosophy behind *EPIC* is to use whatever hardware you have available to solve a specific problem as quickly as you can. We are willing to sacrifice disk space in order to achieve maximal speed. It is worth noting that *EPIC* may not be able to DRC large chips if there is just enough disk space to do a serial run using ECAD's optimized file deletions.

As mentioned in the previous chapter, it would not be hard to modify *EPIC* to optionally delete intermediate files once they are not needed. This would bring *EPIC*'s peak disk space usage down considerably. But since *EPIC* schedules so as to minimize execution time, rather than disk space, it still wouldn't be as stingy as an optimized serial DRC. To further close the gap, *EPIC* could be modified to avoid storing every intermediate file on the master's filesystem. Instead, a slave would copy its task's input files directly from the slave that produced them (or from the master if the task is a root node in the data dependency graph). Rather than having the slave copy its task's output files back to the master, the master would just note where the file resides. *EPIC* could then copy the final output files (such as the DRC error summary and layout files) back to the master's filesystem. As mentioned in the previous chapter, this would cut down the file transfer overhead by as much as a factor of two. The disadvantage is that a crashed slave's previous work would have to be redone.

A more practical consideration about the preprocessor is that it's rearrangements of the command file make it mechanically difficult to identify the VAX/VMS commands



Fragment from a DRC rules program:

```
AND POLY DIFF GATE ; Figure out the gate area
WIDTH GATE LT 4.0 OUTPUT GWID 32 32 ; Gate width >= 4u
```

Corresponding execution control file fragment:

```
task "AND POLY DIFF GATE ; Figure out the gate area"-
    /INPUT =(POLY.DAT,-
             DIFF.DAT)-
    /OUTPUT=(GATE.DAT)-
    /DCL= ("$@SYS$LOGIN:MOSIS.COM 32")

task "WIDTH GATE LT 4.0 OUTPUT GWID 32 32 ; Gate width >= 4u"-
    /INPUT =(GATE.DAT)-
    /OUTPUT=(GWID32.DAT)-
    /DCL= ("$@SYS$LOGIN:MOSIS.COM 33")
```

Corresponding execution command file fragment:

```
$GOTO 'P1' !Jump to the task number specified as first parameter
$ !
$32: !AND POLY DIFF GATE ; Figure out the gate area
$RUN SEGCAD$ECAD:LOGICAL
    2 POLY DIFF GATE 1000 MIC 0

$EXIT
$ !
$33: !WIDTH GATE LT 4.0 OUTPUT GWID 32 32 ; Gate width >= 4u
$RUN SEGCAD$ECAD:SPACING
    1 GATE GATE 0.000 4.000 MIC 1000 OS
    0 0 0 0 0 0 0 0 0 0 0
            NOT-CONJUNCTED
    1 GWID32 GWID32 32 32 100

$EXIT
```

Figure 4.1: MOSIS CMOS DRC rules fragment, ECF fragment, and COM fragment

needed to execute any particular task in the rules file. For this reason, directly decomposing the preprocessor's command file was not a successful strategy.

A better approach is to decompose the rules program and run the preprocessor separately on each statement. Every command file generated by the preprocessor is parsed to remove the extraneous initialization and error merging code. The remaining text from each command file is used to construct a single command file. To execute a single task in the execution control file, this command file is invoked so as to execute the correct segment of code. A preprocessor was written to automatically convert a DRC rules program into an execution control file and an execution command file. It is called ECAD2ECF. Figure 4.1 shows the output of ECAD2ECF for a fragment of a rules program written to design rule check VLSI designs layed out using the 4μ MOSIS CMOS process [Mead, Conway 1980].

Two stages of the DRC are not covered by the tasks described in ECAD2ECF's execution control file. It is not clear whether the initial separation of each layer from the layout file is an inherently parallel operation. This operation is most likely implemented by examining the whole layout in one pass, appending to a given layer file whenever it encounters geometry for that layer. One thing that is clear about this initial stage is that the input file is large, since it contains the geometry for every layer. It would not be efficient for a slave to move this file across the network, perform the initial separation, and copy all the layer files back to the master. Instead, this stage is executed by the master, using a subprocess. Further preparation of each layer is described in the execution control file, and executed normally by the slaves. This preparation includes the full instantiation of the geometry in the layer, a polygon sorting step, and the merging together of overlapping polygons.

Similarly, the final stage of the DRC is executed by the master's subprocess. This stage involves compiling the information generated by the execution of each rule into a summary file and an error layout file. Conceptually, this step could be done in parallel by merging together the individual error files in a binary tree. If each error file has to be shipped over the network to a slave, this would probably not save any time. Using a VAXcluster, there is more of a potential gain. Unfortunately, there is no way to do a multi-stage merge using the DRACULA programs. The input files for the summary programs

are data files with an unknown record structure, and the summary files can't be converted back to the input format.

4.1.1 Predictions

According to the data dependency graph (Appendix B) for Digital's CMOS process rules, the maximum extent of parallelism is very high. After the execution of the tasks in the top row of the graph, which do the initial preparation of each VLSI layer, and the execution of the tasks in the second row of the graph, which mask out the geometry that is not to be checked, there are many tasks whose outputs are not used as inputs by any other tasks. Those correspond to simple DRC rules such as single-layer width and spacing checks. We call them "terminal tasks". *EPIC*'s task scheduler does very well in the presence of a large number of terminal tasks. They are computable early on in the computation, but their execution can be delayed until a processor has nothing else to do. They help "fill in the gaps" of processor idleness.

Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14-15	16-19	20-∞
Ticks	125	63	42	32	25	21	18	16	14	13	12	11	10	9	8	7
Speedup	1x	2x	3x	3.9x	5x	6x	6.9x	7.8x	8.9x	9.6x	10.4x	11.4x	12.5x	13.9x	15.6x	17.9x

Figure 4.2: Optimistic analysis of DEC CMOS rules based on data dependency

There are 125 tasks in the CMOS data dependency graph. Assuming that each task executes in one tick of time, a serial DRC will run in 125 ticks. If there are no communication costs, then with two processors, the job can be run in 63 ticks. As the number of processors grows, the data dependency will begin to constrain the maximum speedup we can hope to achieve. This is illustrated in the graphs on the corner of each page of the thesis (see the Preface), and in Figure 4.2.

The most striking feature of this chart is that it indicates that up to fourteen machines can be almost fully utilized in a parallel DRC. The analysis neglects communications overhead, but that is not why it is overly optimistic. The fault lies is in the assumption that each task takes unit time. Depending on the VLSI layout, the checking of rules that deal with active area or polysilicon might require the examinination of more complex geo-

metrical structures than the checking of rules that deal with well area or diffusion implant. *EPIC*'s task scheduling algorithm is equipped to deal with nonuniform task execution estimates, but ECAD2ECF does not provide the estimations. It would be interesting to statistically determine good estimates for the execution time of each task. Unfortunately, time did not permit this.

Processors	1	2	3	4	5	6	7	8	9-∞
Ticks	58	29	20	15	12	10	9	8	7
Speedup	1x	2x	2.9x	3.9x	4.8x	5.8x	6.4x	7.25x	8.3x

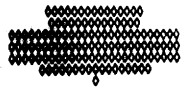
Figure 4.3: Optimistic analysis of MOSIS CMOS rules based on data dependency

The MOSIS CMOS design rule set is much simpler than DEC's, and hence is implemented in fewer rules file statements. Thus there is not as much potential for parallelism. This is balanced by the fact that for a chip of any given complexity, it is far easier to check the MOSIS rules than the DEC rules. The analysis of the MOSIS rules is in Figure 4.3.

4.1.2 Testing

Obtaining consistent results for *EPIC*/DRACULA has been difficult. We are more interested in the elapsed time of a DRC run than we are in the cumulative CPU time. Since the "multiprocessor" used for the test runs is just a set of timesharing VAX computers which are all connected to Digital's local Ethernet, the response time of both the network and the system has been unpredictable. Even late at night, many of the systems are loaded with batch jobs and high priority file system backups.

Several steps were taken toward minimising external factors that could alter the elapsed time for a test. Exploratory test runs were conducted at various times during the day, indicating that the computers were most responsive very early in the morning. Each result presented here was taken from the best of several runs on a particular multiprocessor configuration. In addition, we tried to make the test results at least partially immune to the timesharing competition of other batch jobs by running at a higher priority.



Nevertheless, the slaves typically received less than 80% of the CPU, as determined by the ratio of execution CPU time to elapsed execution time for every slave. Various factors contribute to this that may or may not be related to the parallel processing scheme. Page faults, for example, can be caused by timesharing competition for physical memory, which is unrelated to *EPIC*. On the other hand, page faults can also be caused by the increased number of image activations incurred due to the subdivision of the DRC job. The runs on DECnet suffer even more, because DRC program invocations are often interspersed with file transfer commands, possibly causing the DRC program pages to be swapped out. It should be noted that since the measured CPU percentage was generally greatest for the serial runs, the observed speed-up factors may be smaller than those that might be achieved using *EPIC* on a single-user multiprocessor.

The number of processors available for testing was limited, since several of the group's computers were recently upgraded from VAX 11/780 to VAX 11/785 computers. From a software point of view, the upgrade is very transparent. The only noticeable change is the improved response time. But to make a meaningful statement about the speedup factor *EPIC* provides to DRACULA, we need to compare the elapsed time for a parallel run on a fixed number of identical processors to the elapsed time for a serial run on one of those processors.

MicroVAX computers provide one possible alternative. They are starting to proliferate in quantity throughout the Hudson plant and it is possible to get exclusive access to them at night. So assuming they all have the same amount of physical memory, their performance should be fairly predictable. Unfortunately, most MicroVAX computers are configured with far too little disk space and paging file space to run a substantial DRC. Small DRCs aren't very informative, since the amount of time required to execute each task becomes small enough so that the communications overhead is substantial. Since *EPIC* is geared toward accelerating the verification of much larger chips, data gleened from DRCs run on the available MicroVAX computers will be overly pessimistic.

Sufficient resources were not available to fully test my predictions for the maximum extent of parallelism in DRC. A VAXcluster with six machines was available for testing during off hours, but it consisted of three VAX 11/780 computers, two VAX 11/785 com-

puters, and one VAX 8600 computer. In addition, three VAX 11/780 computers connected by Ethernet were available. Six MicroVAX II computers were also available, but were not generally capable of DRCing my benchmark.

The results presented here consist of *EPIC* runs using up to three VAXclustered VAX 11/780 computers and up to six independent VAX 11/780 computers. The independent VAX 11/780 computer tests were accomplished by not informing *EPIC* that the three VAXclustered computers shared the same filesystem. File transfers were made with DECnet protocol, so the tests suffered the same overhead that would have been incurred if the computers had not been VAXclustered together. The elapsed time from these tests is compared to the elapsed time for a serial run on one VAX 11/780 computer. By testing how well *EPIC* performs using just one processor, we attempt to isolate the control communications overhead incurred due to *EPIC*.

EPIC's raw elapsed times are measured from the time the `MASTER` program is invoked to the point after the run when the last slave is killed. We also give the average percentage of slave time dedicated to task execution, file transfer, and idle time. As discussed in Chapter 3, the idle time does not include the time at the beginning and end of each run when there is no work for the slaves to do. Finally, we give the ratio of the slaves' total execution CPU time to elapsed execution time, which provides a measure of how much our results suffered due to competion for the CPU.

In addition to analyzing the raw elapsed times, we try to determine why the performance didn't quite match the speed-ups predicted in Figure 4.2. Those optimistic figures didn't take into account the time required to split the chip into its constituent layers or the time required to merge the error reports back together. These times are subtracted from the raw elapsed times and the analysis is repeated using the modified data. The remaining non-linearities are small enough to be accounted for by *EPIC*'s overhead, and by other factors that are difficult to control, such as competion for the CPU, page faulting, and an increased number of image activations.

According to the tests in Figure 4.4, *EPIC* offers a significant performance enhancement over serial `DRACULA`. I was able to try ECAD's `Parallel DRACULA` on three VAXclustered VAX 11/780 computers using the same benchmark. The tests indictated

Control	ECAD		$\mathcal{EPIC}$/VAXcluster			$\mathcal{EPIC}$/DECnet				
Processors	1	3	1	2	3	2	3	4	5	6
Raw times (in seconds)										
Elapsed	18350	10371	19288	10270	7076	11743	8020	5771	5158	4462
Speedup	1x	1.8x	0.95x	1.8x	2.6x	1.6x	2.3x	3.2x	3.6x	4.1x
After subtracting initial layer split and final error merge										
Elapsed	17855	9876	18793	9848	6581	11248	7525	5276	4663	3967
Speedup	1x	1.8x	0.95x	1.8x	2.7x	1.6x	2.4x	3.4x	3.8x	4.5x
% File	0%	0%	0%	0%	0%	4.9%	6.3%	7.3%	8.9%	11%
% Exec	100%	?	99%	99%	99%	94%	93%	91%	90%	88%
% Idle	0%	?	0.56%	0.76%	0.89%	0.95%	0.96%	1.5%	0.95%	1.3%
% CPU	79%	?	80%	79%	77%	71%	74%	77%	74%	76%
See Page	94	94	95	97	99	101	103	105	107	109

Figure 4.4: Results and analysis of DEC CMOS DRC tests

a speed-up of 1.8 using three machines. This is less of a speed-up than was reported in ECAD's article [Nielson 1986], which reported a speed-up of 1.78 for two machines. This disparity may be due to excessive competition for the CPU, a factor that was difficult to determine because the ECAD controller runs several jobs simultaneously on *each* processor. On the average, the ECAD jobs each got 43% of the CPU, but there were typically two or three jobs on each processor at any given time, so it was difficult to determine how much the DRC was slowed by timesharing overhead.

On the same benchmark, with the same hardware configuration, $\mathcal{EPIC}$ demonstrated a speed-up of 2.6. This is not conclusive, however, and we suspect this data doesn't tell the whole story for two reasons. ECAD's results were most likely based on the DRC of a larger chip than the one used for this benchmark, which reduces the relative overhead of submitting a new batch job for each task. The competition for the CPU was possibly an important issue, but it is difficult to determine the extent of its effect.

In addition to the difference in runtimes between the $\mathcal{EPIC}$ and ECAD benchmarks,

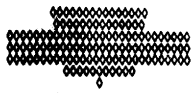
the tests indicate the relative versatility of $\mathcal{EPIC}$'s approach. Only three homogeneous VAXclustered processors were available, but six homogeneous processors were available through DECnet. Since $\mathcal{EPIC}$ is capable of using DECnet without VAXclusters, we were able to perform tests using more processors. The relative availability of VAXclustered versus independent computers at DEC may indicate that $\mathcal{EPIC}$ is more generally useful than `Parallel ECAD`. This may become less important if VAXclusters become more prevalent in the future.

There are several more interesting pieces of information that can be gleaned from the data in Figure 4.4. First, we mention that since we ran the master and one slave on a single processor, there was a nonlinearity in the DECnet tests. $\mathcal{EPIC}$ notices when the master and a slave are running on the same processor and uses this knowledge to "short-circuit" that slave's DECnet file transfers with a local `$COPY` command. The impact of this short circuiting can be seen by comparing SLAVE1's file transfer times with those of any other slave, on all the charts of $\mathcal{EPIC}$/DECnet tests in Appendix C.

For all the DECnet tests, the file transfer time rose with the number of processors. Not enough data is present to determine the relationship between the file transfer overhead and the number of processors (i.e. linear, polynomial, or exponential).

A definite pattern was not observed for the the idle time overhead, but it never exceeded 1.5%. In the tests made here, no slave was ever idle for lack of work to do. Idle time accumulated due to network message passing latencies. We would expect the absolute message passing time to remain unaffected by the number of processors, since the number of tasks remains constant. Naturally, since the elapsed time of the DRC shrinks as the number of processors grows, we would expect the relative overhead of the message passing latency to increase. But the dominant factor in message passing latency is probably network congestion, which varies greatly over time. As discussed in Chapter 3, the VAXcluster runs use DECnet for control communication, so they are also affected.

