INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The design of a synthetic workload generator

Fortin, Michael Richard, Ph.D.
The Ohio State University, 1991
The Design of a Synthetic Workload Generator

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Michael Richard Fortin, B.S., M.S.

The Ohio State University

1991

Dissertation Committee:

Douglas S. Kerr
Mervin E. Muller
Phillip E. Krueger

Approved by

Advisor
Department of Computer and Information Science
To Marta, my best friend.
Acknowledgments

I must thank my advisor, Dr. Douglas S. Kerr, for his guidance, input and patience. A good advisor finds time for his students and Doug always found plenty of time for me. For this I will be forever grateful. I would also like to thank Dr. Mervin Muller, and Dr. Phillip Krueger for their involvement on my committee and for their suggestions that improved and strengthened my dissertation. I would also like to thank Dr. Muller for his support and guidance through the years.

I wish to express special thanks to those who have provided advice, support and friendship: Julie and Roger Barnes, Keith Boyer, Dan Butzer, Linda Condron, David Ebert, Olivier Fischer, Richard Fox, Dana Frost, John and Susan Gawkowski, Jane Grissom, Loyde Hales, Bob Henkel, Chris Holmes, Craig Joseph, Shang-Juh Kao, Anne Keuneke, Marty Marlatt, Hal and Eva McMillan, Rob Mickey, Conleth and Chris O’Connell, David Ogle, Richard and Arlene Parent, Ernie Stavely, Jeff Steinbeck, Gregor Taulbee, and Mike Weintraub.

I must also thank my many friends at Hewlett-Packard: Larry Gray, Philippe Lindheimer, Jay Silverthorn, Kevin Wilson, Michael Molloy, Carl Morgenstern and Jack McClurg. Their support and guidance will always be appreciated.

Special thanks go to my family: Roger, Janet, Tom, Steve, Dave, Jessica, Jennifer, Jeff, Melissa, Elsie and Marta.
VITA

September 17, 1963 .......................................................... Born - Bethlehem, PA

1985 ..................................................................................... B.S. Computer Science
Xavier University
Cincinnati, Ohio

1985-1987 .......................................................................... Graduate Teaching Associate
Computer Science Department
The Ohio State University
Columbus, Ohio

1987 .................................................................................... M.S. Computer Science
The Ohio State University
Columbus, Ohio

1987-Present ...................................................................... Graduate Research Assistant
The Ohio State University
Columbus, Ohio

Publications


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Database Systems: Prof. Douglas S. Kerr
- Operating Systems: Prof. Sandra A. Mamrak
- Programming Languages: Prof. Neelam Soundararajan
Table of Contents

Acknowledgments...............................................................................................iii
VITA.....................................................................................................................iv
List of Tables ........................................................................................................x
List of Figures .....................................................................................................xii

CHAPTER PAGE

I  Introduction................................................................. 1
  1.1 Performance Evaluation Categories......................................................... 2
  1.2 Experimental Performance Evaluation ....................................................... 5
    1.2.1 Workload Classes........................................................................ 7
  1.3 The Scope of the Dissertation ..................................................................... 8
  1.4 Organization of the Dissertation ................................................................. 9

II  The Problem............................................................... 10
  2.1 Other Work with Workloads..................................................................... 10
    2.1.1 Synthetic Workloads and Benchmarks........................................... 13
  2.2 The Workload Generator ......................................................................... 15
    2.2.1 The Workload Generator Approach............................................. 16

III Methodology.............................................................. 17
  3.1 Chapter Overview ..................................................................................... 18
  3.2 Phase 0: Identifying the Workload ............................................................ 19
  3.3 Phase I: The Capture Phase - Real Workload Monitoring ....................... 19
    3.3.1 Monitoring Techniques & Tools.................................................. 20
3.3.2 A Sample Sequence of Events .......................................................22
3.3.3 Capturing File I/O Data .................................................................23
3.3.4 Capturing User-Mode and System-Mode CPU Data ....................23
3.3.5 Capturing Memory Reference Data ...............................................25
   3.3.5.1 Reference Set Size: Problem Example ............................28
   3.3.5.2 Reference Set Size: Problem Solutions ...........................31
3.3.6 Phase 1 Conclusions .....................................................................34
3.4 Phase 2: Duplicate Generation and Data Analysis .....................................35
   3.4.1 The Structure of the Profile Files ...................................................37
   3.4.2 Merging Intrusive Profiles into Non-Intrusive Profiles ...............38
      3.4.2.1 The Merging Algorithm ...................................................41
   3.4.3 Task Synchronization ....................................................................42
   3.4.4 Executable Generation .................................................................44
3.5 Phase 3: Duplicate Execution - The Playback Phase ..................................45
   3.5.1 Traversing the Profile Data Structure ...........................................45
   3.5.2 The Duplicate Execution Cycle .....................................................46
      3.5.2.1 Satisfying CPU Goals ......................................................47
   3.5.3 Satisfying Resident Set Requirements: Text and Data ..........49
   3.5.4 Satisfying File I/O Requirements ..................................................50
   3.5.5 Problems with a Time-Based Approach .......................................51
3.6 Chapter Conclusions .....................................................................................52

IV A Tested Workload ......................................................................................55
   4.1 The Tested Workload...........................................................................56
      4.1.1 Evaluating Separate Runs of the Same Invariant Process ..........57
      4.1.2 Evaluating the Generated Duplicates During Playback ..........60
      4.1.3 Analysis and Meaning of Cumulative Resource Distributions...60
   4.2 CPU Activity Comparisons ..................................................................66
   4.3 File I/O Comparisons .........................................................................68
   4.4 Memory Page Reference Comparisons ...............................................69
      4.4.1 Reference Set Precision ............................................................69
      4.4.2 Reference Set Comparisons .......................................................73
         4.4.2.1 Main Memory Size Changes and Page Faults ................81
   4.5 Multiple Client Workload Emulations ..................................................84
      4.5.1 Evaluating Multiple Client Workload Emulations ...................85
      4.5.2 File Server CPU Utilization .......................................................86
6.3.3 Screen I/O Monitoring ................................................................. 147
6.4 Final Statement ................................................................................ 147