The tests run here indicate that the speed-up factor was beginning to fall off as the number of processors increased to five or six. This is expected in the DECnet tests, since the data communication overhead increases with the number of processors. It is likely that the we will not be able to use fourteen independent processors to achieve our goal of



completing a DRC as fast as the data dependency will allow. Nevertheless, the results are good enough to justify the additional hardware expense in a production environment. On the other hand, the three processor VAXcluster test results were sufficiently promising to warrant additional experimentation. It would be intriguing to see how many VAXclustered processors we can use before the speed-up factor begins to fall off.

Both the DECnet and VAXcluster results may have been more optimistic if the test case used a larger chip. Since the execution time of the DRC tends to grow faster than the size of the files, the data communication overhead would probably become less significant. The control communication overhead would vanish quickly, since it grows with the size of the design rule set, not the chip size.

So far, with up to six processors, *EPIC*'s task scheduling strategy has been essentially optimal. If as we increase the number of processors, inefficient task scheduling becomes a bottleneck, we will probably be able to improve the task scheduling by supplying statistical estimations of the length of each task, based on previous runs.

Thus *EPIC* potentially offers the mechanism to run DRCs as fast as the critical path through the data dependency graph will allow. To achieve this goal, we need to do the following:

- Use more VAXclustered processors.
- Obtain exclusive access to them, so the test results will be repeatable.
- Develop statistical estimations for the execution time of each task, so task scheduling will (hopefully) not be a bottleneck.

The difficulties I encountered while running DRCs on MicroVAX computers do not represent an unsolvable problem. By configuring them with enough physical memory and disk space, a group of MicroVAX II computers connected by a dedicated Ethernet would work well as a low-cost, high-performance DRC server. If ten MicroVAX II computers can offer an 7x speedup for DRC (the optimistic analysis indicated 9.6), then they offer a faster turnaround time than one VAX 8600 computer (which runs roughly 5 times as fast as the MicroVAX II computer), for roughly the same monetary cost.

4.2 Circuit Extraction

Digital's circuit extractor [Tarolli, Herman 1983] has been adapted for parallel execution using *EPIC* in a system called MACE (a Multiprocessing Approach to Circuit Extraction) [Levitin 1986]. MACE attempts to take advantage of the geometric locality of VLSI by dividing the layout into *swaths* (strips) which are processed in parallel. Unfortunately, this is a much more difficult task than it is for design rule checking [Bier, Pleszkun 1985]. It is not clear how to correctly handle the case when a swath's border crosses a transistor. However, by carefully choosing the swath boundaries, it is sometimes possible to avoid this case. For a chip of sufficient size, it may not be possible to draw a straight line across it without hitting a transistor. For this reason, MACE has only been tested with relatively small cells. As stated before, *EPIC* is geared for larger scale problems so that the overhead of control communication becomes negligible.

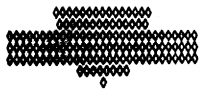
The results as of this writing have not indicated a significant speed-up. The layouts were partitioned into two swaths. The extraction was performed separately on each swath using two slaves, and the two resulting circuits were merged together afterwards. In practice, the speed gained through parallelism in the extraction phase was overwhelmed by the cost of merging the circuits together. The serial extraction actually took less elapsed time than the parallel extraction and merge [Levitin 1986].

4.3 Compiling and Linking Programs

The automatic translation of `makefiles` to execution control files is fairly straightforward. Writing `Make2ECF` was simply a matter of changing the syntax of each task description.

Since `make` was used to control generation of the *EPIC* executables, and since *EPIC* is composed of many different modules, it was reasonable choice for a benchmark. The data dependency graph for compiling and linking *EPIC* is in Appendix B.

The chart in Figure 4.5 shows the results of simulating the execution based on unit task length and zero communications cost. The shape of the data dependency graph is far



Processors	1	2	3	4	5-6	7-9	10-18	19-∞
Ticks	25	13	9	7	5	4	3	2
Speedup	1x	1.9x	2.8x	3.6x	5x	6.3x	8.3x	12.5x

Figure 4.5: *EPIC* analysis of Makefile simulation based on data dependency

more regular than that of either DRC rules set. Due to the relative absence of terminal nodes, it was not always possible to "fill in the gaps" of processor idleness. Therefore, the processors were not well utilized if there were more than seven of them, even though the minimum (and maximum) extent of parallelism is 19.

Control	make	*EPIC*/VAXcluster			*EPIC*/DECnet		
Processors	1	1	2	3	2	3	4
Raw statistics (seconds)							
Elapsed	619	630	340	253	477	392	332
Speedup	1x	.98x	1.8x	2.5x	1.3x	1.6x	1.9x
% File	0%	0%	0%	0%	19%	25%	34%
% Exec	100%	97%	96%	95%	79%	71%	62%
% Idle	0%	3.2%	3.9%	4.8%	2.2%	3.9%	4.2%
% CPU	66%	69%	68%	69%	59%	57%	61%
See Page	111	112	114	116	118	120	122

Figure 4.6: Results and analysis of `make epic`

Figure 4.6 shows the results of *EPIC* compilation tests run on a VAXCluster with up to 3 VAX 11/780 computers and on DECnet with up to 4 VAX 11/780 computers. The VAXclustered run showed a reasonable speedup with up to three processors, but more tests will have to be run to see how well these results will scale.

The tests run with independent VAX computers indicate that the compilation of *EPIC* is not sufficiently compute-bound to allow it to be efficiently distributed over an Ethernet. As the chart shows, the file transfer overhead grew rapidly as the number of

processors increased. Running parallel **make** over DECnet may become profitable if the *data prefetch* and *delayed reporting* extensions of Chapter 3 are applied to $\mathcal{EPIC}$.

Chapter 5

Conclusion

5.1 Summary

In this thesis we have presented $\mathcal{EPIC}$, the implementation of a software methodology for coarse grained parallel processing. It is based on a computational model that is applicable to a variety of different problems. We have described the characteristics that a program must possess in order to be accelerated by $\mathcal{EPIC}$. In addition, we have described the adaptation of several existing applications to parallel computation using $\mathcal{EPIC}$, with varying degrees of success.

Parallel DRC was particularly successful. The tests run indicate a performance increase that justifies the usage of the extra hardware. The base DRC program used in this thesis was ECAD's `DRACULA`, but any design rule checker that uses intermediate files could have been used. The strategy for running DRCs in parallel presented here is only one of two promising approaches. We divided the DRC by allocating different rules in the design rule set to each processor. Also, the data partitioning scheme of [Bier, Pleszkun 1985] will work with any design rule checker, and can be readily adapted to $\mathcal{EPIC}$.

5.2 Directions For Future Research

The results presented in this thesis did not fully test the claims made about the extent of parallelism of either DRC or `Makefiles`. With more time and resources, it would

be interesting to try to execute a DRC as quickly as the critical path will allow. This would also give the task scheduling algorithm a more a substantial workout. The five and six machine tests showed optimal performance from the task scheduler, but that was too easy. To better support the claims made here about the scheduler's near-optimality in most real data dependency graphs, we need to run more tests with more machines.

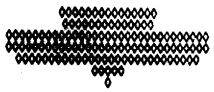
One way to further increase the extent of parallelism in VLSI design rule checking is to combine rule partitioning with data partitioning. Essentially, once the chip is divided into separate slices, several processors could be allocated to each slice, and each slice could be be checked by exploiting rule-based parallelism. The whole computation could be controlled by $\mathcal{EPIC}$ using a single execution control file. Another strategy would be to use a two-level hierarchy of star networks, with each master reporting to the *grandmaster*. The single-master approach requires a bit of effort to prevent naming conflicts with intermediate layer and error files, but offers the advantage of automatically load-balancing the computation if any of the slices finish before any of the others.

5.2.1 Other Applications

$\mathcal{EPIC}$ provides the basis for the acceleration through parallelism of a potentially wide variety of existing software. Any computation controlled with Unix `Makefiles` can be automatically converted to be run in parallel with $\mathcal{EPIC}$. Another VLSI CAD application that has the potential for acceleration via $\mathcal{EPIC}$ is mask pattern generation software. In particular, ECAD's MDP[1] software uses the same rules file format and preprocessor as DRACULA, so it may work with the existing ECAD2ECF preprocessor with only minor syntactic additions. This was not explored further due to lack of time.

Using $\mathcal{EPIC}$ on VAXclusters, the data communications overhead becomes negligible, and the set of programs that can be profitably accelerated through parallelism expands greatly. One application that comes to mind is merge-sorting. This classic binary divide-and-conquer algorithm is ideal for $\mathcal{EPIC}$. It would be fairly easy to adapt an existing merge-sort program for use with $\mathcal{EPIC}$. The constraining factor is the time required to

[1] MDP is a trademark of ECAD corporation



write the partial lists into disk files. But this overhead is also incurred in serial merge-sorts if the list being sorted is too large to fit into physical memory.

5.2.2 Reducing the overhead

The disk file overhead issue brings to light another issue. $\mathcal{EPIC}$ addresses a very coarse parallelism. The control communications overhead forces us to apply the constraint that a problem must be subdivided into tasks that each task a "long time" to execute. But $\mathcal{EPIC}$'s model of parallelism doesn't require the loose coupling of the Ethernet environment. A more tightly coupled multiprocessor would be able to accelerate a wider range of applications. The concepts used in $\mathcal{EPIC}$ could be applied to a controller on such a processor. It would be interesting to see how such a system might develop.

5.2.3 Lessons Learned about Distributed Programming

In the past twenty years, there have been dramatic improvements in the quality of the tools used for programming. In particular, the recent advent of source line debugging for high level programming languages on the VAX/VMS operating system has allowed the programmer to more fully concentrate on the most interesting aspects of his task. Unfortunately, this capability is often less accessible to those writing distributed or asynchronous programs. If a program is invoked by creating a process on a remote processor, how will the debugger interact with the terminal? It is possible to work around this problem by having the remote process allocate a terminal that is directly connected to the remote processor. That is not very helpful if there are many processors or if they are physically inaccessible. Much work needs to be done in the area of distributed programming environments.

Similarly, software engineering has advanced considerably from the days of `FORTRAN` and `COBOL`. The concepts of structured programming, data abstraction, object oriented programming, data driven programming, and so on are well documented, publicized, and lectured about in our undergraduate halls. In the course of implementing the message passing facility of $\mathcal{EPIC}$, less familiar methodologies had to be adopted to insure consistent data structures within a single processor, and to avoid deadlocks between two

communicating processors while guaranteeing message delivery. If parallel processors are to become a popular hardware platform, we must learn how to program them as well as we know how to program serial machines.

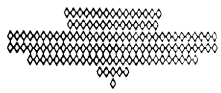
5.3 Conclusion

Several factors affect how well the potential for acceleration of CAD tools through parallelism will scale with time. As the complexity of VLSI circuits rises, the extent of parallelism will rise due to geometric locality in the layouts, the constant overhead of $\mathcal{EPIC}$'s control communication will become negligible, and the overhead of data communication will most likely become less significant. Data communication will almost certainly not become more significant as the complexity of the chips rises. This is based on the assumption that CAD tools have time complexity $\geq O(n)$ where n represents the size of the input files, since they must at least examine all their input. Hierarchical CAD tools are included in this assumption, because the file representation is hierarchical as well. Empirically, the time complexity for flat DRCs has been observed to be roughly $O(n^{1.2})$ with n being the number of transistors [McGrath 1985].

Another factor that will determine how much extra speed we can squeeze out of parallelism is the power of the processors on which we run the CAD tools. The VAX 8600 computer will run roughly four times as fast as the VAX 11/780 computer. Since Ethernet technology is used as the control communications medium for both processors, the control communications overhead on VAX 8600 computers may be as much as four times as significant as the tests presented here indicate.

This statistic is best put into perspective by comparing it to the difference between the complexity of circuits being fabricated in 1977, when the VAX 11/780 computer was introduced, and the complexity of the circuits of 1985, when the VAX 8600 computer was introduced. While processor speed may have improved by a factor of four, VLSI circuit complexity has increased by a factor of about twenty-five [Allen 1983].

Thus we predict that parallelism will continue to be a viable means for accelerating layout verification of VLSI circuits in years to come. $\mathcal{EPIC}$ provides an inexpensive means



of substantially improving the throughput of existing software. As advances are made in both processor speed and the exploitation of hierarchy in CAD tools, parallelism can still be used to further reduce the execution time.

Appendix A

EPIC/DRACULA User's Manual

Parallel DRC is a method for running the ECAD's VLSI design rule checker (DRACULA). By dividing the run into separate portions to be run on several computers, Parallel DRC reduces the amount of time required for a DRC run. A DRC using the standard method of running on one computer may require several days to run on a large chip. This time can be reduced to an overnight run using Parallel DRC.

This appendix describes the following aspects of Parallel DRC:

- How Parallel DRC works
- Potential Benefits from running Parallel DRC
- Environment for running Parallel DRC
- How to run Parallel DRC

A.1 How Parallel DRC Works

The program used to run Parallel DRC is called *EPIC* (Exploiting Parallelism In CAD). This program sets up processes on several computers to run portions of the DRC. The computers are logically arranged in a star network. The central computer, called the *master*, manages the work of all the other computers, called *slaves*. The entire design rule check is broken into separate tasks, with each task roughly corresponding to a single DRC rule. The master dynamically assigns tasks to the slaves, telling them what files are

needed to run the task. The slaves copy the files to their own directories and run the tasks. The master keeps a record of the files each slave has. As each task is completed, the slave notifies the master and sends the DRC output files to the master's directory. The master then assigns another task to the slave. The execution proceeds in this manner until all the tasks are completed. The last step is for the master to combine all the separate error files (.ERR) into one file and append it to the summary (.SUM) file.

The MONITOR program allows the user to initiate and control the parallel execution, and provides a periodically updated display of the status of each slave's process.

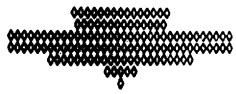
A.2 Potential Benefits From Running Parallel DRC

To evaluate whether or not you want to use *EPIC* to run ECAD DRC, you must understand the basic principle behind it. DRC is not really a single program that must be run from start to finish by a single CPU. It is a collection of related programs, which are typically run one after another. Each program 'communicates' to the others simply by reading and writing disk files.

EPIC provides a mechanism to distribute the execution of these programs over several computers on a network. This distribution is very efficient in that almost no work is duplicated by the extra computers. The only extra work involved is the file transfers needed to move the input and output files to the appropriate CPUs.

Preliminary tests of Parallel DRC have demonstrated a speedup of 4.5x using 6 computers to check a medium size chip. The speedup ratio will approach the number of computers as the chip gets larger, since the time required to run the DRC rises faster than the size of the data files.

The greatest practical advantage of Parallel DRC occurs with chips that take serial DRCs several days to run on a loaded VAX computer. During working hours, the DRC has to fight for CPU time with interactive processes, thereby reducing everyone else's efficiency while further delaying the completion of the DRC. Using *EPIC*, it will be possible to complete the DRC overnight. That translates into a faster turnaround time for the layout designers, and less aggravation for the other users of the computer facility.



A.3 Environment For Running Parallel DRC

A.3.1 Requirements For $\mathcal{EPIC}$

$\mathcal{EPIC}$ requires no special hardware configurations. It runs on any number of VAX computers, each running VAX/VMS Version 4 or later, and all connected by DECnet. The system runs in a heterogeneous environment of VAXclustered and unVAXclustered nodes. Informing $\mathcal{EPIC}$ which nodes are VAXclustered results in increased performance, due to the decrease in file transfer overhead. Running on Microvax computers is possible if there is enough disk space to hold the ECAD software and the chip data.

$\mathcal{EPIC}$ requires that on each system, you have an account with the following characteristics:

Proxy:	`*::USERNAME ==> USERNAME`
Privileges:	`NETMBX, TMPMBX, GRPNAM`
Buffered I/O Byte Count Quota:	13000
Timer entry queue quota:	10
Open file quota:	100
Subprocess quota:	5

You should define a logical $\mathcal{EPIC}$ to point to the area where the $\mathcal{EPIC}$ programs reside on your system. In addition, you need to set up two command files in your `SYS$LOGIN` area: `MASTER.COM` and `SLAVE.COM`. You can copy examples of these files from the $\mathcal{EPIC}$ distribution area.

You will want to run the parallel DRC using a different subdirectory for each slave. This is obvious for unVAXclustered computers, but even when two nodes share a file system, their slaves should be provided with separate subdirectories. This is due to a restriction in the ECAD `DRACULA` system that causes input files to be read-locked even if they will not be rewritten. This eliminates the possibility of file-sharing, even on a VAXcluster, because if a process tries to open a file that a parallel process has already locked, a fatal error will be signalled. VAXclusters are still helpful, provided the master is running on the VAXcluster, since $\mathcal{EPIC}$ is smart enough to use local file transfers rather than DECnet file transfers between VAXclustered nodes.

EPIC allows you to map your slaves to your processors any way you want. In other words, you can have any number of slaves on each CPU. For Parallel DRC, the most efficient strategy is to assign only one slave for each processor. You can run the master on a processor that is already running a slave, since master doesn't consume very much CPU time.

A.3.2 ECAD DRACULA Requirements

You must have the ECAD system installed on each filesystem. VAXclusters only need it installed once, rather than once for each CPU.

A.3.3 Input Requirements

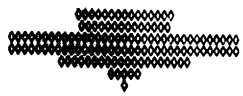
The input requirements are exactly the same as those for serial ECAD DRC. You must have a layout file in some format understood by ECAD, and you must know the primary cell name. You must also have a rules file (.DRC) describing the geometric tolerances for the appropriate process technology. The rules file is used to generate control files that allow *EPIC* to run the Parallel DRC.

A.4 Running A Parallel DRC

A.4.1 Preprocessing Steps

The *EPIC* kernel has no knowledge of DRC. It can run DRC only by providing with it a parameter file, called an *execution control file* (with extension .ECF). This file can be generated directly from the DRC rules file using the program ECAD2ECF.EXE. This program also generates a command file that contains the DCL code that directly drives ECAD DRC. ECAD2ECF.EXE is easy to run, though it may take over an hour on a well-loaded VAX 11/780 computer. The following is an example of its use. We assume that CMOS.DRC is a rules file in the current default directory.

```
$ RUN/NODEBUG EPIC:ECAD2ECF
Ecad file name: CMOS.DRC
```



```
ECF file name: CMOS.ECF
COM file name: CMOS.COM
%DELETE-W-SEARCHFAIL, error searching for !AS
$
```

CMOS.ECF must then be placed in the master's subdirectory. It contains information about each task needed to control the parallel execution. Specifically, for each task, it indicates all of the input files, all of the output files, and all of the DCL commands needed to generate those output files.

CMOS.COM must be placed in the SYS$LOGIN: area of each slave. We place it in SYS$LOGIN rather than in the slave subdirectory so that we only have to store this rather large file once per VAXcluster (see the discussion above about file sharing on VAXclusters).

Generally, the rules file for a given technology will remain fairly stable throughout time. The only information that changes more often are the description parameters at the top of the rules file. These might change with each run. We want to avoid running the preprocessor as much as possible, since it is fairly time consuming. The best approach is to run it once for each generation of the process technology, using generic description parameters. Then, for each new set of description parameters, you must generate a new .ECF and a new .COM file by doing the appropriate global string replacements in the generic .ECF and .COM files. A program, FIXECAD.EXE, is provided for this purpose. It is fairly easy to use, and doesn't take very much time (typically less than a minute). It prompts for the old and new .ECF and .COM file names, and for the old and new description parameters. Since the program does unintelligent global string replacements, you must choose your generic description parameters so they will be unique. The appendix contains an example of the use of FIXECAD that also demonstrates appropriate generic description parameters.

Sample .DRC, .ECF and .COM files for several technologies are provided in the *EPIC* distribution. You may want to use these if they are sufficiently up-to-date. You will still need to use FIXECAD to update the description parameters.

A.4.2 Running *EPIC*

All user interaction with the *EPIC* system is through the `MONITOR.EXE` program. It is recommended that you run this program on the same processor as the master, though it is not required. `MONITOR` uses the VAX/VMS Screen Management facility (SMG), so you must run it from a DEC supported terminal such as a VT100 or a VT200 series terminal. You can also run `MONITOR` in batch mode or from a command file. Normally you will want to initiate the program interactively, since the network connections that will be made occasionally fail on the first try due to timeouts or network flakiness. To save typing, you have the option of initiating the start-up from a command file and continuing or fixing any problems interactively.

To start `MONITOR`, use "`$ RUN/NODEB EPIC:MONITOR`". Your screen will then be divided into three segments. The top third contains process monitoring information. Each row in the display corresponds to a slave's subprocess, and is periodically updated to display a variety of statistics including CPU time, elapsed time, the name of the current program, and the number of tasks it has completed. The middle third is for error messages, status messages, and other diagnostics. The bottom third is for your input.

The normal state of the program is that no prompt is offered. This is so that the monitor can respond to any messages it receives from the master. There is no master initially, so this may seem confusing. As soon as the user types something, monitor provides a prompt in the bottom window and echoes what was typed thus far. While in this "read line" mode, the monitor cannot react to messages from the master, so the normal state is not to provide the prompt. If you type at monitor and it doesn't echo, that means it isn't finished doing what you last told it to do. If you start to type something to monitor and decide not to issue a command, just type **CTRL/U RETURN** to get rid of the prompt.

Normally, the first thing to do is to create a master. Use the command

CREATE/MASTER/PROXY node comfile ecfile file-prefix cluster-list

If you do not type in the arguments, you will be prompted for them. The standard DCL parser and line editor are used, so you will be able to use the arrow keys to edit your input. Two special purpose keys are also assigned. PF1 terminates the current line (executes it) and clears the bottom two thirds of the screen. PF2 terminates the current

line and repaints the entire screen.

The first argument, **node**, is the name of the node on which the master will be run. Don't put the double colon (::) in here, just the name of the node. The second argument, **comfile**, is usually `MASTER`, though you may have more than one version of this file that does different things with default directories and renaming of `NETSERVER.LOG`. Don't bother to specify the file extension, and don't include a device or directory specification; the file must reside in `SYS$LOGIN`. The third argument, **ecffile**, is the name of the Execution Control File (for example `CMOS.ECF`), only don't bother to include the extension when you type it here. You can specify a device and directory, but you don't need to if it is the same as **file-prefix**, the fourth argument. **File-prefix** is the master's subdirectory. It can include a device and directory specification. The initial input file must be in this directory, and all intermediate files and the error summary file will be placed there, so there must be enough room on the disk. The last argument, **cluster-list**, is a list of machines that share the same filesystem as the MASTER's node. Include **node** in this list. This information is used to optimize file transfers by using local `$COPY`s rather than decnet transfers when appropriate.

The PROXY qualifier is used because in some future version of *EPIC*, we may support password access.

After pressing carriage return, the MONITOR causes a process to be created on **node**. This process executes **comfile**, which should run `EPIC:MASTER.EXE`, which will acknowledge communication with monitor. It will then try to read in **ecffile**. You will be told the outcome of this attempt, and that will be your cue to begin creating slaves.

CREATE/SLAVE/PROXY **name node comfile file-prefix**

The only new parameter is the **name** parameter. This is used because more than one slave per machine is supported by *EPIC* (though not recommended for `DRACULA`). The name is used as a substring in file names, process names and in the group logical name table. It should contain only alphanumerics, and be no more than eight characters long. One would generally include the node name as part of this name when running on a VAXcluster, so the log files will be identifiable.

CREATE/SLAVE is not really executed by the monitor. The text of the command

is sent to the master, and the master executes the command. This allows you to queue up several CREATE/SLAVE commands without waiting for the command to finish. Diagnostic messages will indicate the success or failure when the information becomes available. Success will also be indicated by a new active line in the upper third (process monitoring section) of the screen.

EPIC supports the use of more than one VAXcluster. The following command tells *EPIC* about a VAXcluster other than the one specified in the CREATE/MASTER command:

SET/CLUSTER=node1, node2, node3, node4 ...

The refresh cycle for process display is initially set to one minute. You can reset it to (for example) five seconds with the following command:

SET/REFRESH=0 0:00:05.00

If for any reason you need to kill a slave, use the following command:

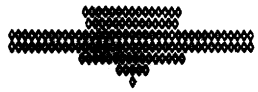
KILL/SLAVE slave's-node slave's-name

Again this command is not really executed by the monitor. The text of the command is sent to the master, and it does the dirty work. The result should be evident from the diagnostic message and process display. You can also do the dirty work yourself by stopping the slave's process on its node. In any case, *EPIC* will reassign that slave's task to another slave, and the computation will continue. If a slave fails due to a system crash, *EPIC* will behave similarly. The computation will go on with the expected degraded performance. You can also add a slave at any point in the computation with the CREATE/SLAVE command.

You can kill the whole computation, including the master, with the KILL command. This is a clean way to abort the computation. The log and summary files for the processes, though not for the DRC, will be generated. You can also stop the master's process yourself, and the slaves will terminate themselves soon thereafter.

You can use the monitor's EXIT command to get back to DCL. It is OK to do this while a computation is running. To get back in touch with a master that you have left on its own for a while, get back into the monitor, and use the command

MONITOR/PROXY master's-node



Performance will be much better if you do this while logged into master's-node.

A.4.3 Triggering The Parallel DRC

This is essentially automatic. As soon as the CREATE/MASTER completes, the master begins an initial step in the DRC in a subprocess. This is a task that must be completed before any of the slaves can be given any work. Normally, you will have created all the slaves before the MASTER finishes this step, but you can create slaves at any time, and they will be put to work if there is work to be done.

For completeness, we mention that the subprocesses in which the actual DRC is run do not inherit any process logical names or symbols you may have defined in your `LOGIN.COM`. This should not affect an ECAD DRC, but if you create a file `SYS$LOGIN:EPICINIT.COM`, it will be executed by the each subprocess before it starts running the DCL commands specified in the `.ECF` file.

A.4.4 Summary Files

In addition to the DRC summary file that is created in the master's subdirectory, *EPIC* leaves several other files in various places around your file system. Two summary files will be created in `SYS$LOGIN` on the master's computer. `EPICSTATUS.LOG` will contain a chart indicating the cpu time, the real time, and some other parameters for each slave. `EPICEXEC.PS` is a Postscript file that can be printed on an Apple Laserwriter[1]. It contains a graphical representation of the parallel execution. The leftmost column indicates the elapsed time at several points on the Y-axis. Each vertical column represents the activity of a slave. Each diamond is the execution of a task or rule. The height of the diamond is proportional to the amount of time it took to execute it. Each line segment between two tasks represents a data dependency between those tasks, and roughly corresponds to a file transfer. System `.LOG` files documenting the actual VAX/VMS programs run to execute the DRC are generated in whatever directory was the default directory when `EPIC:SLAVE` and `EPIC:MASTER` were initially run. `MASTER.LOG` and `SLAVE.LOG` are generated according

[1]Laserwriter is a trademark of Apple Computer Corporation

to the contents of MASTER.COM and SLAVE.COM. MASTER.LOG contains all the diagnostic messages sent to the middle screen of the monitor.

A.5 Appendix

A.5.1 Sample Run Of EPIC:FIXECAD

Note: You don't have to specify anything for the old and new versions of a field if you don't want to change that field. Every time a substitution is made, the old line and the new line are printed out. Much of this was editted out of the example below.

```
$ run epic:fixecad
Old COM:          cmos
Old ECF:          cmos
New COM:          field
New ECF:          field
Old Indisk:       infile.gds
New Indisk:       field.gds
Old Outdisk:      outfile.err
New Outdisk:      outfield.err
Old Print:        summary
New Print:        summary
Old Primary:      maincell
New Primary:      field
Old System:       gds2
New System:       gds2
Old Dir:          segcad$ecad:
New Dir:          segcad$ecad:

    1  TREEMAIN
    1  TREEFIEL

$ASSIGN INFILE.GDS FOR009
$ASSIGN FIELD.GDS FOR009

    0  TREEMAIN
    0  TREEFIEL

      1000          1  MAINCELL
      1000          1  FIELD

$ASSIGN OUTFILE.ERR FOR009
$ASSIGN OUTFIELD.ERR FOR009

    0  TREEMAIN  OUTMAINCELL
    0  TREEFIEL  OUTFIELD

            ,TREEMAIN.DAT-
            ,TREEFIEL.DAT-

    /DCL=   ("$@SYS$LOGIN:CMOS.COM                1"-
    /DCL=   ("$@SYS$LOGIN:FIELD.COM                1"-
```

A.5.2 Execution Control File

The following is an example of one task in the ECF file created above.

```
task " NOT TOTNWL MASKLR NWELL"-
    /INPUT =(TOTNWL.DAT-
            ,MASKLR.DAT-
            )-
    /OUTPUT=(NWELL.DAT-
            )-
    /DCL=   ("$@SYS$LOGIN:FIELD.COM              16"-
            )
```

A.5.3 Command File

Each page of the .COM file corresponds to an .ECF task, such as the one above. At the beginning of the .COM file, there is a $ GOTO 'P1', which explains how the correct step gets executed.

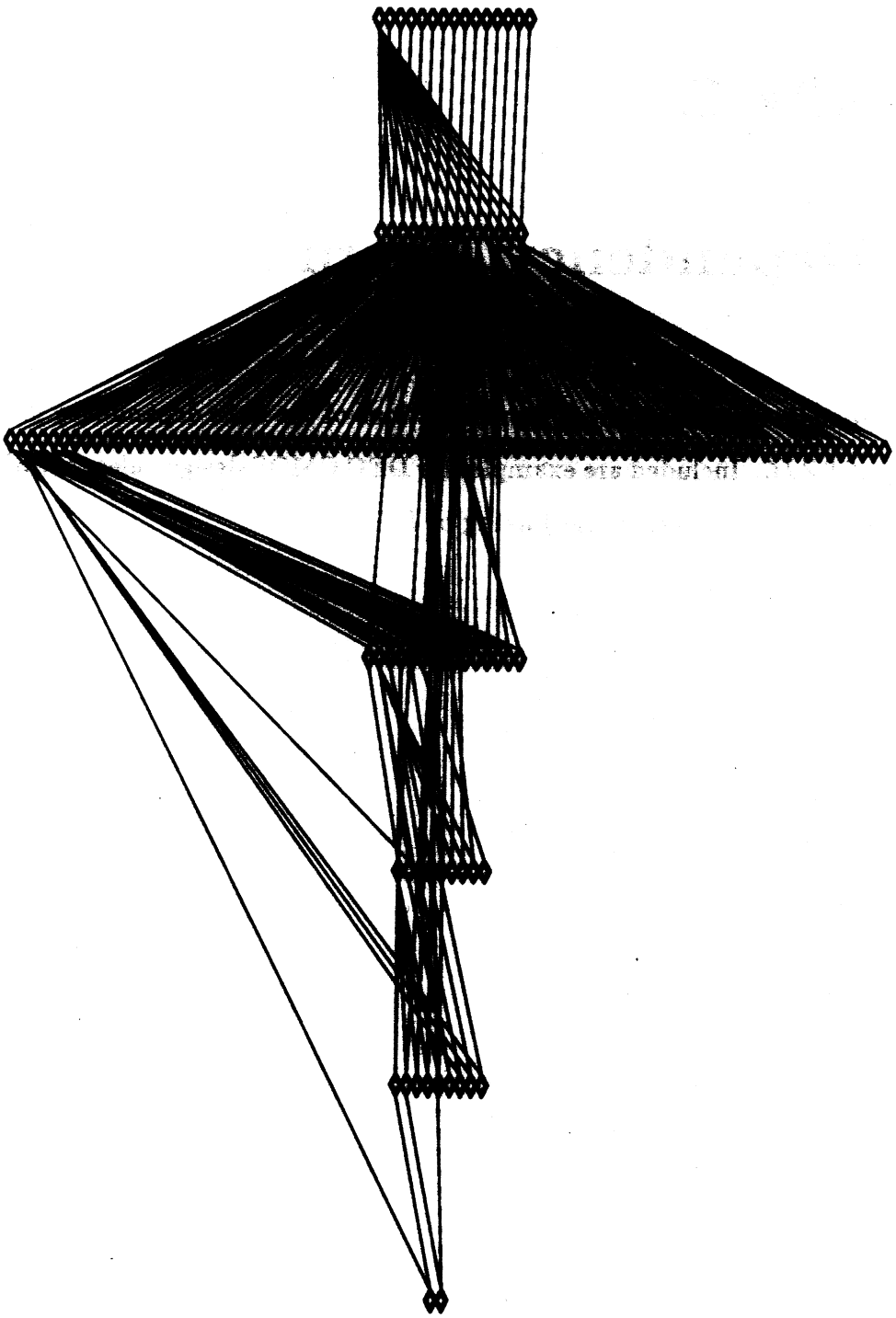
```
$            16:
$ !  NOT TOTNWL MASKLR NWELL
$ !
$SET PROCESS/NAME=              16GDSIN
$RUN SEGCAD$ECAD:LOGICAL
    3  TOTNWL    MASKLR    NWELL          1000  MIC          0

$IF .NOT. $STATUS THEN GOTO LQUIT
$OUTPUT:
$IF P2 .EQS. "OUTPUT" THEN GOTO LQUIT
$EXIT
```