APPENDICES

A. Hardware & Software Configurations .......... 148
B. Collectible Data Items.............................. 150

Bibliography.............................................................................................. 152
List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Input and Output Operations Performed</td>
</tr>
<tr>
<td>2</td>
<td>Average Percent Differences for Real Processes</td>
</tr>
<tr>
<td>3</td>
<td>Average Percent Differences for Duplicate Processes</td>
</tr>
<tr>
<td>4</td>
<td>Overhead Page Statistics</td>
</tr>
<tr>
<td>5</td>
<td>Referenced Set Sizes for Real and Duplicate Processes</td>
</tr>
<tr>
<td>6</td>
<td>Real Workload Page Fault Statistics (Pages)</td>
</tr>
<tr>
<td>7</td>
<td>Duplicate Workload Page Fault Statistics (Pages)</td>
</tr>
<tr>
<td>8</td>
<td>Real Workload Interval of Activity Durations (Seconds)</td>
</tr>
<tr>
<td>9</td>
<td>Duplicate Workload Interval of Activity Durations (Seconds)</td>
</tr>
<tr>
<td>10</td>
<td>Real Workload Server CPU Busy Percentages (5 Second Samples)</td>
</tr>
<tr>
<td>11</td>
<td>Duplicate Workload Server CPU Busy Percentages (5 Second Samples)</td>
</tr>
<tr>
<td>12</td>
<td>Real Workload Disk Reading Statistics on 10 MB clients (KiloBytes)</td>
</tr>
<tr>
<td>13</td>
<td>Duplicate Workload Disk Reading Statistics on 10 MB clients (KiloBytes)</td>
</tr>
<tr>
<td>14</td>
<td>Real Workload Disk Writing Statistics on 10 MB clients (KiloBytes)</td>
</tr>
<tr>
<td>15</td>
<td>Duplicate Workload Disk Writing Statistics on 10 MB clients (KiloBytes)</td>
</tr>
<tr>
<td>16</td>
<td>Real Workload Response Times on 12 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>17</td>
<td>Duplicate Workload Response Times on 12 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>18</td>
<td>Real Workload Response Times on 10 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>19</td>
<td>Duplicate Workload Response Times on 10 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>20</td>
<td>Real Workload Response Times on 8 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>21</td>
<td>Duplicate Workload Response Times on 8 MB Workstations (Seconds)</td>
</tr>
<tr>
<td>22</td>
<td>Integer and Floating Point User-Mode CPU Times: Relative to CPU A</td>
</tr>
<tr>
<td>23</td>
<td>Input and Output Operations Performed</td>
</tr>
<tr>
<td>24</td>
<td>Tick Differences for Reading and Writing Operations</td>
</tr>
</tbody>
</table>
25 Workload Paging Statistics (Pages).................................................................123
26 Real Workload Server CPU Busy Percentages on 12 MB clients...............125
27 Duplicate Workload Server CPU Busy Percentages on 12 MB clients........125
28 Real Workload Disk Reading Statistics on 12 MB clients (KiloBytes).........126
29 Duplicate Workload Disk Reading Statistics on 12 MB clients (KiloBytes)....126
30 Real Workload Disk Writing Statistics on 12 MB clients (KiloBytes)........127
31 Duplicate Workload Disk Writing Statistics on 12 MB clients (KiloBytes)....127
32 Real Workload Response Times on 14 MB Workstations (Seconds)..............130
33 Duplicate Workload Response Times on 14 MB Workstations (Seconds).......130
34 Real Workload Response Times on 12 MB Workstations (Seconds)..............131
35 Duplicate Workload Response Times on 12 MB Workstations (Seconds).......131
36 Real Workload Response Times on 10 MB Workstations (Seconds)..............132
37 Duplicate Workload Response Times on 10 MB Workstations (Seconds).......132
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Performance Evaluation Approaches .................................................. 4</td>
</tr>
<tr>
<td>2</td>
<td>Phase 1 - Capture Data on Real Workload ........................................... 19</td>
</tr>
<tr>
<td>3</td>
<td>How monitoring data is gathered ....................................................... 21</td>
</tr>
<tr>
<td>4</td>
<td>Available Memory affects on User-Mode and System-Mode CPU activity .... 24</td>
</tr>
<tr>
<td>5</td>
<td>Desired relationship between a Real and Duplicate Memory Page ............ 25</td>
</tr>
<tr>
<td>6</td>
<td>State Diagram Depicting Page Aging .................................................. 27</td>
</tr>
<tr>
<td>7</td>
<td>Actual Reference Set Size Distribution ............................................ 29</td>
</tr>
<tr>
<td>8</td>
<td>Reported Reference Set Size Distribution .......................................... 30</td>
</tr>
<tr>
<td>9</td>
<td>Phase 1 - Capture Data in Two Passes ............................................... 34</td>
</tr>
<tr>
<td>10</td>
<td>From Two Data Files to Individual Profile Files .................................. 36</td>
</tr>
<tr>
<td>11</td>
<td>Subset of an Intrusive Profile File's Contents .................................... 37</td>
</tr>
<tr>
<td>12</td>
<td>Merging Algorithm ............................................................................. 41</td>
</tr>
<tr>
<td>13</td>
<td>Task Synchronizer Process .................................................................. 43</td>
</tr>
<tr>
<td>14</td>
<td>Example Task Ordering for Three Processes ........................................ 44</td>
</tr>
<tr>
<td>15</td>
<td>Create Executables using Profile Files ............................................... 45</td>
</tr>
<tr>
<td>16</td>
<td>After Interval Timer Decision Algorithm ............................................ 48</td>
</tr>
<tr>
<td>17</td>
<td>Sampling Intervals may not be in Alignment ........................................ 57</td>
</tr>
<tr>
<td>18</td>
<td>First-Run and Second-Run C2's CPU Activities (Depicted as Spikes) ...... 58</td>
</tr>
<tr>
<td>19</td>
<td>First-Run and Second-Run: Cumulative C2 CPU consumption .................. 59</td>
</tr>
<tr>
<td>20</td>
<td>Real and Duplicate Cumulative CPU consumptions for C2 ..................... 60</td>
</tr>
<tr>
<td>21</td>
<td>User-Mode CPU Time Distributions ..................................................... 63</td>
</tr>
<tr>
<td>22</td>
<td>Elapsed Time Distributions .................................................................. 63</td>
</tr>
<tr>
<td>23</td>
<td>Area of the Real Process’ Elapsed Time Distribution ............................ 64</td>
</tr>
<tr>
<td>24</td>
<td>Real and Duplicate Cumulative CPU consumptions for GNU Emacs .......... 67</td>
</tr>
</tbody>
</table>
Real and Duplicate Cumulative CPU consumptions for C1 ................................. 67
An Interval of Work for an Invariant Process .......................................................... 70
Percent Difference Algorithm for Reported Page Reference Sets ............................ 71
Real and Duplicate Text Segment Comparisons ....................................................... 75
Real and Duplicate Data Segment Comparisons ..................................................... 76
Reference Set Size Comparison curve ................................................................. 79
File Server CPU Utilization: 10 MB Clients ......................................................... 89
Total Reading plus Writing Disk I/O: 10 MB Clients (Kilobytes) ............................. 89
Compile Response Times: 10 MB Clients .............................................................. 93
Compile Response Times: 8 MB Clients ............................................................... 93
A Duplicate’s progression rate can differ for two reasons ..................................... 100
CPU Consumption: Real C1 vs. Duplicate C1s (CPU C) ....................................... 106
CPU Consumption: Real C2 vs. Duplicate C2s (CPU C) ....................................... 107
CPU Consumption: Real GNU Emacs vs. Duplicates (CPU C) ............................. 107
CPU Consumption: Real C1 vs. Duplicate C1s (CPU A) ....................................... 111
CPU Consumption: Real C2 vs. Duplicate C2s (CPU A) ....................................... 112
CPU Consumption: Real Emacs vs. Duplicate Emacs (CPU A) ............................ 112
CPU Consumption for Real and Duplicate GEO Processes .................................. 118
CPU Consumption for Real and Duplicate ENV Processes .................................. 119
CPU Consumption for Real and Duplicate X Processes ..................................... 119
Total Reading plus Writing Disk I/O: 12 MB Active Clients .................................. 128
Compile Response Times: 14 MB Clients ............................................................ 130
Compile Response Times: 12 MB Clients ............................................................ 131
Compile Response Times: 10 MB Clients ............................................................ 132
CPU Consumption for Real, Integer and Float ENV Processes (CPU A) ............. 136
CPU Consumption for Real, Integer, and Float GEO Processes (CPU A) ............ 137
CPU Consumption for Real and Duplicate X Processes (CPU A) ....................... 137
Chapter I

Introduction

The performance of a computer, or distributed system of computers, is a function of many components and their relationships, including the hardware, the operating system, and the application programs to name a few. No single evaluative approach promises to address all issues, and no single metric holds universal value. Furthermore, the performance information required by system designers may not coincide with the performance data needs of computer buyers, capacity planning managers, application users, or others.

The evaluation of a computer system can take on many forms depending on the architecture of the system and the evaluator's perspective. A computer system can be viewed as a collection of hardware and software resources made available to the end users. In such a case, both the availability of the resources and the efficiency of resource usage represent significant performance metrics.

An alternative viewpoint defines a computer system as a collection of tools capable of responding to user requests. From this perspective, tool response times represent important performance metrics to the user community.
Numerous performance tools for evaluating computers have existed for many years. They have continued to grow in popularity as the computers, themselves, have. Measurement tools are most notably used for comparing systems for purchasing decisions. Just as it is unwise for an automobile customer to purchase without checking a car's miles per gallon rating or horsepower, it is equally ill advised for a computer purchaser to buy without having some rudimentary knowledge of a machine's abilities and limitations.

Performance tools for computer systems are used for more than just purchasing aids. They can be indispensable aids for system tuning, for configuration planning, and for pinpointing bottlenecks as well. A computer's performance depends on the hardware, the operating system, the size and nature of the work to be done, and the amount of human effort required. Considering all these distinct and diverse considerations, measuring the performance of a computer system typically requires more than just one approach.

Although singular approaches may be useful in characterizing a few of the performance aspects, they are inevitably incapable of addressing them all. For instance, even if the exact number of floating point operations per second could be ascertained from some benchmark, that benchmark would still say nothing about the machine's networking abilities, I/O performance, or ease of use.

In the remainder of this chapter we will briefly introduce three major performance evaluation methodologies and highlight the advantages and drawbacks associated with each.

1.1 Performance Evaluation Categories

Historically, there are three distinct performance evaluation categories: Experimental Performance Evaluation, Simulation Modeling, and Analytical Modeling. Of the three, the experimental methodologies are the ones that provide the most realistic performance data
and information. Unfortunately, the techniques are normally very expensive in time and resources, and sometimes impractical.

As an example, consider a cluster of diskless client workstations being serviced by one file server. For a given workload, it is possible to determine how many diskless clients can be supported by the lone file server. Application emulation programs can, at considerable expense, be constructed to emulate the activities of users at each of the diskless clients. Response time measurements, server CPU utilization, or other metrics can then be used to determine the performance of the cluster. If the application emulations are accurate, a reasonable upper limit for the number of supportable clients is obtainable. On the other hand, if the application emulations have flaws, the results will also be flawed.