Appendix B

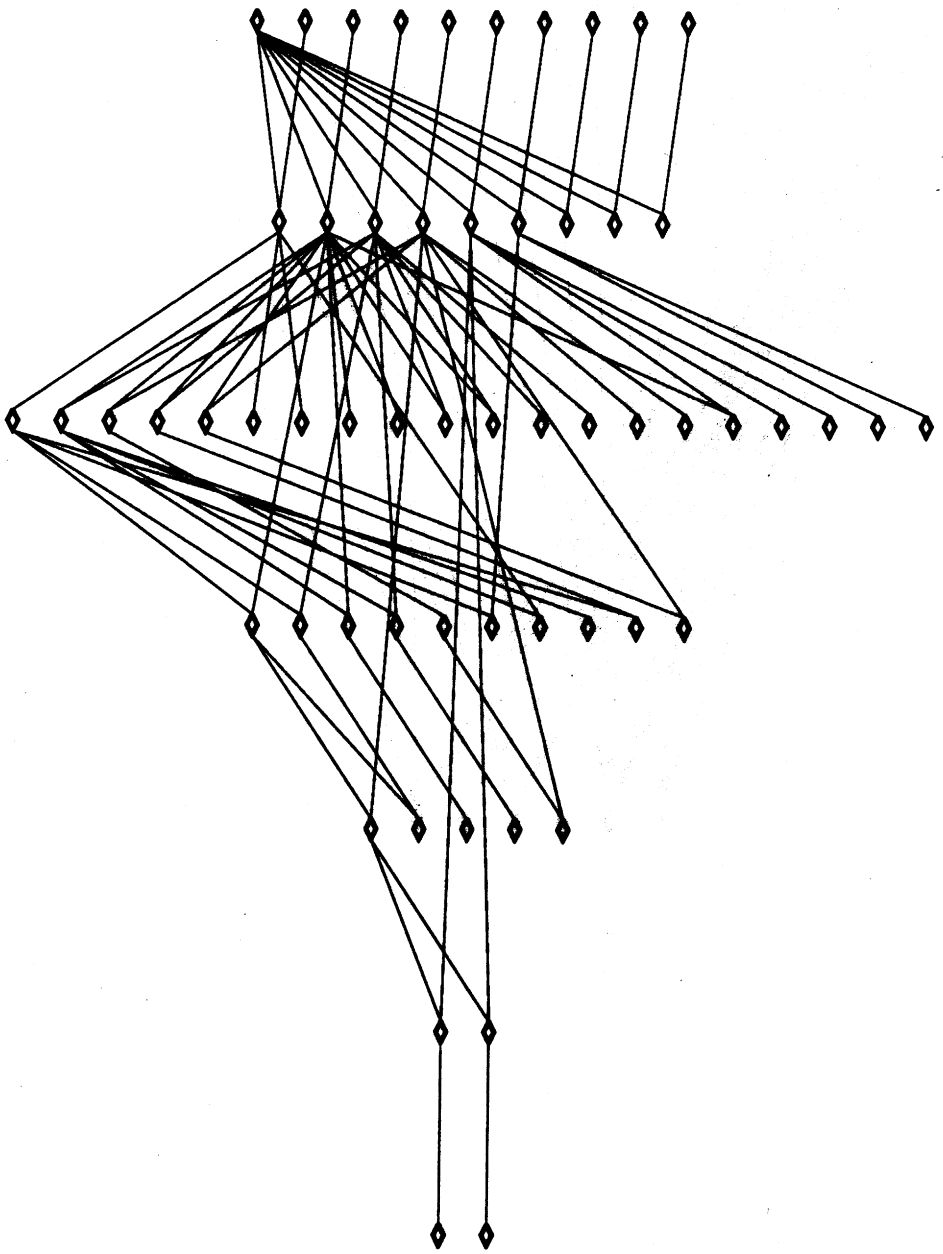
Data Dependency Graphs

This appendix contains printed representations of the data dependency graphs used in the testing of *EPIC*. Included are examples for DEC CMOS design rules, MOSIS CMOS design rules, and the compilation and linking of *EPIC*.

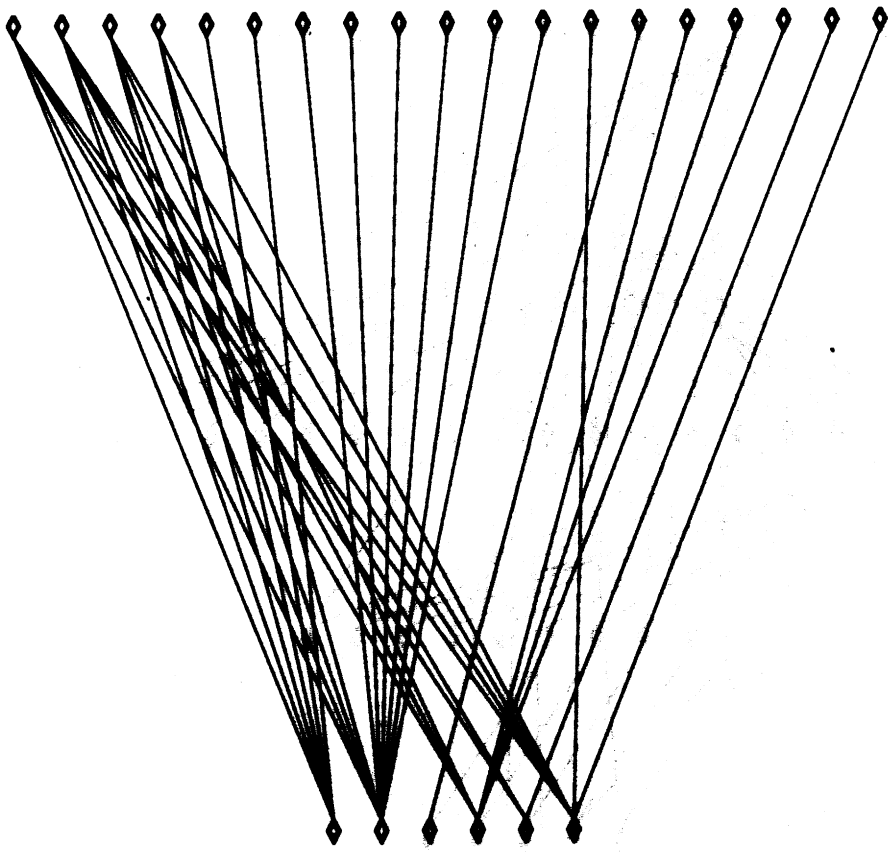
DEC CMOS DRC Data Dependency Graph

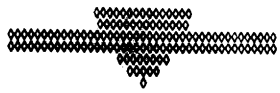


MOSIS CMOS DRC Data Dependency Graph



"make EPIC" Data Dependency Graph





Appendix C

Data from the testing of *EPIC*

This appendix contains raw statistics generated by *EPIC* for the test runs with a varying number of processors. Each section consists of all the data for a single application. Each subsection has a table of statistics and a graphical log for a single run. The leftmost column of the graphical log indicates the elapsed time at several points on the Y-axis. Each vertical column represents the activity of a single slave. Each diamond is the execution of a task. The height of the diamond is proportional to the amount of time it took to execute the corresponding task. Each line segment between two tasks represents a data dependency between those tasks, and roughly corresponds to a file transfer.

C.1 DRACULA with DEC CMOS rules

C.1.1 Serial DRACULA on a VAX 11/780 computer

Buffered I/O count:	8839	Peak working set size:	8060
Direct I/O count:	66129	Peak page file size:	19635
Page faults:	239201	Mounted volumes:	0
Charged CPU time:	04:00:11.84	Elapsed time:	05:05:50.41

C.1.2 Parallel DRACULA on three VAXclustered VAX 11/780 computers

9-APR-1986 07:22:43.28

Accounting information (for the "MASTER" process):

Buffered I/O count:	6728	Peak working set size:	8000
Direct I/O count:	12629	Peak virtual size:	18898
Page faults:	80985	Mounted volumes:	0
Images activated:	544		
Elapsed CPU time:	0 00:23:23.61		
Connect time:	0 02:52:51.39		

Total "SLAVE" statistics:

Elapsed seconds:	32717
CPU seconds:	14210

C.1.3 *EPIC* using one VAX 11/780 computer

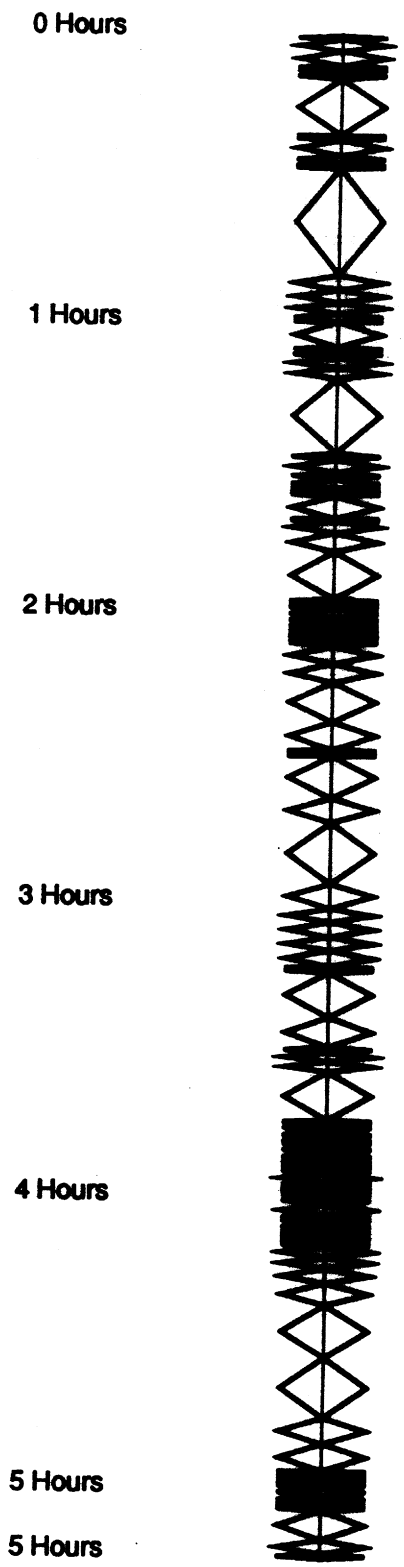
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0

29-MAR-1986	18:45:38.05	BUFFIO:	3549
ELAPSED:	05:21:28.13	DIRIO:	487
CPU:	0:01:13.19	FAULTS:	720

Subprocess statistics (all times in seconds)

Node	File CPU	File Time	Exec CPU	Exec Time	Idle Time	Total CPU	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	288	466	18808	289	0	0	0	1474	4427	20360	1956	2364
Slave	0	1	15108	18698	105	15120	125	0	222	11678	64707	20713	8000	557686
Total	0	1	15396	19139	18913	15409	125	0	[illegible]	[illegible]	69134	41073	9956	560050

DECCMOS.ECF run on 29-MAR-1986 18:45:45



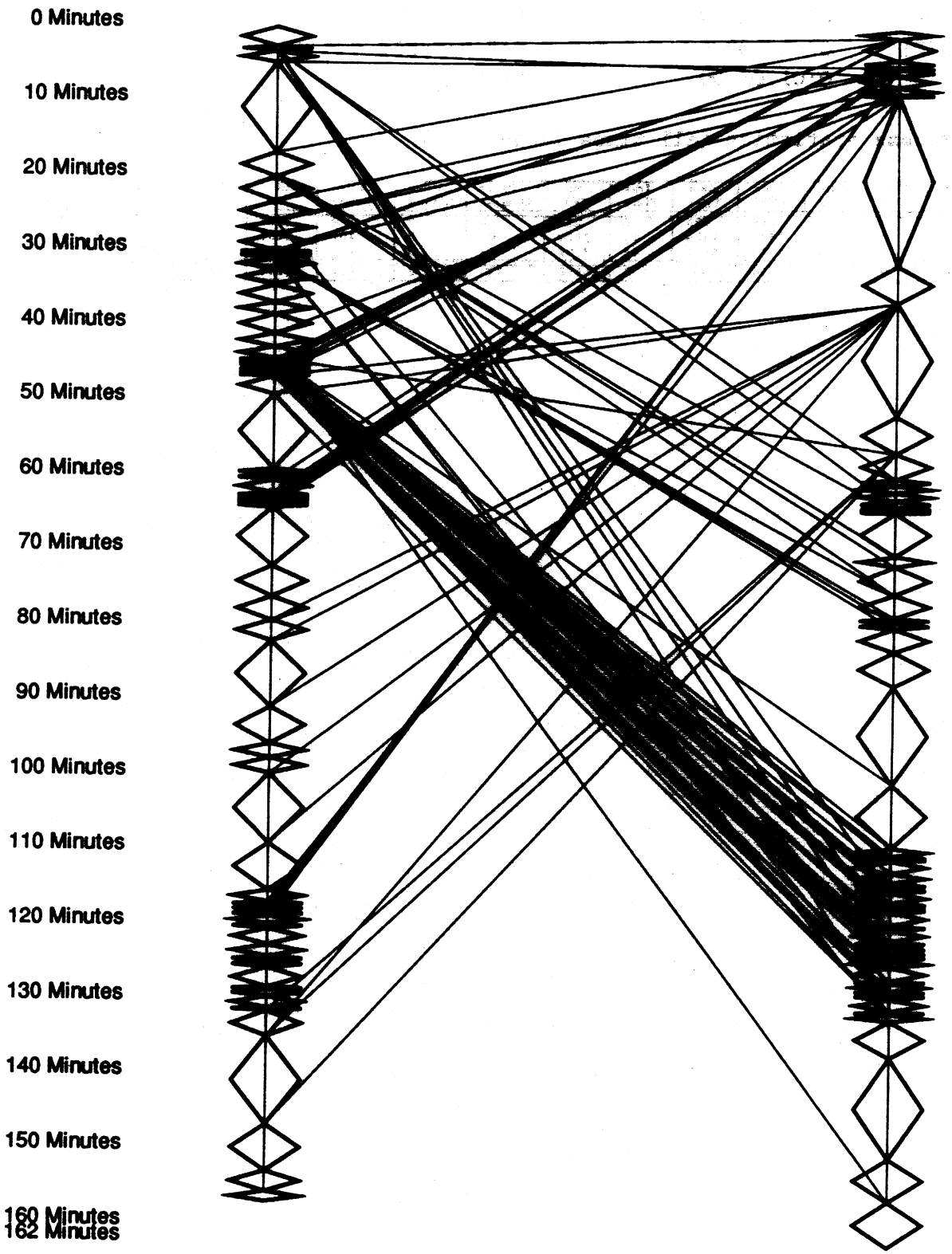
C.1.4 *EPIC* using two VAXclustered VAX 11/780 computers

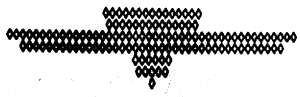
```
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
28-MAR-1986  07:21:18.95          BUFIO: 3170
ELAPSED:     02:51:09.61          DIRIO: 858
CPU:         0:01:07.09           FAULTS: 702
```

Subprocess statistics (all times in seconds)

Node	File CPU	File Time	Exec CPU	Exec Time	Idle Time	Total CPU	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	286	[illegible]	9734	[illegible]	0	0	0	[illegible]	4549	20360	1955	2374
Slave1	0	0	7632	9634	91	7688	61	0	107	[illegible]	34087	20585	8000	274384
Slave2	0	0	7432	9847	54	7439	64	0	115	[illegible]	29976	20585	8000	288740
Total	0	0	15350	19503	[illegible]	15365	125	0	222	[illegible]	68612	61530	17955	565498

DECCMOS.ECF run on 28-MAR-1986 07:21:27





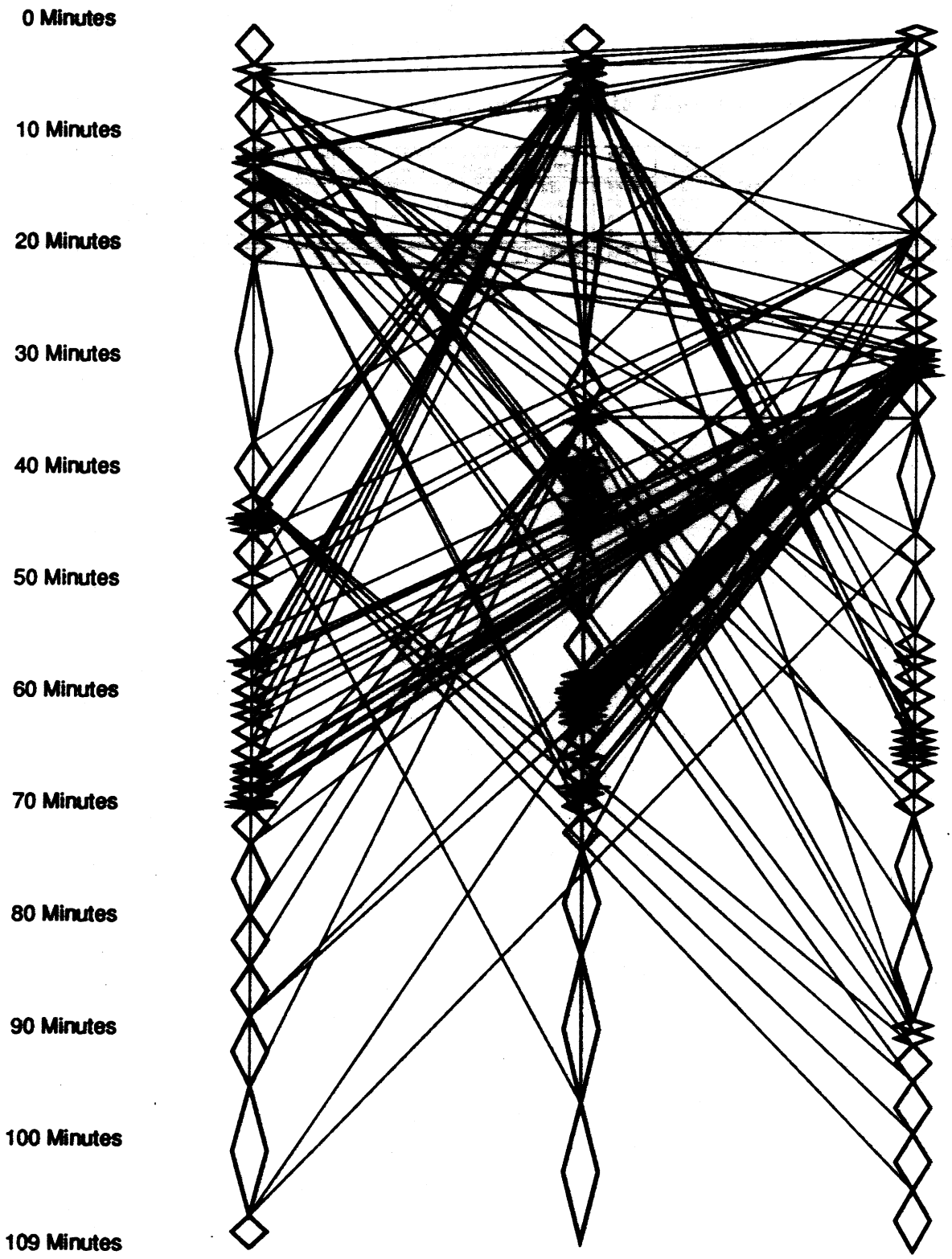
C.1.5 *EPIC* using three VAXclustered VAX 11/780 computers

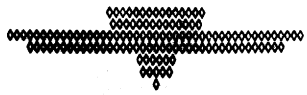
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
27-MAR-1986 06:28:09.75 BUFIO: 3179
ELAPSED: 01:57:56.11 DIRIO: 899
CPU: 0:01:06.27 FAULTS: 709

Subprocess statistics (all times in seconds)

Node	File CPU	File Time	Exec CPU	Exec Time	Idle Time	Total CPU	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	287	485	6526	288	0	0	0	1274	4613	20360	1955	2374
Slave1	0	0	4855	6507	57	4860	44	0	80	4854	25753	20585	8000	219370
Slave2	0	0	5122	6427	68	5127	42	0	68	3592	19198	20585	8000	138945
Slave3	0	0	5019	6499	66	5023	39	0	74	3661	24168	20577	8000	197093
TOTAL	0	0	15313	19918	6723	15525	125	0	222	13381	73732	82107	25955	557782

DECCMOS.ECF run on 27-MAR-1986 06:28:18





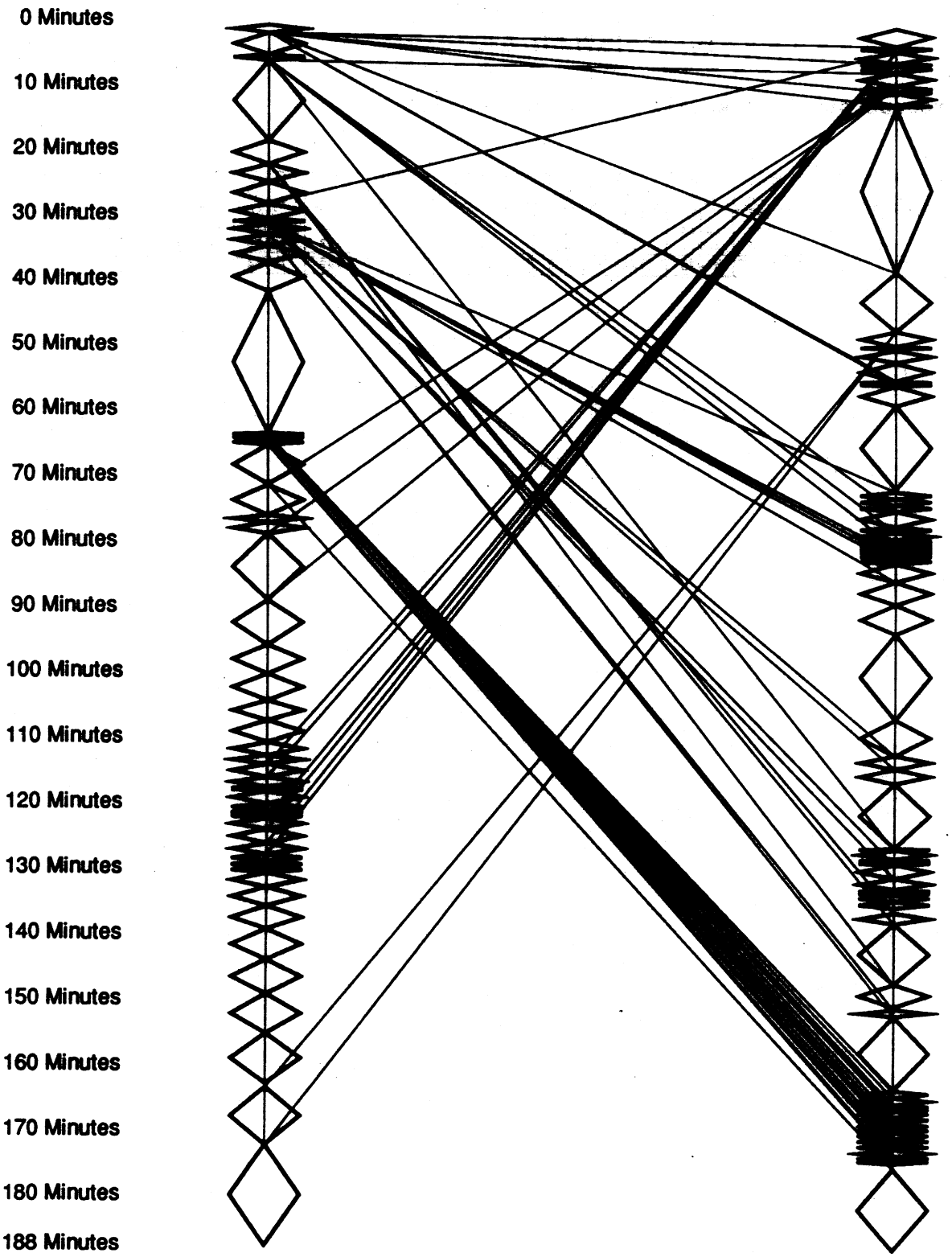
C.1.6 *EPIC* using two independent VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
 9-APR-1986 03:31:27.82      BUFIO:  3434
 ELAPSED:   03:15:42.65      DIRIO:  508
 CPU:        0:01:23.19      FAULTS: 830
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	274	460	11270	276	0	0	0	1150	3922	20282	1956	2366
Slave1	57	210	7514	10943	61	7577	52	81	80	[illegible]	37374	20585	8000	284208
Slave2	161	881	7635	10227	153	7803	73	108	113	[illegible]	37444	20585	8000	282679
Total	218	1091	15423	[illegible]	11484	15656	125	189	[illegible]	[illegible]	78740	61402	17956	569253

DECCMOS.ECF run on 9-APR-1986 03:31:36



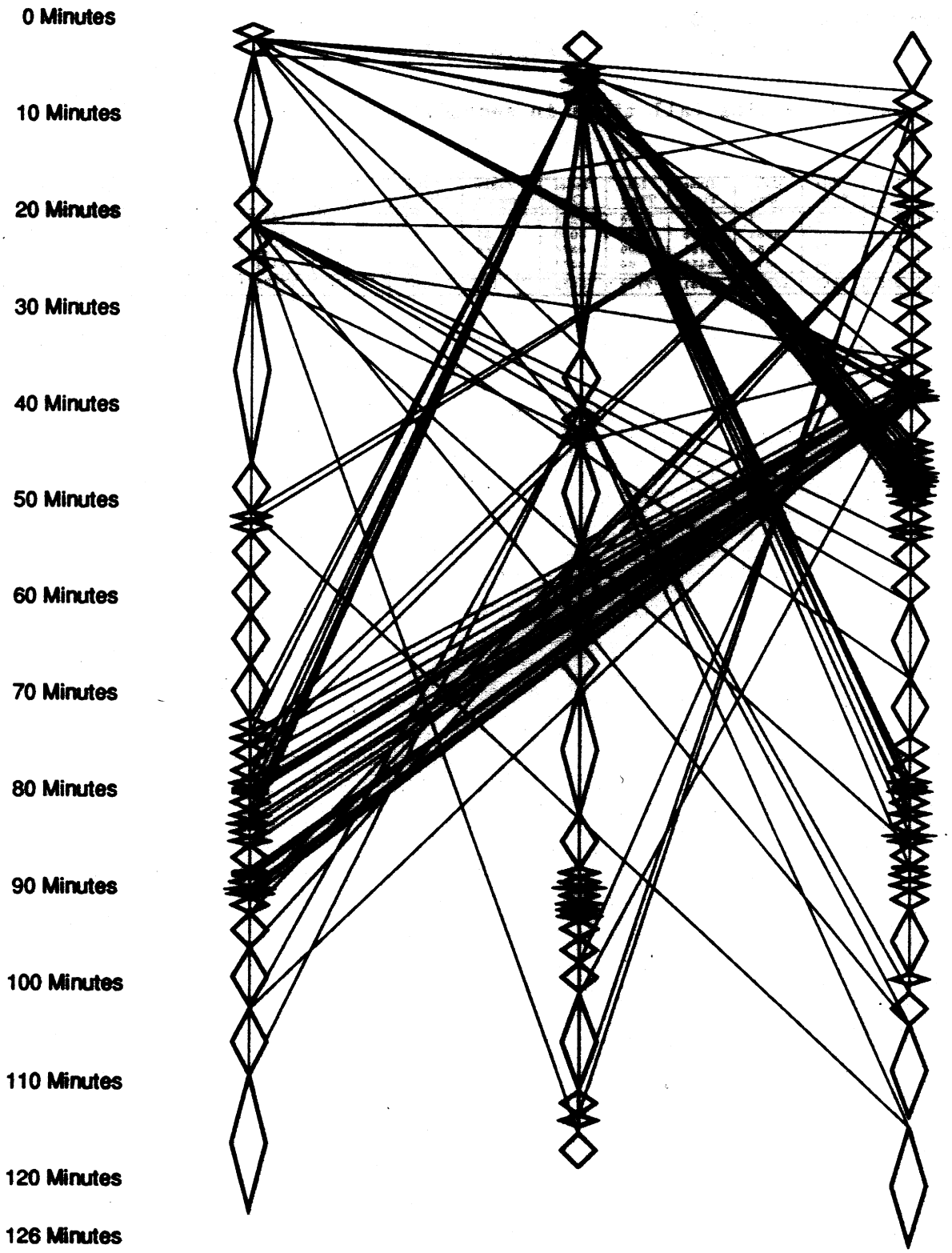
C.1.7 *EPIC* using three independent VAX 11/780 computers