The construction of accurate application emulators demands a large quantity of time for analysis, development and testing. Trying to determine average response times for certain operations of a distributed application often requires a high degree of understanding of the applications' inner workings. These can only be obtained through painstaking analysis and use of the application itself. Furthermore, the application emulation routines must be constructed and tested. Application emulation requires a number of dedicated systems on which the distributed application, and the application emulations, can be tested. Also, the experimenter must create a reproducible atmosphere if the tests are to be valuable.

In addition to being costly in time and resources, experimentation typically has limited capacity planning potential. From the above example, if the diskless clients' resource requirements were expected to change in the future, then the previous application emulations become obsolete. Changes must be made to account for the new resource demands, and a new series of testing must be performed.
Additional testing for every potential change is not possible because of the expenses involved. In general, experimentation is better suited for measuring current conditions and for comparing existing alternatives.

For capacity planning questions, Analytical modeling can be a better choice. Once developed, parameters can be adjusted to account for many of the future circumstances that might arise. Unfortunately, there are some severe accuracy questions that arise due to the reliance upon numerous basic assumptions about system behavior. If the assumptions are valid then the predictions of the models typically are valid as well. In general, analytical models are only as good as those simplifying assumptions.

Somewhere between the analytical and experimental methodologies lies the simulation modeling approach. Simulation models attempt to reduce the costs and execution times of running experiments by recreating a system's operations logically, and by introducing some simplifying assumptions. The simplifying assumptions are normally

![Figure 1](image-url)

**Figure 1  Performance Evaluation Approaches**
not as broad in scope as those used in analytical models, but the validity of simulation models depends on those assumptions, as does the validity of the analytical models. Simulations provide some of the capacity planning capabilities of the analytical approach and some of the accuracy commonly associated with experimentation.

The relationships that exist among the three performance evaluation approaches are diagramed in Figure 1. The high accuracy of the experimentation approach is the approach's most redeeming quality. Conversely, the high costs in development and use are the major drawbacks of experimentation. In this experimental performance evaluation thesis, we hope to take advantage of the high accuracy aspects while reducing some of the costs of development and use.

1.2 Experimental Performance Evaluation

In this section we briefly describe the relationship between benchmarks and workloads within the experimental performance evaluation classes. One frequent misconception is that benchmarks represent the entire field of computer performance evaluation. In fact, benchmarking merely falls within the experimental performance evaluation category because it involves a high degree of hands-on experimentation.

Benchmarks are used to take practical measures of the performance of a single machine, or for comparing the performance of several computer systems. This is readily accomplished by applying one or more of the commonly known benchmarks, or a site dependent benchmark, to the computer systems and then comparing the results. Benchmarks can be used to determine just about every type of measure from simple command response times to ratios such as Transactions Per Second and the ubiquitous MIPS (Millions of Instructions Per Second).
Today’s benchmarks are conventionally identified as being “user-level” or “operating system-level” in nature, with some overlap. Operating system-level benchmarks are directed towards the inner dimensions of a computer system and report on metrics of throughput and efficiency. A benchmark designed to determine the speed with which new processes are capable of being spawned, clearly fits within the realm of operating system-level benchmarks.

User-level benchmarks, as the name implies, are focused more upon the activities of the users. Response time measures for single commands, or for groups of commands, are typical metrics that can be associated with user-level benchmarks. Because this type operates at a level above the proprietary operating system, they have proven to be useful in comparing architectures with different underlying operating systems. User-level benchmarks are the best means for comparing the performance of applications on differing operating system, or hardware platforms.

For any user-level benchmark to determine system performance accurately, it is first necessary to adequately characterize the jobs, or workload, running on the system. The benchmark must represent the actual user’s demands and work, or risk being considered meaningless. Even with benchmarking results to back them up, some claims are expressed in such a way that they are without meaning. For example, the claim “this file server can support up to X diskless clients,” is meaningless in and of itself. To have meaning it must be accompanied by a qualifier expressing the type of work being performed on the clients. The value of X is entirely dependent upon the number and types of file service requests made by the X clients. We would all be better off if individuals adjusted their claims to “this file server can support up to X diskless clients with this type of client workload, and Y diskless clients with this other type of client workload.”
Many of the older database benchmarks were designed just for this purpose. Common database transactions were grouped together as if they were actual user-issued commands, and then performed and measured on numerous competing database platforms. Performance results are consequently given with respect to a particular workload. Vendor claims could easily be understood in light of the accompanying workload descriptions.

1.2.1 Workload Classes

User-level benchmark workloads are typically divided into three classes: Natural, Synthetic and Hybrid. Workloads within the natural class are comprised of precisely the same tasks that the real user issued. The intent is for them to be indistinguishable from the real users in terms of the work done, the ordering of the issued tasks, and the rate with which the tasks are executed. As a result, natural workloads enjoy a high degree of accuracy. Unfortunately, the accuracy comes with a significant overhead price tag.

Synthetic workloads consist of artificially selected jobs. Synthetic workloads are constructed to represent the functionality of the real user's activities with accuracy, but they do not issue the exact same jobs as the real users. The artificial selection of jobs can involve a vast amount of analysis and observation of the real users and their job requests, or it can be based upon a series of guesses about user activity. Considering some are clearly more accurate than others. For example, the Whetstone[18] and Dhrystone[56] benchmarks were arrived at by an initial investment of time and energy and were, in their day, considered to be both representative and accurate. On today's large-cache machines, they no longer are considered as being very accurate by most people.

The chief flaw surrounding synthetic workloads is their scope; they tend only to be accurate for a narrow set of tasks. Nevertheless, synthetic workloads are often chosen over natural workloads because they offer a higher degree of control over the experimentation.
Synthetic workloads are also easier to operate than natural workloads. Synthetics require less time and effort when performing tests.

Hybrid workloads, as implied by the name, are an amalgamation of the natural and synthetic types. A good hybrid workload should achieve much of the accuracy associated with natural workloads as well as incorporate the control aspects of the synthetic workloads. They often offer the best opportunity for emulating the workload on a given system without expending an enormous quantity of time and resources. Application emulators, software or hardware routines that emulate users at terminals, can be regarded as being members of the hybrid class.

We have provided an overview of natural, synthetic and hybrid workloads in this section. Additional issues pertaining to the three are discussed in the next chapter.

1.3 The Scope of the Dissertation

One of the more difficult questions confronting evaluators who rely upon synthetic workloads concerns the accuracy of the workloads. Simplistic workloads can be created with little effort or expense but cannot be relied upon to perform their functions well. Time and effort generates better suited workloads that more accurately reflect the true working environments for which they were designed. This dissertation research details a new, automated methodology for creating accurate synthetic workloads.

Our thesis is that many synthetic workloads can be created with ease; that these workloads can have direct relationships to real workloads; and that the accuracy of the created workloads can be expressed by making comparisons to the real workloads. It is our contention that an automated means of generating synthetic workloads can considerably reduce the time and effort commonly associated with experimental performance evaluation in general, and with workload development in particular. Furthermore, with a large and
well-varied array of synthetic workloads at one's disposal, a more thorough testing of a computer system's memory, CPU, I/O and more, can be accomplished.

The methodology involves an extensive monitoring campaign, a workload creation phase, and a playback cycle. For this project, numerous specifically designed monitoring tools were required, as were some typically available tools. Their design and use are detailed within the dissertation.

The results from testing two workloads in both single machine and full cluster environments are provided as a means for gauging the accuracy and reliability of the Workload Generator methodology. One of the workloads, the tested workload was initially used to help tune, test and debug the Workload Generator. Since the tested workload was used to tune the Workload Generator, it cannot also be used to predict how well the Workload Generator might work on a new workload. Testing with a second workload, the experiment workload, is therefore warranted.

1.4 Organization of the Dissertation

Chapter II gives an overview of the related research on workloads. A detailed description of the Workload Generator methodology and design principles are provided within Chapter III. The results from testing the tested and experiment workloads are discussed in Chapters IV and V respectively. The tested workload was used in the design and development stages. The experiment workload provides an additional verification of our approach. Chapter VI contains our conclusions and recommendations for future work.
Chapter II
The Problem

In this chapter we detail some prior work with workloads, identify problems, and then specify the thesis topic.

2.1 Other Work with Workloads

Virtually all attempts to compare two or more systems, or versions of the same system, can be subjected to harsh criticisms regarding bias. That is the nature of the business. Different systems are just that, and when comparing them, their strong and weak points will undoubtedly be put to the test. Critics of a particular performance evaluation methodology typically complain that one of the systems, possibly their own, has been unfairly treated. They argue that the methodology stresses some aspects too much and other aspects too little. Furthermore, if the methodology were adjusted then they claim that the results would depict a different reality.