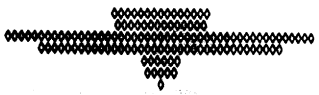
```
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
 7-APR-1986 06:43:13.96          BUFIO:  3478
 ELAPSED:   02:13:40.10          DIRIO:  502
 CPU:       0:01:18.17           FAULTS: 827
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	272	[illegible]	7376	272	0	0	0	[illegible]	3763	20232	1956	2366
Slave1	45	150	5081	7156	36	5000	37	65	45	[illegible]	25005	20585	8000	175632
Slave2	137	673	5160	9731	109	5304	58	87	89	[illegible]	21092	20577	8000	222310
Slave3	101	564	4671	6453	65	4976	30	53	43	7101	26876	20585	8000	165362
Total	283	1387	[illegible]	[illegible]	7788	[illegible]	125	[illegible]	177	[illegible]	76826	81979	25956	565670

DECCMOS.ECF run on 7-APR-1986 06:43:21





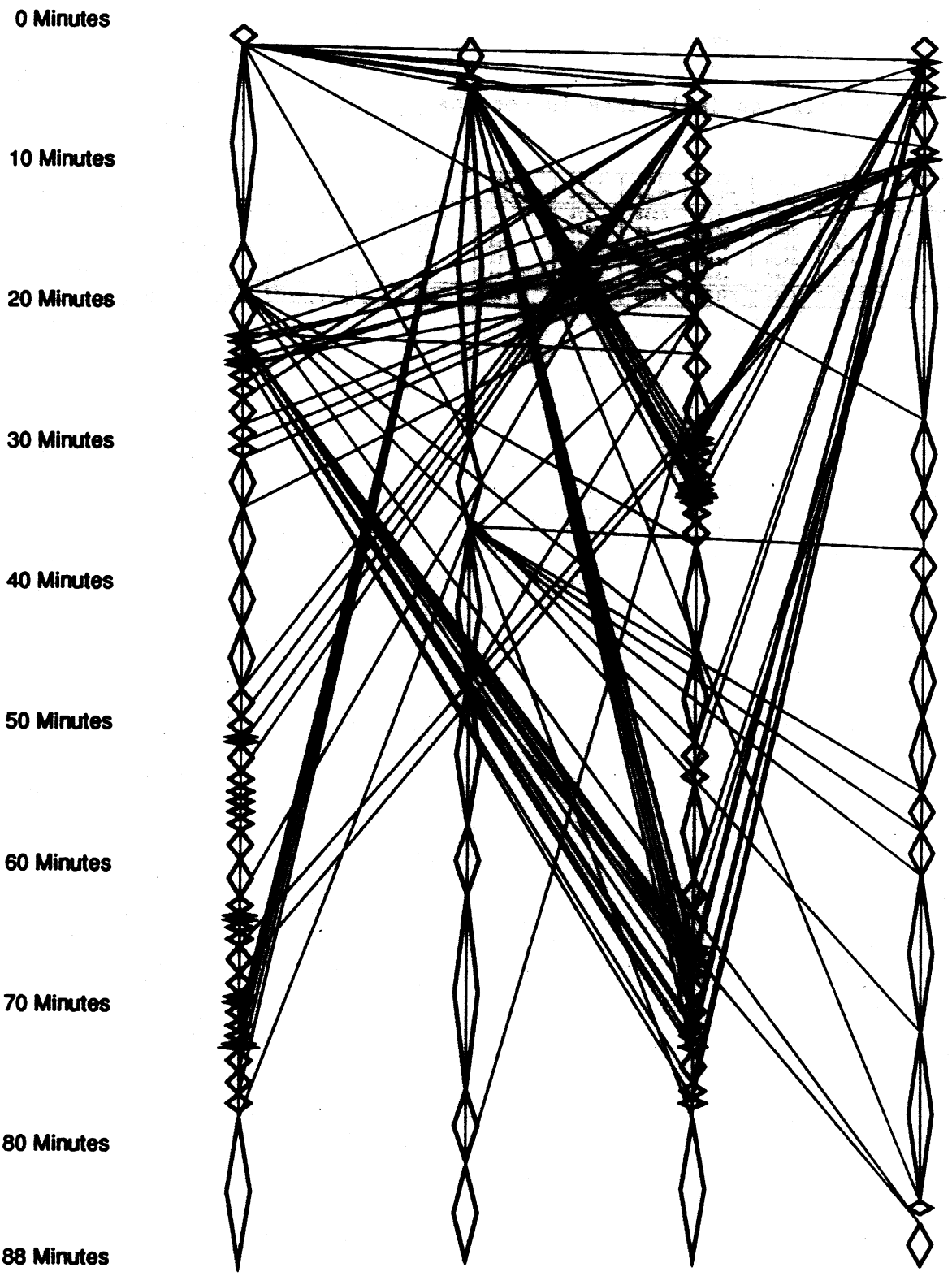
C.1.8 *EPIC* using four independent VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
 6-APR-1986 02:12:26.93          BUFIO:  3379
 ELAPSED:   01:36:10.71          DIRIO:  398
 CPU:        0:01:20.47          FAULTS: 837
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	276	450	5298	278	0	0	0	1150	3781	20232	1956	2367
Slave1	47	153	3606	5070	56	3659	48	73	66	3887	16503	20585	8000	146560
Slave2	84	448	3976	4764	74	4063	22	43	30	5843	17962	20577	9130	167037
Slave3	60	373	3830	4857	38	3892	11	20	12	4418	16357	20585	8000	67148
Slave4	115	572	3568	4540	142	3827	44	74	65	7459	17173	20577	8000	190293
Total	306	1546	15256	19700	5605	15579	125	210	173	22757	71776	102556	35086	573405

DECCMOS.ECF run on 6-APR-1986 02:12:44



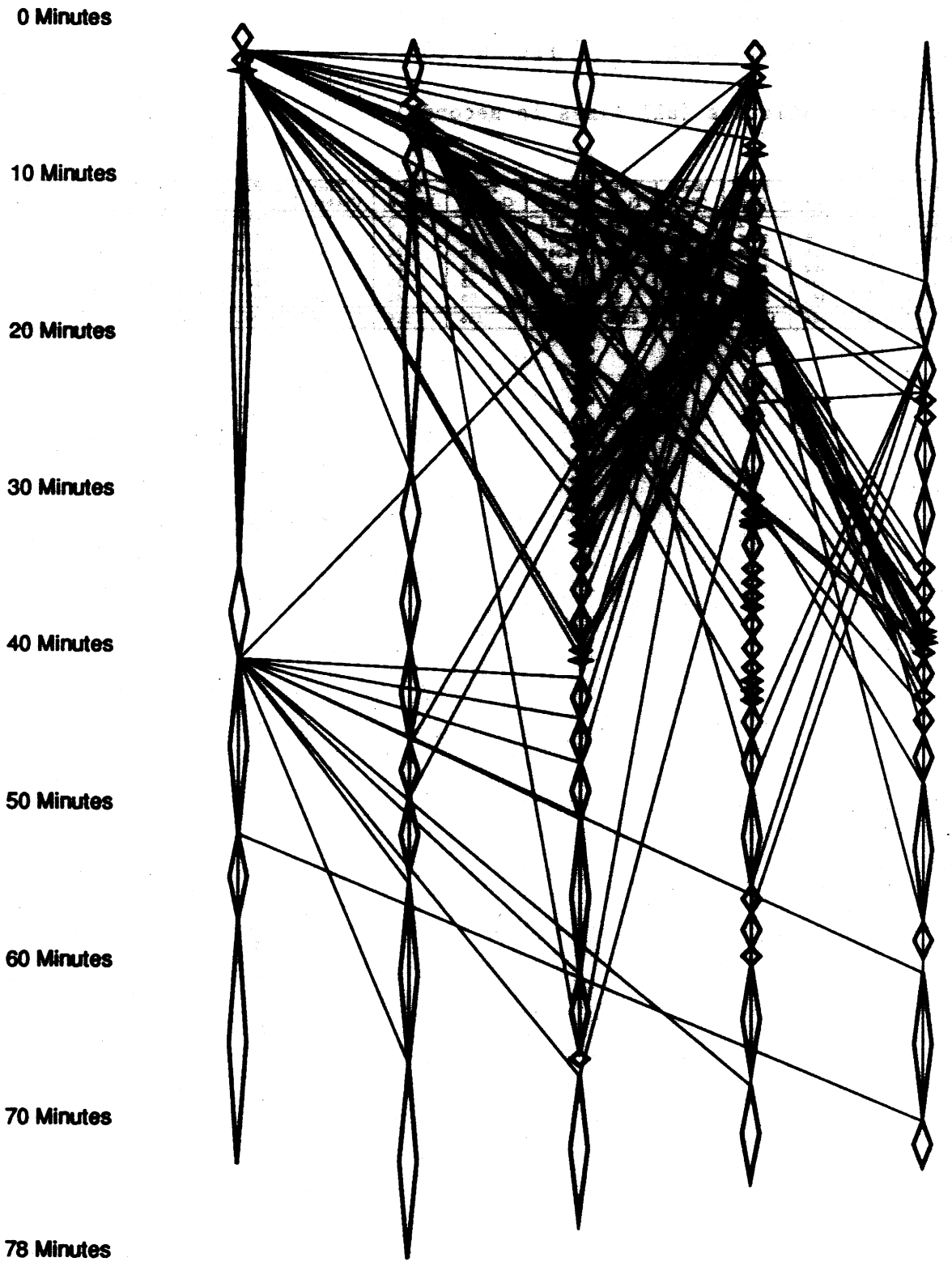
C.1.9 *EPIC* using five independent VAX 11/780 computers

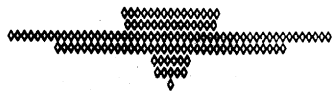
```
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0
 12-APR-1986 06:22:44.11        BUFIO:  3505
 ELAPSED:    01:25:58.26        DIRIO:  418
 CPU:         0:01:10.51        FAULTS: 865
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	272	[illegible]	[illegible]	274	0	0	0	[illegible]	3608	20282	1956	2367
Slave1	16	66	[illegible]	[illegible]	7	2707	9	16	9	[illegible]	14909	20585	8000	79186
Slave2	59	366	[illegible]	[illegible]	22	3867	12	26	14	[illegible]	13547	20585	8000	108445
Slave3	80	492	[illegible]	[illegible]	71	2079	38	57	53	[illegible]	12969	20577	8000	78610
Slave4	102	582	[illegible]	[illegible]	74	3155	45	75	67	[illegible]	14065	20577	9067	230234
Slave5	83	482	[illegible]	[illegible]	39	[illegible]	22	42	28	[illegible]	12800	19857	4096	96186
Total	340	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	125	[illegible]	[illegible]	[illegible]	[illegible]	122413	39119	595028

DECCMOS.ECF run on 12-APR-1986 06:22:51





C.1.10 *EPIC* using six independent VAX 11/780 computers

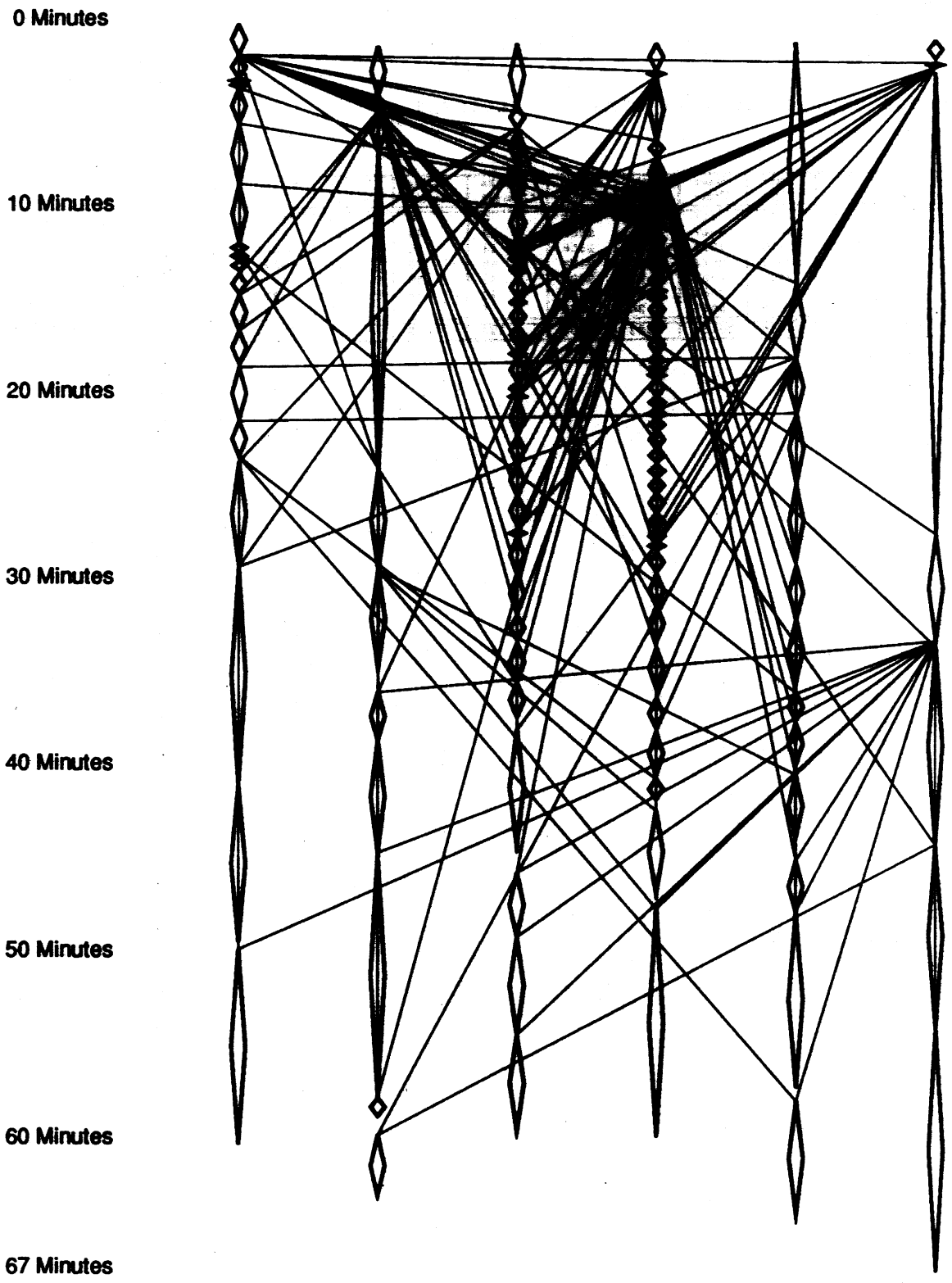
MASTER Statistics for EPIC run using ECF file DECCMOS.ECF
EPIC Version V1.0

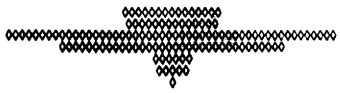
10-APR-1986 06:12:36.06		BUFIO:	3529
ELAPSED:	01:14:21.67	DIRIO:	405
CPU:	0:01:14.59	FAULTS:	879

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	272	417	4852	272	0	0	0	1885	3432	20282	1956	2366
Slave1	80	130	2004	3447	51	2088	19	36	21	2164	11960	20585	8000	131915
Slave2	76	445	2478	3310	18	2556	11	25	12	4886	14704	20585	8000	91191
Slave3	110	532	2463	2944	100	2578	43	78	55	6525	10487	20577	9019	155079
Slave4	86	485	2402	2906	87	2492	32	52	44	4972	10587	20577	8000	82328
Slave5	80	456	2807	3363	20	2888	18	26	14	4226	12186	19857	4096	66539
Slave6	52	308	2910	3687	26	2995	7	14	6	3725	14984	19857	4096	92317
Total	434	2358	14336	20154	4852	13701	125	226	154	27938	78540	142270	43167	621735

DECCMOS.ECF run on 10-APR-1986 06:12:43





C.2 Compiling and Linking $\mathcal{EPIC}$

The following statistics were generated by VMS after compiling and linking $\mathcal{EPIC}$ and its preprocessors.

```
Accounting information:
 Buffered I/O count:         962  Peak working set size:       3886
 Direct I/O count:          2599  Peak virtual size:           7904
 Page faults:              31011  Mounted volumes:                0
 Images activated:            26
 Elapsed CPU time: 00:06:48.74
 Connect time:     00:10:19.12
```

C.2.1 *EPIC* using one VAX 11/780 computer

MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF

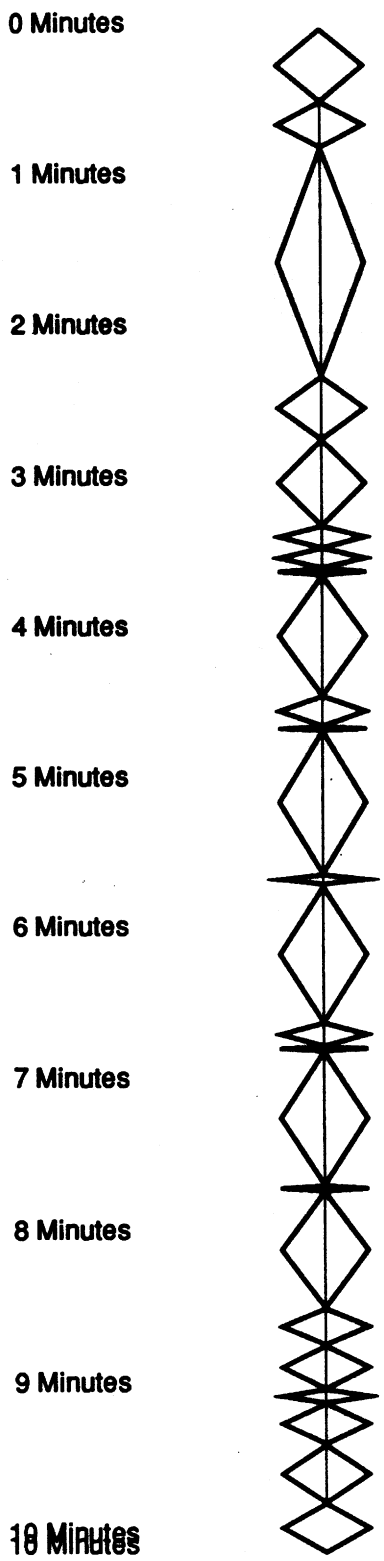
EPIC Version V1.0

30-MAR-1986	13:50:40.42	BUFIO:	732
ELAPSED:	00:10:30.09	DIRIO:	81
CPU:	0:00:14.42	FAULTS:	231

Subprocess statistics (all times in seconds)

Node	File CPU	File Time	Exec CPU	Exec Time	Idle Time	Total CPU	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	0	0	[illegible]	1	3	0	0	[illegible]	21	[illegible]	201	270
Slave	0	0	404	384	19	[illegible]	25	0	[illegible]	[illegible]	2677	[illegible]	[illegible]	[illegible]
Total	0	0	404	[illegible]	[illegible]	[illegible]	25	0	[illegible]	[illegible]	[illegible]	[illegible]	4198	31504

MAKEEPIC.ECF run on 30-MAR-1986 13:50:48.10



C.2.2 *EPIC* using two VAXclustered VAX 11/780 computers

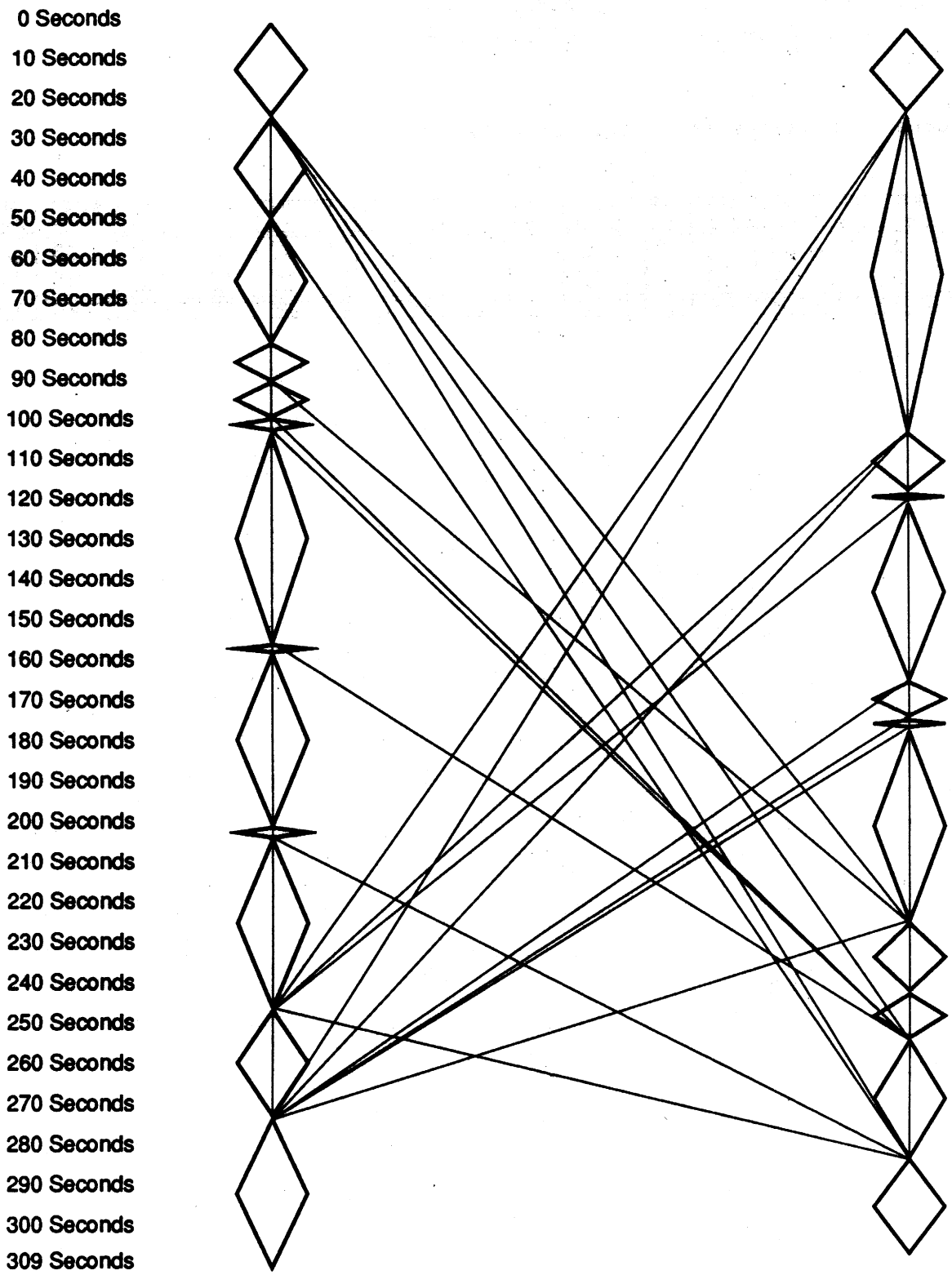
MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF
EPIC Version V1.0

1-APR-1986	02:55:59.84	BUFIO:	923
ELAPSED:	00:05:39.54	DIRIO:	101
CPU:	0:00:16.24	FAULTS:	248

Subprocess statistics (all times in seconds)

Node	File CPU	File Time	Exec CPU	Exec Time	Idle Time	Total CPU	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	0	0	334	1	0	0	0	30	14	2283	291	272
Slave1	0	0	207	304	10	200	13	0	27	402	1310	4541	2729	17120
Slave2	0	0	195	200	14	196	12	0	25	505	1439	5365	3857	14207
Total	0	0	402	504	358	408	25	0	52	1027	2763	12180	6877	31599

MAKEEPIC.ECF run on 1-APR-1986 02:56:17



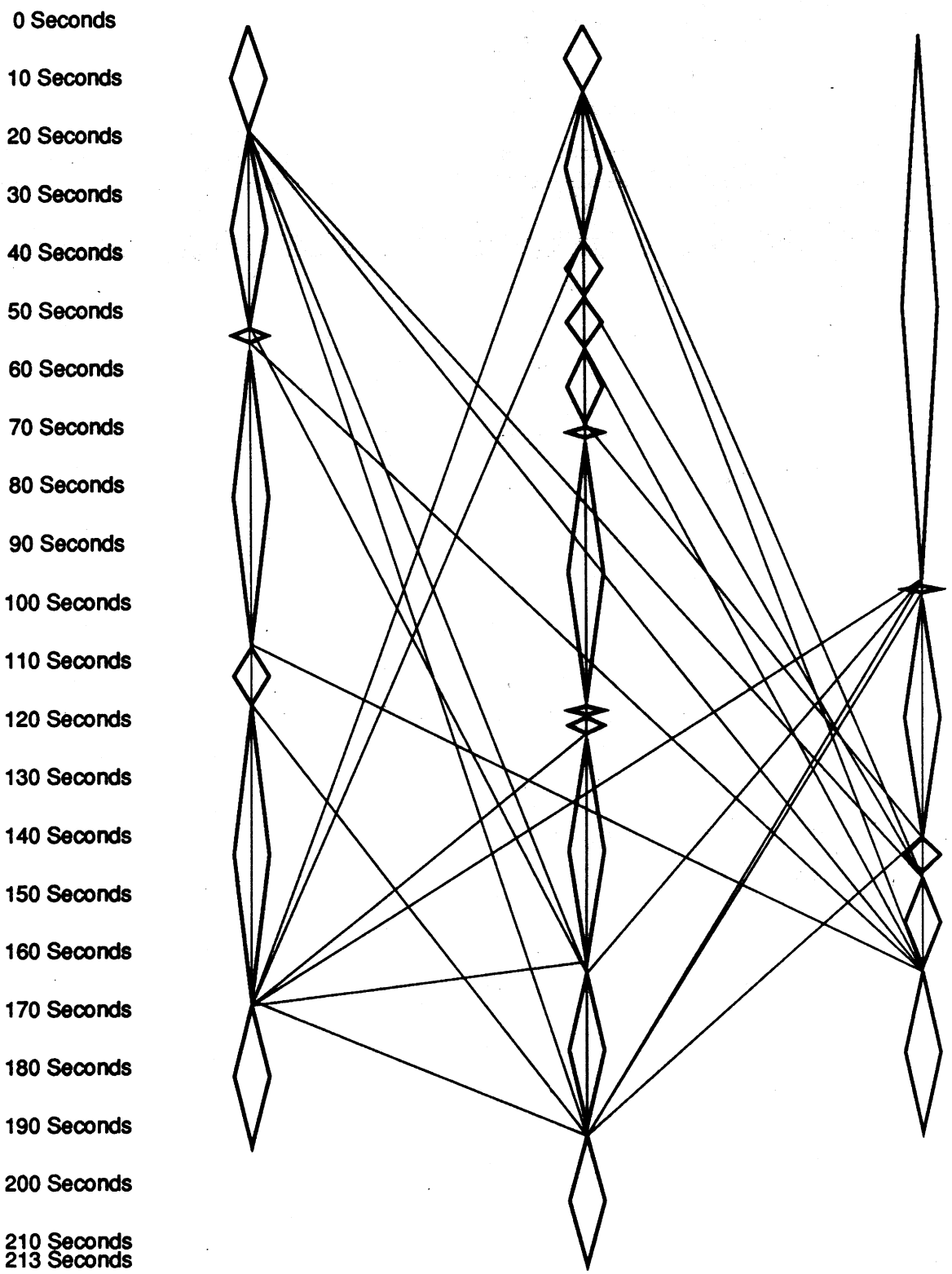
C.2.3 *EPIC* using three VAXclustered VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF
EPIC Version V1.0
 6-APR-1986 23:27:38.08          BUFIO:  926
 ELAPSED:   00:04:12.70          DIRIO:  166
 CPU:        0:00:16.29          FAULTS: 259
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Complt	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	0	0	246	1	0	0	8	[illegible]	14	2288	291	270
Slave1	0	0	141	198	6	148	7	0	12	291	702	4541	[illegible]	10804
Slave2	0	0	143	204	15	145	12	0	26	443	1185	4605	3147	13631
Slave3	0	0	118	179	8	120	6	0	14	205	900	5365	4013	7171
Total	0	0	402	581	275	400	25	0	52	[illegible]	2810	16794	10491	31876

MAKEEPIC.ECF run on 6-APR-1986 23:27:45



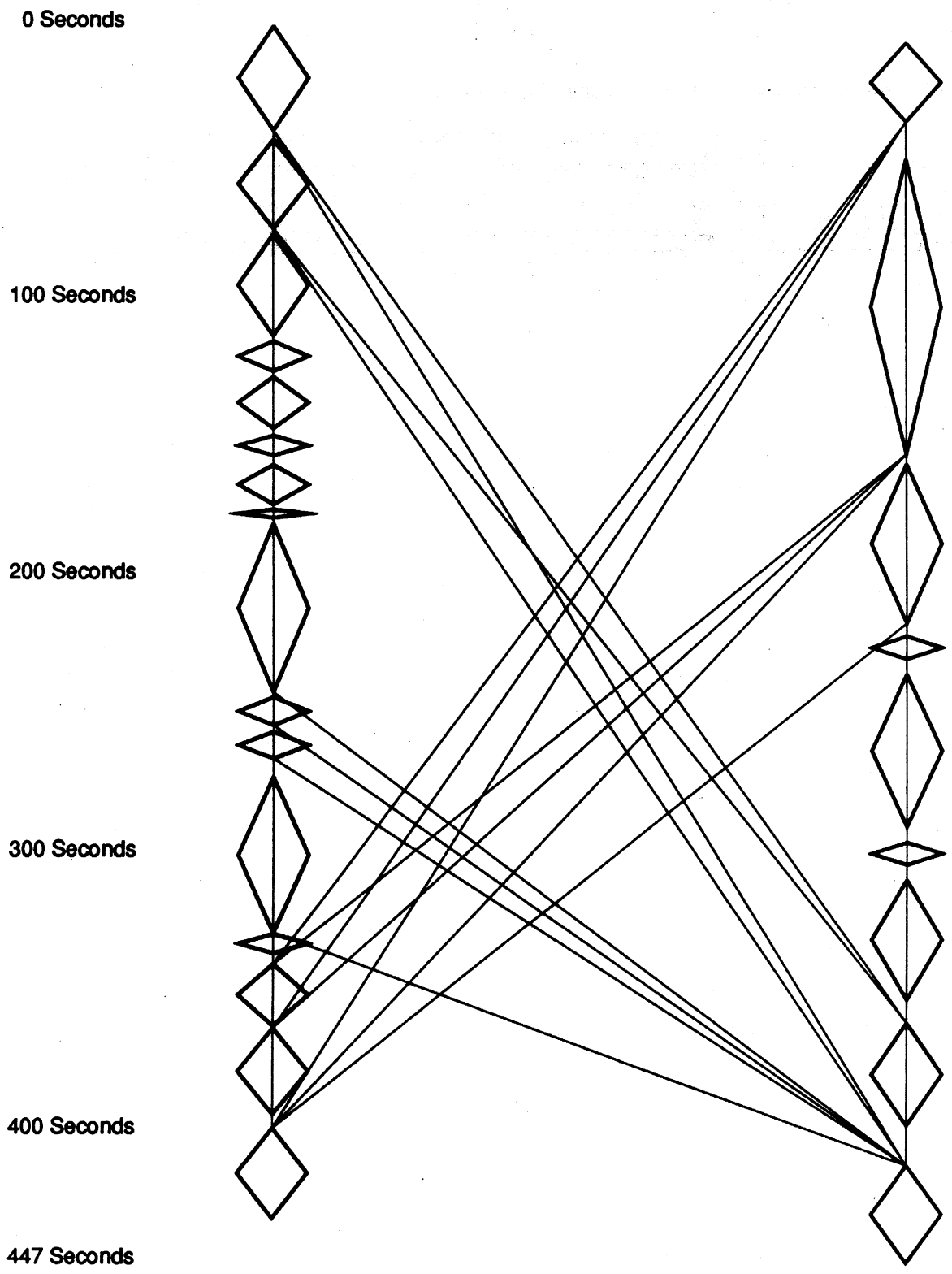
C.2.4 *EPIC* using two independent VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF
EPIC Version V1.0
 3-APR-1986 02:27:38.02          BUFIO:        1028
 ELAPSED:   00:07:57.36          DIRIO:        51
 CPU:        0:00:18.33          FAULTS:       286
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Compit	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkgset Peak	Page Faults
Master	0	0	0	0	471	1	0	0	0	30	13	2155	201	270
Slave1	17	65	194	356	14	214	16	31	14	964	1836	4605	3178	19402
Slave2	17	98	208	320	15	237	9	22	10	1117	1854	5873	3890	15340
Total	34	163	402	685	500	442	25	53	24	2111	3203	12133	7350	35012