One of the more troublesome aspects of computer performance evaluation is determining just what the methodology ought to stress. Arguably, the best answer to this is, "the methodology should stress what the true working environment would stress." The question is, "what is the true working environment"? How can one approximate it? In
general terms, working environments are assumed to be comprised of those commands which are most frequently issued. For example, in [12] the authors monitored what actual users did during one working day. From the data gathered they identified the most commonly executed commands and used those in the construction of their workloads. Because of the large quantity of different commands, they found it necessary to group commands of similar characteristics together. Command types were classified by their component resource demands; namely their CPU, memory, and I/O demands.

Such a methodology is intriguing and has merit. The grouping of commands by their resource demands, and the subsequent inclusion of the proper number of commands from the available types in the workload, reveals what workload characterization is all about. It is not so much what the commands are, but rather, what the resource demands are, that make up a suitable and reliable workload. Simply put, it is undeniably easy to lump several commands together and call it a workload. It is, however, a much more difficult task to order and temporally space\textsuperscript{1} those commands so they make resource demands as a real user would.

Workload types used in benchmarking can be assigned to various classes. Hinnant\cite{35}, defines the constraints of three such classes; Natural, Synthetic, and Hybrid workloads. Hinnant identifies natural workloads as those which "consist of the exact tasks that users perform, usually in the same order and proportions that the users perform them." The peculiar nature of these tasks provides the natural workloads with their primary advantage over the other two classes. Because they are typically generated by first recording what actual users do, and then played back in nearly identical form, this class of workloads

\textsuperscript{1} Temporally space refers to the think times used to space out the execution of sequential commands.
provides for a high degree of reliability and accuracy. Unfortunately, that accuracy is tempered by the difficulties associated with recording appropriate representative workloads, and further compounded by the issue of replication and control of the benchmark. Experimentation with natural workloads normally requires a vast amount of time and a dedicated set of machines to be used for the benchmarking tests. Many performance evaluators cannot make the necessary sacrifices to make natural workloads a practical alternative for them.

A second class of workloads, synthetic workloads, “consist of tasks that are artificially constructed and that represent the functionality of natural workloads in a concise manner[35].” Hinnant is quick to point out that the artificial tasks that make up synthetic workloads are its main weaknesses. He contends that they can be determined by an enormous quantity of work and analysis, or by guess-work.

Two of the more notable synthetic workloads include the Whetstone[18], and Dhrystone[56] benchmarks. The impact of the two is so pervasive that floating-point and integer performance are often reported in Whetstones and Dhrystones respectively. One problem is that the accuracy for these types of synthetic workloads is valid only for a narrow band of tasks, floating-point and integer tasks in these cases. Synthetic workloads and benchmarks will be discussed in more detail in the next section.

Hybrid workloads are an amalgamation of synthetic and natural workloads. If effectively designed, they can incorporate the advantages and avoid many of the drawbacks associated with the other two classes. In Hinnant’s words, “they perhaps offer the best chance of simulating a system’s workload and having the simulation complete in a reasonable amount of time.” The superior control aspects associated with the synthetic
workloads should be blended with the excellent accuracy of the natural workloads in any good hybrid workload.

A fine line separates the hybrid and synthetic workload classes. Consider an application emulation workload, consisting of a number commonly executed UNIX commands bundled together within a looping construct. One individual may regard such a workload as being of the hybrid variety. After all, the workload is comprised of actual user commands encased within a synthetic construct, i.e. the loop. And yet, another individual might regard it as being synthetic in nature. Though the commands themselves are real, the bundling of the commands, including their ordering, the temporal spacing, and the frequencies with which they are issued, are prone to affect the natural aspects of the workloads. Hence, such a workload can be regarded as being a synthetic workload or a hybrid one. This dissertation will refer to these as synthetic workloads.

2.1.1 Synthetic Workloads and Benchmarks

The Whetstone[18] synthetic workload was developed in 1976. It was designed to measure scientific computing in FORTRAN, making extensive use of floating point arithmetic. Thus, the Whetstone is a user-level benchmark that also stresses one component of the system. As a performance metric, the elapsed time required to execute one million of the Whetstone’s instructions is reported.

The synthetic Dhrystone[56] benchmark was developed in 1984 by Reinhold P. Weicker. It consists of numerous complex data structures, pointers, memory and string operations. Actual system statistics were used to base the construction of the Dhrystone workload. The code is comprised of assignment statements (53%), control and flow statements (32%), and function call statements (15%). The number of Dhrystones that can
be performed in a given unit of time is used as the performance measure. This is just a CPU measure.

Linpack[19], another floating point intensive workload/benchmark, was designed by Jack Dongarra at the Argonne National Laboratory. Most of Linpack's work is done in a single subroutine, making it accurate for linear equation applications but not much else. Linpack reports MFLOPS, or Millions of Floating-Point Operations Per Second.

In addition to those single-user CPU benchmarks, there are other benchmarks designed to stress disk I/O, LAN I/O, performance through memory, and graphics performance. An example disk I/O benchmark was developed by Greenfield and Godsey[32]. Example LAN I/O benchmarks are presented by L. Press[49]. Graphics performance benchmarks are discussed by B. Brown and R. Judd[9]. The Perfect Club[7] benchmarks were designed for the scientific domain, with particular emphasis on supercomputers. There are also multi-user synthetic workloads. The Sequent[55], MUSBUS[44] and OSU User Emulations[11][26][27] are all variations on the same theme. In each, various common UNIX commands are grouped together in scripts. Within the scripts, the commands are separated by random sleep times to emulate the thinking times that real users would exhibit. By executing these scripts on a single machine, or across a network, a multi-user environment is created.

Arguably the most important current effort in benchmarking is SPEC[6], (Systems Performance Evaluation Cooperative). It began because many benchmarking experts felt that existing benchmarks were inadequate. One goal of SPEC is to collect, standardize and distribute programs to be used as benchmarks.

With the introduction of the TPC, (Transaction Processing Performance Council), benchmarks, benchmarking in the database world has recently received a resurgence in
interest. In *The Benchmark Handbook For Database and Transaction Processing Systems*, the author provides a thorough history of TPC and its relationship to the Wisconsin[8], TP1[4], and other database benchmarks.

Workload generation and use are not easy tasks. A significant amount of time is required, including time to study what actual users do, time to develop a flexible and expandable user emulation, and time to perform the actual experiments, which should be highly reproducible.

Once the costly development phase of a synthetic workload has been finished, it can then be put to a wide variety of uses. Some of the more familiar include the testing of server and client memory upgrades, the testing of disk configuration alternatives, the comparison of different system parameter settings, the examination of proposed network layouts, the analysis of response time variations, and much more. A well designed and flexible synthetic benchmark can be used to evaluate different schemes for providing the users with the functionality that they demand. For example, the OSU benchmarks have been used to study various means of delivering full screen editing to the client workstations.

2.2 The Workload Generator

Two of the more significant drawbacks associated with synthetic and hybrid workloads concern the scope and representativeness of the workloads, and the amount of time and effort required in the development stages. Much time and expense can be exerted in the creating of a workload that may be representative of one environment but may not have much scope beyond that one environment. To increase the number of environments that these workloads might accurately represent, additional time is required.

Our hypothesis is that it is feasible to monitor live workloads, create duplicate workloads, and play back those duplicates, such that the resulting duplicate behavior is
representative of the real workload's behavior. Furthermore, it is our assertion that workloads can be generated in this manner with greater ease than if they were built by hand.

2.2.1 The Workload Generator Approach

An extensive monitoring phase of a real workload in use represents the foundation of our approach. As will be shown, the accuracy of the monitored data is paramount. Flaws in the monitoring phase permeate all other phases and reduce the Workload Generator's ability to create a mirror image of a real workload.

The generator requires a great deal of data from the monitoring phase. To satisfy those demands, we have had to devise methods for overcoming intrusiveness issues. This is particularly the case when gathering data concerning memory page referencing patterns.

A generation phase follows the monitoring. This second phase is responsible for analyzing data and for generating program code. The generated code must be capable of operating as the real process that it is was designed to mimic. Text and data segment size demands, as well as CPU and I/O utilization requirements must all be within the capabilities of the generated code. Code is generated for each real process to be emulated.

After the generation of the workload code is completed, the playing back of the duplicate processes remains to be done. The playback phase includes validation through monitoring, so comparisons can be made between the real and duplicate workloads.

Chapter III addresses the key issues associated with each of the four phases and provides insight into the strengths and weaknesses of our approach.
Chapter III
Methodology

Evaluating the performance of workstations, or clusters of workstations, is often a laborious and difficult process. It is essential that the tests be repeatable and that the results be reproducible or the test results may be viewed with skepticism. Repeatability and reproducibility can be considered easy tasks if one is just concerned with the performance of a single machine's CPU, floating point coprocessor, memory accessing rate, or the like. The tasks are more difficult when trying to judge a workstation's performance on applications or other workloads which are comprised of several processes that compete for the system's resources.

A means of better understanding workloads and how they stress workstations and clusters of workstations, coupled with an automated method of generating and altering workloads will become a valuable tool for performance evaluation in the future. The Workload Generator was designed to be such a tool.

The underlying purpose of the Workload Generator concerns the accurate Capturing and Playing Back of a workload's behavioral characteristics. The capturing is performed by several monitoring tools designed to gather per process resource usage data, including
memory referencing patterns, CPU utilizations, and I/O statistics. We collect data reflecting the resource demands of processes because it is exactly at the resource level that the playback phase is designed to operate. Our claim is that two processes which make use of system resources in an identical fashion, are processes which can be used interchangeably for performance studies. Furthermore, the processes can still be useful even if the system resources are not used in an identical fashion.

We claim that it is possible to gather sufficient data about a **Real Process** so a **Duplicate Process** can be created and then used as a substitute for the **Real Process**. Throughout the remainder of the thesis, the term **Real** will be used to denote a real process, or set of processes, that was monitored during the capture phase. The term **Duplicate** will be used to refer to a process which was generated to perform as a **Real** process.

### 3.1 Chapter Overview

This chapter discusses the architecture of the Workload Generator and its four phases of operation. The chapter begins by detailing the initial phase, Phase Zero. Phase Zero involves the identification of the processes which comprise the workload. The next sections are devoted to the specialized monitoring tools used to capture data about the processes and the workload as a whole. The capturing of the data from the real processes is Phase 1. Phase 2, which includes data analysis and process generation, follows. The final sections of this chapter are devoted to the final phase, Phase 3, which includes executing and evaluating the duplicate processes.

In covering the four phases, many problems and solutions, including from monitoring deficiencies, will be presented and discussed.
3.2 Phase 0: Identifying the Workload

At any given moment on a multitasking system, with just one user, there might exist two dozen or more processes. Many are system processes, or daemons, which wake up periodically to perform necessary system tasks. Other processes, and there may be many, are owned by the lone user on the system. Of the user’s processes, we must identify those which make considerable resource demands and those which do not.

For instance, consider a user running a CAD/CAM application comprised of four processes but only three actually consume CPU time, make I/O demands or occupy memory. The fourth might be the parent of the other three processes. After spawning the three child processes the parent just waits for them to complete. It does little more than occupy a process table entry. The fourth process is not of much interest. Time and effort should not be spent modeling that process.

3.3 Phase 1: The Capture Phase - Real Workload Monitoring

What we gather with our tools is as important as how we gather it. For our duplicate workloads to emulate real workloads accurately, data concerning CPU usage, memory

Phase 1: Capture

![Diagram](image)

Figure 2 Phase 1 - Capture Data on Real Workload
accessing patterns, and I/O demands must be obtained for all processes which we desire to emulate. In essence, the captured data will be used to represent the *Real processes*, and the generation of *Duplicate processes*, as well as the execution of those duplicates, will be fueled by the data that we collect. The capture phase, as represented in Figure 2, consists of monitoring a workload and logging the data.

Accurate and efficient monitoring is critical. Flawed data collected during this phase would permeate all other phases, resulting in inaccurate duplicate workloads. The later sections contain a lengthy and detailed description of our monitoring techniques and tools.

For our experiments, we made use of Hewlett-Packard 9000 Series 300 and 400 Unix workstations. Though several configurations were used, the majority of our experiments involved 16 MB diskless workstations with 50 MHz Motorola 68030 CPUs. The lone file server in all tests, an HP 9000/375, was equipped with three 660 MB SCSI disk drives and 32 MBs of main memory. Detailed hardware and software configurations are given in Appendix A.

3.3.1 Monitoring Techniques & Tools

Phase 1 is responsible for starting the monitoring software which gathers data on a real workload. To perform the monitoring, the Workload Generator makes use of two of Hewlett-Packard’s monitoring programs, the *Collector* and the *Measurement Interface daemon (MIdaemon)*, see Figure 3. The MIdaemon is responsible for maintaining and manipulating the Measurement Interface (MI). The MI can be thought of as a one way window looking into the computer and is implemented as a shared memory segment. The MI defines many counters and other types of performance information relating to process sizes, available system memory, and timing data. It is the MIdaemon’s job to convert trace information generated by the kernel into those counters and time values. No data is logged,
i.e. written to disk, by the MIdaemon. It continually overwrites existing counter values as more recent information becomes available, via the kernel event traces.

An additional software tool is required to log and further manipulate what has been stored in the MI shared memory segment. The Collector is used for this purpose. The following schematic portrays the relationships that exist between the HP-UX kernel, the MIdaemon, the MI, the Collector and the log files.

By design, the Collector is a very "streamlined" monitoring tool. Users must first use a front end to the Collector which prompts for pertinent information, allowing the user to select what rate to sample, what data to gather, and much more. By utilizing a front end to the Collector, only sections of interest to the user get compiled into the collector code. Unneeded code is removed during compilation, thereby creating individual collectors which are both highly specialized and efficient.

The Collector can report more than 400 separate data items covering nearly every detail regarding the system's performance. Specifics concerning the global state of the

![Diagram](image.png)

**Figure 3** How monitoring data is gathered.
machine, the CPU, the disk(s), the network, the memory, and per process information are among the obtainable data items. Appendix B provides a more detailed assessment of the collectible data items.

Intrusiveness could be a problem with the Collector. Not only is it possible to collect different types of information at distinct sampling intervals, but it is also feasible to collect information only after predefined conditions have been met. With all of that available functionality, eliminating undesired aspects during compilation is the best way to create as non-obtrusive a monitoring tool as possible.

In addition to being as small in size as required, the Collector also logs data in as minimal a space as it can. Reducing the I/O burden helps reduce the intrusiveness of a monitor tool. The format for its log files, termed Parameterized Data Files (PDF), is entirely dependent upon what is being collected. This of course alleviates the Collector from having to write out more bytes than necessary, thereby reducing its burdens upon the system even further.

3.3.2 A Sample Sequence of Events

To understand how measurement works let us consider a simple example. A process, \( P \), makes a file open request. At first, the file open system call is made by the process. This in turns sparks the kernel to act on behalf of the process. One of the kernel's actions is to create a trace event indicating that process \( P \) has made a file open request. The trace event is later noticed by the MIdaemon process, which must perform a kernel call to notice the trace event. Next, the MIdaemon increments the necessary counters located within the MI shared memory segment. The updated file open counter, as well as all other MI information, can henceforth be extracted from the MI. The Collector performs that task and eventually records the file open event in the Parameterized Data File.
3.3.3 Capturing File I/O Data

File reading and writing data is the simplest and most straightforward data to understand. With the MIDaemon and Collector tools, we collect data on a per process basis pertaining to file reads, file writes, file opens and file closes. Most important of all is how many bytes were read or written, and when they were read or written. We record this activity to duplicate it later.

3.3.4 Capturing User-Mode and System-Mode CPU Data

Two CPU utilization statistics are most relevant for our purposes, User-Mode and System-Mode CPU activities. The User-Mode variety of CPU activity refers to the time in which a process’ instructions are actually consuming CPU clock cycles. Incrementing variables, computing values, making procedure calls, and evaluating conditional statements are all examples of activities which require User-Mode CPU processing.

System-Mode CPU activity refers to the time in which the kernel consumes CPU cycles on behalf of a process. All system calls, such as file opens, file reads, and get time of day requests to name a few, invoke kernel routines which perform activities for the calling process. Explicit system calls are only partially responsible for kernel interventions, even simple memory references may be enough to spark the kernel into action. Memory references to non-resident pages force a demand paging request upon the kernel. The system activity associated with requesting and bringing in the desired page, or pages, is performed by the kernel and later charged to the process as System-Mode activity.

The following scenario further portrays the User- and System-Mode distinctions. It also helps to identify a critical characteristic of User-Mode CPU activity. Consider a program segment which repeatedly accesses all elements of a 4 MB data structure. If the program segment runs continuously with changes being made only to the amount of
available memory on the machine, due to the activity of other processes, then accurate monitoring of each execution reveals that the User-Mode CPU activity remains approximately constant. However, the System-Mode activity increases as the available memory decreases.

The non-linear increase in System-Mode CPU activity is due to the increased level of demand paging activity. As the system's available memory becomes increasingly scarce, more kernel activity is required for the program segment. And yet, no matter how much system interruption occurs, the exact same number of program instructions get executed in User-Mode, therefore the User-Mode CPU time remains almost constant\(^1\). Figure 4 highlights this property. As will be explained, this characteristic of User-Mode CPU activity is important when collecting data regarding memory references. We have developed a method which allows us to monitor the real workloads intrusively in one pass, and then merge the data into another, less intrusive monitoring pass. The consistency of the

![Figure 4](image.png)

**Figure 4** Available Memory affects on User-Mode and System-Mode CPU activity.

---

\(^1\) Repeated runs of a sample program, requiring more than 200 seconds of User-Mode CPU time for each run, indicate that User-Mode CPU time varies by less than one tenth of a second. Accuracy to \(1/100\)th of a second is common.
User-Mode CPU time is the key to the merging. More on this will be detailed in the later sections.