MAKEEPIC.ECF run on 3-APR-1986 02:27:46



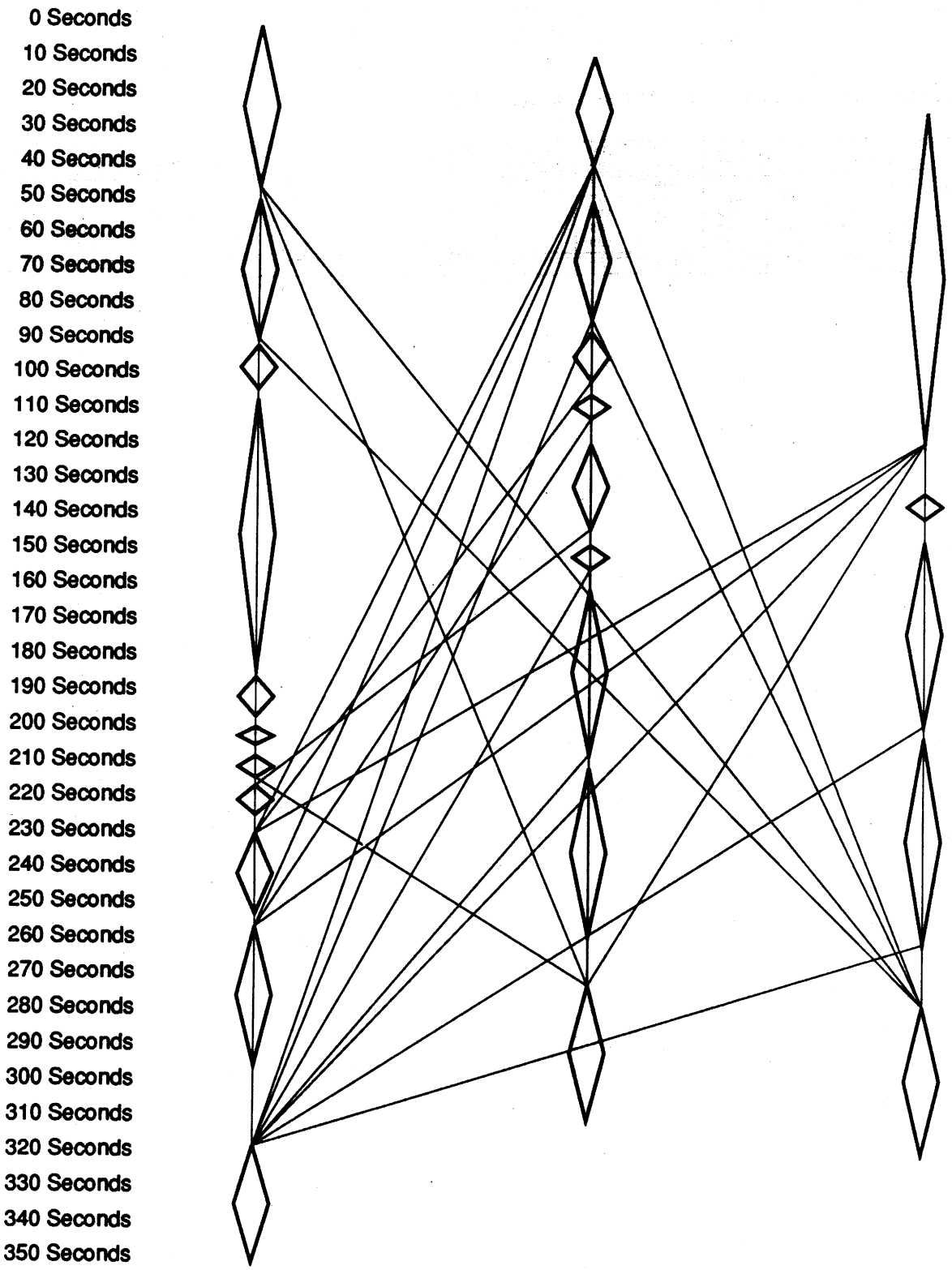
C.2.5 *EPIC* using three independent VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF
EPIC Version V1.0
 3-APR-1986 02:06:48.99          BUFIO:        1067
 ELAPSED:    00:06:32.75         DIRIO:        64
 CPU:         0:00:18.25         FAULTS:       294
```

Subprocess statistics (all times in seconds)

Node	File Cpu	File Time	Exec Cpu	Exec Time	Idle Time	Total Cpu	Tasks Compl	Files Moved	Files Cached	Buffered IO	Direct IO	Virt Peak	Wkset Peak	Page Faults
Master	0	0	0	0	385	1	0	0	0	58	13	2283	291	272
Slave1	16	72	120	281	10	147	11	26	11	780	1696	4541	2790	13139
Slave2	10	85	150	217	11	162	5	13	2	670	700	5365	4011	10630
Slave3	13	93	124	208	18	130	9	20	8	800	813	4187	2653	12673
Total	39	250	403	701	424	440	25	61	16	2307	3282	16548	9745	36714

MAKEEPIC.ECF run on 3-APR-1986 02:06:57



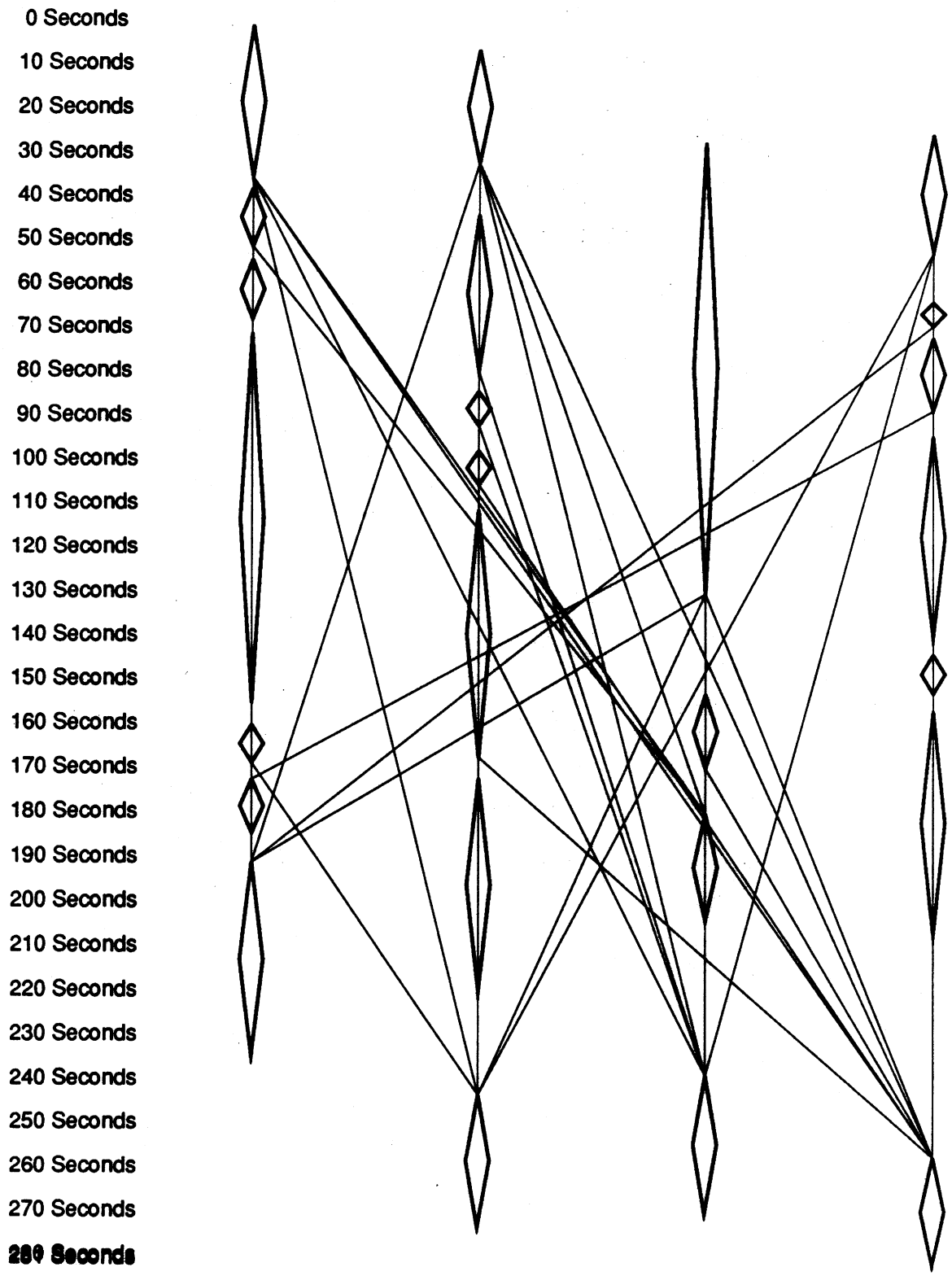
C.2.6 *EPIC* using four independent VAX 11/780 computers

```
MASTER Statistics for EPIC run using ECF file MAKEEPIC.ECF
EPIC Version V1.0
 3-APR-1986 02:16:04.72          BUFIO:        1086
 ELAPSED:    00:05:32.38         DIRIO:        44
 CPU:         0:00:17.95         FAULTS:       311
```

Subprocess statistics (all times in seconds)

Node	File Opn	File Time	Exec Opn	Exec Time	Idle Time	Total Cpu	Tasks Compltd	Files Moved	Files Cached	Buffrd IO	Direct IO	Virt Peak	Wkset Peak	Page Faults
Master	0	0	0	0	329	1	0	0	0	30	12	2585	281	272
Slave1	8	41	87	106	10	97	7	16	3	474	819	4541	3080	8867
Slave2	11	96	114	172	14	128	7	17	2	748	780	4157	2727	10067
Slave3	9	110	85	150	9	95	4	12	4	622	798	5965	3832	9730
Slave4	12	123	116	143	12	100	7	10	4	645	835	4624	3222	11247
Total	40	370	402	603	365	451	25	54	13	2719	3164	20670	13369	37183

MAKEEPIC.ECF run on 3-APR-1986 02:16:12



Appendix D

$\mathcal{EPIC}$ Messages

This Appendix contains all the messages sent as control communication. They effectively define the architecture of the software behind $\mathcal{EPIC}$.

D.1 Messages sent from user to monitor

EXIT
: Terminate the MONITOR program. This does not affect the operation of the master.

CREATE/MASTER **node com-file ecf-file working-directory cluster-list**
: Create a master

CREATE/SLAVE **name node com-file working-directory**
: Tell the master to create a slave and put it in the database

MONITOR **master's-node**
: Establish communication with an already-existing master

KILL
: Tell the master to terminate the computation and generate the log files

KILL/SLAVE **slave-node slave-name**
: Tell the master to terminate the slave and insert its task (if any) into the ready queue

SET/CLUSTER = **(node1, node2 ...)**
: Tell the master to define a set of nodes to be clustered together

SET/REFRESH = **time interval**
: Tell the master to set the interval at which the process rate is refreshed

D.2 Messages sent from monitor to master

SET/CLUSTER = (node1, node2 ...)
Define a set of nodes to be clustered together

SET/REFRESH = time interval
Set the interval at which the process rate is refreshed

EXIT
Terminate the computation and generate the log files

KILL/NAME = slave's name /NODE = slave's node
Terminate the slave and insert its task (if any) into the ready queue

CREATE/SLAVE name node command-file working-directory
Create a slave and put it in the database

D.3 Messages sent from master to monitor

MESSAGE msg
Allows the master to put an arbitrary message on the monitor's screen

STATUS line-number contents
Send statistics line describing slave's subprocess' CPU usage to the monitor's process display

DONE
Indicates to the monitor that the whole computation has completed.

D.4 Messages sent from master to slave

START task-name /INPUT=(in1, in2 ...) /OUTPUT=(out1, out2 ...) /DCL=(dcl1, dcl2 ...)

start the task with the specified inputs, outputs and dcl commands

EXIT
Terminate the slave subprocess and exit

FREE
Charge elapsed time to the FREE counter, rather than the IDLE counter

SET/REFRESH = time interval
Set the interval at which the slave sends process line information

D.5 Messages sent from slave to master

COMPLETED
The slave completed its task

FAILED **reason**

The slave failed its task

STARTED

The slave has retrieved the input files and started the task

MESSAGE **msg**

Allows the slave to put an arbitrary textual message into the master's log file

STATUS **status line**

Send statistics line describing slave's subprocess' CPU usage to the master for the monitor's process display

FINAL **statistics**

Send final statistics about the slave's subprocess' CPU usage, etc., to the master.

Bibliography

[Allen 1983] J. Allen, *Introduction to VLSI Design*, M.I.T. Video Course Study Guide, Cambridge, MA, August 1983

[Arnold 1985] J.M. Arnold, *Parallel Simulation of Digital LSI Circuits*, MIT Laboratory for Computer Science, TR-333, 1985

[Arnold, Ousterhout 1982] M.H. Arnold, J.K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking", *Proceedings of the 19th Design Automation Conference*, June 1982

[Bier, Pleszkun 1985] G.E. Bier, A.R. Pleszkun, "An Algorithm for Design Rule Checking on a Multiprocessor", *Proceedings of the 22nd Design Automation Conference*, July 1985

[Chapman, Clark 1984] P.T. Chapman, K. Clark, Jr., "The Scan Line Approach to Design Rules Checking: Computational Experiences", *Proceedings of the 21st Design Automation Conference*, June 1984

[Deutsch, Newton 1984] J.T. Deutsch, A.R. Newton, "A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation", *Proceedings of the 21st Design Automation Conference*, June 1984

[Hammer 1986] M.E. Hammer, private communication, Hewlett Packard Corporation, February 1986

[Kung 1976] Kung, H.T., *Synchronized and Asynchronous Parallel Algorithms for Multiprocessors*, CMU-CS-76-150, June 1976

[Lee 1978] Lee, R.B., *Performance Characterization of Parallel Computations*, STAN-CSL-TR-158, September 1978

[Levitin 1986] S. Levitin, *A Multiprocessing Aproach to Circuit Extraction*, MIT Master's Thesis in preparation, 1986

[Marantz 1984] J.D. Marantz, *A Parallel Design Rule Checker*, DEC Internal Memo May, 1984

[McGrath 1985] E.J. McGrath, private communication, Digital Equipment Corporation

[McGrath, Whitney 1980] E.J. McGrath, T. Whitney, "Design Integrity and Immunity Checking: a New Look at Layout Verification and Design Rule Checking", *Proceedings of the 17th Design Automation Conference*, June 1980

[Mead, Conway 1980] C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison Wellesley, October 1980

[Mehrotra, Talukdar 1982] R. Mehrotra, S.N. Talukdar, *Task Scheduling on Multiprocessors*, Carnegie Mellon University DRC-18-55-82, December 1982

[Nagel 1975] L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL Memo No. ERL-M520, University of California, Berkeley, May 1975.

[Newton, Sangiovanni-Vincentelli 1983] A.R. Newton, A. Sangiovanni-Vincentelli, *Relaxation-based Electrical Simulation*, University of California, Berkeley, 1983

[Newell, Fitzpatrick 1982] M.E. Newell, D.T. Fitzpatrick, "Exploiting Structure in Integrated Circuit Design Analysis", *Proceedings of the Conference on Advanced Research in VLSI*, January 1982

[Nielson 1986] R.D. Nielson, "Algorithmically Accelerated CAD", *VLSI Systems Design*, February 1986

[Pfister 1982] G.F. Pfister, "The Yorktown Simulation Engine", *Proceedings of the 19th Design Automation Conference*, 1982.

[Seiler 1985] L.D. Seiler, *A Hardware Assisted Architecture for VLSI Design Rule Checking*, MIT Doctoral Thesis, 1985

[Smith, McDonald, Chang, Jerdonek 1984] W.D. Smith, J. McDonald, C. Chang, R. Jerdonek, "NEXT: A Hierarchical Layout Verification System for VLSI", *Proceedings of the IEEE International Conference On Computer Design: VLSI in Computers*, October 1984

[Taylor, Ousterhout 1984] G.S. Taylor, J.K. Ousterhout, "Magic's Incremental Design-Rule Checker", *Proceedings of the 21st Design Automation Conference*, June 1984

[Tarolli, Herman 1983] G.M. Tarolli, W.J. Herman, "Hierarchical Circuit Extraction with Detailed Parasitic Capacitance", *Proceedings of the 20th Design Automation Conference*, June 1983

[Terman 1983] C.J. Terman, *Simulation Tools For Digital LSI Design*, MIT Laboratory for Computer Science, TR-304, 1983

[Whitney 1981] T. Whitney, "A Hierarchical Design-Rule Checking Algorithm", *Lambda*, 1981

[Wilcox, Rombeek, Caughey 1978] P. Wilcox, H. Rombeek, D.M. Caughey, "Design Rule Verification Based On One Dimensional Scans", *Proceedings of the Design Automation Conference*, June 1978

[VMS 1985] Digital Equipment Corp., *VAX/VMS V4.0 System Services Reference Manual*, 1985

CS-TR Scanning Project
Document Control Form

Date : 4/26/96

Report # LCS-TR-363

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 130 (136-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number): 2,4,6,10,12,20,28,76,124,128

Photographs/Tonal Material (by page number):_____

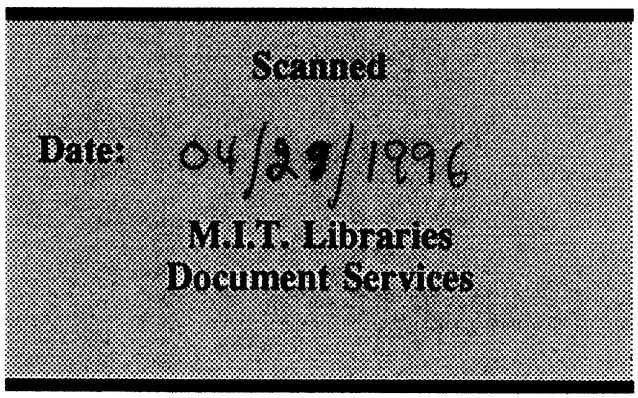
Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-130)	1-130
(131-136) SCANCONTROL, COVER, SPINE, TRGTS (3)	

Scanning Agent Signoff:
Date Received: 4/26/96 Date Scanned: 4/30/96 Date Returned: 5/2/96

Scanning Agent Signature: Richard N. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**