3.3.5 Capturing Memory Reference Data

For any duplicate to behave as its corresponding real process, the memory referencing patterns, down to the page level, need to be emulated. Without that degree of accuracy, paging and swapping decisions, made by the kernel and outside our direct control, are likely to be drastically wrong. It is not enough for a duplicate to address the same number of pages as its corresponding real, there must be a "near" one to one relationship between the pages of the duplicate and the pages of the real process. Figure 5 illustrates our "near" one to one relationship goal.

Establishing and representing the one to one relationship is difficult and cumbersome. The Measurement Interface and associated tools do not provide us with page reference information. The MI, and most other Unix monitoring tools, are only capable of providing the total resident pages for each process. Therefore, we have had to create another specific

---

**Figure 5** Desired relationship between a Real and Duplicate Memory Page.
monitoring tool to collect the page referencing patterns of the Real processes. This tool is known as the Reference Set Collector.

The Reference Set Collector locates, within the Unix kernel, the Page Table Entries for each of the real processes. There exists one Page Table Entry, or PTE, for every page known on the system. Each Page Table Entry is comprised of several status fields which describe the page. Only the reference, modify, and the age fields concern us at this time. All three status types are manipulated by the kernel. The modify and reference fields are one bit in length. The age field is larger. The modify, or dirty, bit is by far the most straightforward of the three. When a page’s contents are altered, the modify bit is set within that page’s PTE. Resetting of the modify bit is done only when the page has been written back to secondary storage.

The reference bit is similar in nature, but it is set whenever a page is touched. That touching might involve modification, or it can be a simple reading of the page’s contents. The reference bit has a meaning which transcends just that, however. It is used with the age field, to determine if a page is eligible for swapping. The age field is used to determine how long it has been since the page was last touched. As will be explained, how long is a relative amount and leads to difficulties when attempting to determine accurate page referencing information.

On most systems, hardware is responsible for setting the reference bit to 1 whenever a page is touched. Software can be used for this purpose if necessary, but hardware is much more efficient, of course. The reference bit remains set until a special kernel process, which we call the Page Stealer, resets it. The Page Stealer process examines active and unlocked pages in the system, making swapping decisions as it goes. Pages which do not get referenced in a particular amount of time become eligible for swapping.
As the Page Stealer examines the PTE's, it resets any set reference bits and updates the ages accordingly. If a reference bit is set, then the corresponding age field is set to zero. If a reference bit is not set and the page is in memory, then the age value is increased by one. As the Page Stealer continues cycling through the active and unlocked pages, some pages inevitably acquire ages which are greater than the ages of other pages. Since the greater aged pages have not been referenced as recently as the others, they are the ones which are more likely to be swapped out of memory should swapping be required.

As an efficiency measure, the Page Stealer process is invoked only when needed and operates for only as long as required. It is awakened when the system's available memory drops below a low-water threshold. It continues to swap out eligible pages until the free memory has risen back above a pre-defined, high-water mark. In this manner, the Page

![State Diagram Depicting Page Aging.](image)

**Figure 6** State Diagram Depicting Page Aging.
Stealer may not be required to examine and reset every set reference bit. It ceases its operation as soon as the free memory is plentiful again.

It is entirely appropriate and correct for the Page Stealer to awaken and run only when needed. However, the fact that it cycles through part of the memory, does present some considerable problems for the Workload Generator’s reference set collector. A set reference bit implies that the page was referenced sometime since the last full cycle of the Page Stealer. However, it does not imply that the page was referenced since the last activity of the Page Stealer process, whenever that may have been. The Page Stealer’s last activity may only have lasted for a moment and it may have occurred many seconds or minutes in the past. Furthermore, the Page Stealer may not have made a full cycle through the PTE’s because adequate free memory became plentiful after a few pages were paged out.

3.3.5.1 Reference Set Size: Problem Example

To reiterate, reference bits get set by hardware in a timely fashion but may not be reset for many seconds or minutes afterwards. To better understand the problem, consider the following situation.

A Process is executed on a system with 8 MBs (or 2,048 pages\(^3\)) of available memory and a low-water mark of 100 pages. The process has a maximum memory requirement of 4.0 MBs (or 1,024 pages). The process starts at time T0, and finishes at time T7. Between time T0 and T1 the process references 768 of its memory pages. Between T1 and T2, the process references only 512 pages of the prior 768 pages. All the process’ 1024 memory pages are referenced between times T2 and T3, and 768 of those get touched during the

\(^2\) This dual threshold algorithm is used so that repeated thrashing around a single mark can be avoided.

\(^3\) The system’s page size is 4096 bytes.
interval before time $T_4$. Of the 768 pages, only 256 of that total are referenced between $T_4$ and $T_5$. The process once again touches all of its pages in the interval preceding $T_6$. The process terminates before $T_7$. Figure 7 depicts the actual reference set size\textsuperscript{4} distribution of the process.

The graph shows the total referenced pages at each interval. At any given time, the system's free memory can be calculated by subtracting the total referenced pages from the Total System Memory. At times $T_3$ and $T_6$, the process references all of its 4 MBs, i.e. 1024 pages. Subtracting that maximum from the system's total memory indicates that the process never depletes the system's free memory to the point where the low-water mark of 100 pages is crossed. In fact, there are always at least 1024 free pages.

\begin{center}
\begin{tabular}{lcccccc}
\hline
Pages & 2048 & 1536 & 1024 & 768 & 512 & 256 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\textbf{Figure 7} Actual Reference Set Size Distribution.
\end{center}

\textsuperscript{4} The actual reference set size for a process is computed by summing the number of memory pages which were referenced during a small time interval.
In such a situation, with no other interfering processes, it is not possible for us to gather accurate reference set information for the process via the age and reference bits. This fact is due to the nature of the Page Stealer process itself. Since the total memory requirement of the process does not force the system's available free memory below the low-water mark, the Page Stealer is not sparked into action. And although the reference bits for the process get set by hardware as the pages are touched, they are never reset by the Page Stealer. Collection processes relying upon the reference bits therefore report unrealistically large reference set sizes for the process. Figure 8 portrays the “reported” reference set sizes for the process.

The reported reference set size distribution of a process, when compared to the actual, can be drastically inaccurate if the available memory is plentiful. In this example, the Page Stealer is never required to reset any bits, therefore, the reported reference set size increases

![Figure 8 Reported Reference Set Size Distribution](image_url)
whenever new pages are referenced but never decreases. By time $T_3$, all the process' reference bits are set and stay set for the remainder of the process' lifetime. An overestimation of the actual reference set size results.

3.3.5.2 Reference Set Size: Problem Solutions

The problem is clear, the reference bits need to be reset in a timely manner. This can be done several ways, all of which require more activity from the Page Stealer process. Of course, additional activity by the Page Stealer will increase the intrusiveness of the monitoring. It is important to note that the Page Stealer's purpose is slightly different from our own. For efficiency reasons, it operates at a slower rate than we desire.

If the reference bits are to be reset more frequently, then the Page Stealer must operate more often. In fact, it needs to operate almost continuously. In the ideal case, every resident page's page table entry should be examined by the Page Stealer at every interval. Otherwise, set bits from one interval will remain set in the next interval. It will then be impossible to determine whether the page was actually referenced in both intervals or if it was referenced only in the first interval.

With the problem well defined, it is appropriate to address potential solutions. A potential solution involves an overhaul of the kernel's Page Stealer process and the meaning of the reference bit. Modifications could be made so the Page Stealer operated more frequently and was designed to reset all set reference bits, instead of just doing so until adequate free memory was available. Tampering with the kernel in this way, although conceptually simple, might introduce unanticipated side effects. Furthermore, if we were to depend upon this means of determining accurate reference set data, then we would be forced to use our "hacked" kernel for all of our testing purposes. That could severely limit the Workload Generator's use and scope.
A second alternative introduces a process to complement the Page Stealer. This process would run with the kernel and would be responsible for resetting set reference bits periodically. This methodology alleviates the "hacked" kernel burden, but deadly side effects, including system crashes, are possible and have been observed. Both of the above alternatives involved too high a degree of kernel tampering which eventually resulted in their being designated as unsatisfactory.

As third alternative, we have found that the Page Stealer process can be pushed into performing the desired actions on a regular basis by locking large portions of the system's memory. This locking of memory reduces the amount of free memory that would otherwise be available\(^5\).

If enough is locked, the Page Stealer is continually confronted with a system that has too little free memory. Its response is to reset reference bits and swap out some pages. The problem with this approach is the swapping. With large amounts of the system's memory locked and unavailable, the system is forced to page and swap almost continuously. This places a high load upon the system and drastically increases the amount of time that a process needs to complete a given task.

A final alternative is to change the system's low-water mark\(^6\). Instead of having to lock memory out of the system so the free memory drops to reach the low-water mark, we can increase the low-water mark so it is easily reached without memory locking. For example, on a system with 8 MBs of memory, instead of locking 6 or 7 MBs of memory, we can increase the low-water mark by 6 or 7 MBs.

\(^5\) Locking memory is logically equivalent to removing memory chips from the machine.
\(^6\) We would like to thank George Jones for this idea in another application.
Once again, the Page Stealer is continually confronted with a system that has too little free memory. Its response is to reset the reference bits, and, instead of swapping or paging out pages, it merely places them on the free list. The free list can grow to a fill a space as large as the low-water mark. For example, on a system with 8 MBs of memory and a low-water mark of 6 MBs the free list can grow to be 6 MBs.

With a large free list to hold the pages in memory, the overhead of swapping and paging is removed. There is still other overhead, however. The Page Stealer process is forced to run almost continuously. In doing so, it takes up CPU time that otherwise would have been available to the other processes on the system. Also, there is considerable overhead involved in the monitoring of a process’ page table entries and reference bits. A single process can have thousands of memory pages and we may be monitoring more than one process at a time. To do that work the monitoring becomes very intrusive.

As a result of our monitoring and of the Page Stealer’s increased activity, the other processes on the system, including the monitored ones, take more wall-clock time to complete their given tasks. We will refer to a system which has this overhead as a “Stressed” environment. The heavy monitoring load and the actions of the Page Stealer present the system with a level of stress that a normal operating environment would never encounter. How then can the data from an intrusive, “Stressed” environment be useful?

A method to transfer data gathered in a “Stressed” environment to a more normal environment is needed. As mentioned in Section 3.3.4, a process, if run repeatedly, accumulates the same quantity of User-Mode CPU time during each of the repeated executions. This characteristic of User-Mode CPU time allows us to merge certain data items from one monitoring pass into the data gathered during another monitoring pass. Thus, we can monitor intrusively in one pass and monitor non-intrusively in another pass.
The detailed page referencing data from the intrusive pass can then be transferred into the data set gathered during the non-intrusive pass.

The non-intrusive monitoring provides a more accurate representation of the processes in a normal environment. Think times, sequencing information, and other "real-time" considerations are much more realistic. Merging data from one pass into the other allows us to take advantage of the benefits that each pass has to offer. A description of the merging technique and the limitations associated with two-pass monitoring can be found in Section 3.4.

3.3.6 Phase 1 Conclusions

Phase 1, the real workload monitoring phase, makes use of some special purpose monitoring tools including the MI, the MIdaemon, the Collector and the Reference Set Collector. The Collector, with the help of the MI and MIdaemon, allows us to capture I/O and CPU data on the real workload's processes efficiently. The Reference Set Collector gathers data concerning the memory page referencing patterns of the processes. The

![Phase 1: Capture Diagram](image)

**Figure 9** Phase 1 - Capture Data in Two Passes
Collector is efficient and non-intrusive, but the Reference Set Collector places a heavy burden upon the system. Because of that, we monitor the real workload’s processes in two passes.

A non-intrusive pass, with only the Collector in operation, is first used to gather data on the processes in a normal environment. A second pass, with both the Collector and Reference Set Collector in operation, is later used to acquire data on the memory page referencing patterns of the processes. During Phase 2, the data from the second pass will be merged into the data gathered during the first pass.

3.4 Phase 2: Duplicate Generation and Data Analysis

Phase 2 has several distinct components, all sharing a common purpose. Each component helps form a transitional bridge between phases one and three, (i.e. between capture and play back). The data from Phase 1 needs to be processed and analyzed, and the source code for each of the duplicate processes needs to be generated.

The data captured during Phase 1 is contained within two large files. One file exists for the Non-Intrusive pass and another for the Intrusive pass. Each of those files contains the collected data for all the processes of the real workload. Our first post processing action is to create one data file for each of the real processes. Figure 10 schematically displays how this is done.

The first step is to divide the two large data files into many smaller files, one for each process. Each of the per-process data files, known as Profile files, contains information about just that one process. At this point there exists a Non-Intrusive Profile file and an Intrusive Profile file for each of the workload’s processes. We need to merge the data from the Intrusive Profiles into their corresponding Non-Intrusive Profile files to utilize the time sequence data in the Non-Intrusive Profile files. An understanding of the merging technique
is only possible if one is aware of the format of a Profile file and what the profile files represent.

Figure 10  From Two Data Files to Individual Profile Files
3.4.1 The Structure of the Profile Files

A Profile File for a process can be thought of as a trace of that process’ activities. Each process of the real workload has two Profile Files, Intrusive and Non-Intrusive, at this stage.

A sampling interval can be of virtually any length, although a 1 second interval is used throughout this thesis. Profile files consist of a sequence of Records, one for each sampling interval. Thus, a process running for 200 seconds would result in 200 records.

Each record of the profile file consists of Data Fields. The data within each field either corresponds to the resource demands that the process made during the sampling interval, or it is a time value indicating how many Ticks have passed since the previous sample was taken. There are exactly 100 ticks per second. The data fields deal with a process’ Data and Text page references, User-Mode and System-Mode CPU activities, and file reading and writing demands. The following figure illustrates the form of an Intrusive Profile file. A Non-Intrusive Profile file has all but one of those fields, the Page Reference Bit Stream Field is absent.

<table>
<thead>
<tr>
<th>Time of Day</th>
<th>User Mode</th>
<th>System Mode</th>
<th>File Read</th>
<th>File Written</th>
<th>Page Reference Bit Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>664644169</td>
<td>13</td>
<td>6</td>
<td>123</td>
<td>0</td>
<td>010001100101100100100100</td>
</tr>
<tr>
<td>664644170</td>
<td>21</td>
<td>12</td>
<td>10</td>
<td>15</td>
<td>110100100100010001000100</td>
</tr>
<tr>
<td>664644171</td>
<td>45</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>110010010001000100010010</td>
</tr>
</tbody>
</table>

Figure 11 Subset of an Intrusive Profile File’s Contents
In essence, each profile record can be considered to be a captured “snapshot” of the process during a particular sampling interval. Furthermore, each represents a desired state that a duplicate will attempt to achieve during a comparable interval.

As an example, consider the 1st profile record from the above figure. That line indicates that the process accumulated 13 Ticks of User-Mode CPU time and 6 Ticks of System-Mode CPU time during the 1 second sample. Furthermore, the process performed 123 file read operations and wrote zero times\(^7\) during the sample. The Page Reference Bit Stream data indicates which pages were marked as being referenced and/or modified during that 1 second sample.

### 3.4.2 Merging Intrusive Profiles into Non-Intrusive Profiles

A merging is possible between data gathered from an intrusively monitored environment and data collected less intrusively. This merging can be done with a high degree of accuracy for all “Invariant” processes.

An invariant process is defined to be one that always executes the same code and always does so in the same order. Invariant processes do not exhibit random behavior of any kind, and consistently progress from beginning to end in exactly the same fashion.

As an example, consider a sorting program which is given, as input, the same data set throughout repeated executions. In every execution, the decisions made by the sorting algorithm are precisely the same. Therefore, the instructions executed remain the same as well.

---

\(^7\) In addition to the number of file read and write operations, a Profile Record also contains data fields indicating how many bytes were read or written, and how many files were opened or closed.
When presented with a different set of data, or the same data in an alternate sequence, the sort may potentially make different decisions. We do not consider sorting algorithms to be invariant across data sets.

Programs requiring user input can be invariant as well. In such cases, the user input is analogous to the sorting algorithm's data sets. If the programs, during repeated executions, are driven by the same user input then the programs are invariant by definition. As with the sorting case, if the user input varies across executions then the invariant nature has been lost.

A more formal definition of an invariant process is expressed in Equation (3.1). \( M \) and \( N \) represent two distinct executions of an Invariant process, with \( I_M \) and \( I_N \) denoting the sequence of respective instructions that they execute. The \( k \)th instruction executed in run \( M \) is denoted by \( I_{M,k} \).

\[
I_{M,k} = I_{N,k} \quad \forall k
\]  

Equation (3.1) states that the code executed, right down to the instruction level, remains the same in each of run of an invariant process. Although invariant processes consistently perform the same instructions from run to run, that of course does not imply that their wall-clock executions times remain constant. Invariant processes are affected by the other processes on the system and must compete for resources in the normal fashion. In one execution of an invariant process it can be forced to wait a total of \( X \) seconds for file I/O, user input, and paging/swapping purposes. During another execution it can be forced to wait for a total of \( Y \) seconds. If \( Y \) and \( X \) differ, then so too will the wall clock execution.
times. Fortunately, wall clock execution time differences do not prevent the merging of data.

For invariant processes, because the same instructions are executed with every run, it is possible to identify and align equivalent locations within the data sets gathered for those runs. This can be done by matching up the accumulated User-Mode CPU times. User-Mode CPU time represents the quantity of CPU time that a process uses to execute its instructions. The User-Mode CPU time required to execute any instruction $I$ is the same for every execution of $I$.

By our definition of an invariant process, each instruction $I_{M,k}$ that is executed during the $M^{th}$ run is equal to $I_{N,k}$ from the $N^{th}$ run. Let $t_k$ represent the amount of User-Mode CPU time necessary to execute instruction $I_k$. The following equations hold true by the definition of User-Mode CPU time.

$$ t_{M, k} = t_{N, k}, \forall k $$  \hspace{1cm} (3.2)

$$ \left( \sum_{k=1}^{i} t_{M, k} = \sum_{k=1}^{i} t_{N, k} \right), \forall i $$  \hspace{1cm} (3.3)

Equation (3.2) simply states that the User-Mode CPU time required to execute an instruction in one run is the same as it would take to execute that instruction in any other run. That equation, coupled with Equation (3.1), clearly implies that User-Mode CPU time accumulates in the same fashion during every run of an invariant process.
The equations above imply that User-Mode CPU time can be used as a means of aligning data gathered during distinct runs of invariant processes. That characteristic of User-Mode CPU allows us to merge data collected from one monitoring pass into data captured during another pass. Our only constraint is that the processes must be invariant processes.

3.4.2.1 The Merging Algorithm

It is necessary to add page reference data collected in the stressed environment to the page reference data collected in the normal environment. Therefore, we want to merge the page reference data from the stressed environment into the rest of the data from the normal environment. The merging is done by aligning common execution points in two separate runs of an invariant process. User-Mode CPU time is used to identify the alignment points. A pseudo-code representation of the algorithm is shown in Figure 12.

```
j = 0
for (i = 0; i < Number of Normal Profiles; i = i + 1)
    Set ALL of Normal Reference Bits [i] to ZERO
    while ( Normal User Mode Time <= Stressed User Mode Time )
        Normal Reference Bits[ i ] = Normal Reference Bits[ i ]
        OR
        Stressed Reference Bits [ j ]
        j = j + 1
    if ( j > Number of Stressed Profiles ) then STOP
end while
end for
```

Figure 12 Merging Algorithm.
The algorithm is based upon our knowledge of User-Mode CPU time and how it accumulates for a process. The term "Normal" refers to Profile records obtained during a Non-Intrusive monitoring pass. They represent the process' behavior in a "Normal" working environment. Conversely, the term "Stressed" refers to the Intrusively monitored environment.

Initially, all the page reference bits in the Normal environment's profile records should be zero because Reference bit data was not gathered during the Normal monitoring pass. After setting the Normal's reference bits to zero, we logically OR in reference bits from the Stressed profile records until we reach a point where the process in the Stressed environment has accumulated as much User-Mode CPU time as the process in the Normal environment. At that point, the Normal's profile record has a bit set for each page that was referenced by the process in the Stressed environment as it performed the same quantity of work as the process in the Normal environment. The procedure repeats until we run out of Normal or Stressed profile records.

3.4.3 Task Synchronization

After each process' pair of Profile files have been merged, a task synchronization process operates on the entire set of merged profile files. The aim is to determine the synchronization patterns which exist between the processes so those characteristics can also be emulated properly.

Many interesting workloads involve several processes which do not run in a purely sequential fashion. Parts of process A may run simultaneously with portions of process B, and portions of A may run prior to portions of process B. A Workload Generator goal is to have the same patterns when A's and B's generated duplicates execute.
By breaking the processes down into smaller chunks, called Tasks, we can determine the ordering in the profiles and enforce the ordering when the duplicates run. The Task synchronization program examines the CPU activity in the profiles for each process and locates ranges in which one process is the only one executing. Every profile record for that process, within the located range, is identified and grouped as a Task. The ordering relationship of that task to all other tasks is sequential, therefore we refer to it as a sequential task. In the example portrayed in Figure 14, \( t^1_1, t^3_1, t^2_1, \) and \( t^2_2, \) are all examples of sequential tasks. Notice that their preceding and following tasks are easily identified, no matter the process from which they may be a part.

The Task synchronizer also identifies ranges in which more than one process is active. A task for each range and for each of the involved processes is assigned. These tasks represent instances when more than one process was active in a time interval and are termed parallel tasks during the time interval. Tasks \( t^1_2 \) and \( t^3_1 \) are examples of parallel tasks.

Figure 13  Task Synchronizer Process
Tasks may be very lengthy and span many profile records, or they may span just one profile record.

At this point within the Generation phase we have a merged profile record corresponding to each process and a detailed understanding, in the form of Tasks, of the interrelationships which exist between the processes of the workload. All that remains to be done within this phase is the actual generation of the duplicate process code.

\[
\begin{align*}
P1: & \quad t_1^1 \rightarrow t_2^1 \rightarrow t_3^1 \\
P2: & \quad t_1^2 \rightarrow t_2^2 \\
P3: & \quad t_1^3 \rightarrow \\
\end{align*}
\]

Figure 14  Example Task Ordering for Three Processes.

3.4.4 Executable Generation

Executables are created in a very straightforward and easy manner. The basic core of each duplicate executable is a C program consisting of several decision making algorithms which determine how the duplicate executable is to behave throughout the course of its life. Each duplicate executable makes its decisions by examining the current profile record contents.

The only abnormal aspect to the generation of the executables concerns the Maximum Desired Text Resident Set Sizes of the duplicate processes. By scanning the complete list of profile records in a profile, the generator determines the maximum required text\textsuperscript{8} resident size for the process. An appropriate number of previously created object modules are
compiled into each executable in order for it to match the desired amount of text. This enables each generated executable to achieve its desired text goals during the playback phase.

We are not required to determine maximum data resident set sizes because data segment size requirements can be met by dynamic allocation during execution. Text requirements have to be met at compile time.

3.5 Phase 3: Duplicate Execution - The Playback Phase

The Playback phase begins with a set of executables, one for each process, and a set of Profile files as well. As previously stated, the duplicate executables emulate the real processes by making decisions based upon the profile record contents.

3.5.1 Traversing the Profile Data Structure

Before continuing any further, an explanation is needed concerning the means in which the duplicate executables make use of and traverse their profile data structures. Recall from

Figure 15 Create Executables using Profile Files

8 Text pages and Data pages are not the same on a Unix system. Text pages contain the code of a program.
Section 3.4.1 that profile files consist of *Records* composed of *Data Fields*. Each record represents an instance in time and we can index through the profile data structure to obtain state information about that time. Therefore, each increment in the index value represents a transition from one sampling interval to the next.

During the initial stages of a duplicate process' execution, it must synchronize itself with all other duplicate executables. It does this via a shared memory segment. One of the duplicate processes is passed a parameter which indicates that it should control the start up synchronization. That duplicate creates the shared memory segment and is responsible for reading the task synchronization data into the shared memory. All duplicates determine the Tasks to which each of their profile records belong. The determination is based upon the timestamp information found within each profile and the beginning and ending points of each Task.

The controlling duplicate's last responsibility as the controller is to signal\(^9\) all processes which are members of the very first Task. The signal instructs the processes to begin their operations. Those processes can then progress freely until they have processed all of their profile records which belong to that Task. The last process to complete the Task is responsible for signalling the members of the next Task. As will be explained further in the next sections, this cycle is repeated until all Tasks have been completed.

### 3.5.2 The Duplicate Execution Cycle

After the initial synchronization, each duplicate proceeds into the main controlling loop. It is within this loop that all behavioral decisions are made. Those decisions are based upon the data contained within profile records and within the Task information.

---

\(^9\) The signalling is done via the Unix *signal* mechanism.
3.5.2.1 Satisfying CPU Goals

Acquiring the proper number of CPU ticks is done by utilizing the `setitimer`, an interval timer, and the `signal` mechanisms common to Unix systems\textsuperscript{10}. The signal system call is used to trap alarms and call declared alarm handling routines. An interval timer can be made use of in three separate ways, all of which are specifiable as parameters to the duplicate executables. If a \textit{Real Timer} is set by `setitimer`, then an alarm will be delivered to the process when the specified allotment of time has elapsed. This is a type of "Wall Clock" timer.

A \textit{Virtual Timer} works differently. Virtual timers count down only when the process is actually executing in User-Mode. Alarms from virtual timers do not sound until the process has acquired the specified amount of User-Mode CPU microseconds.

\textit{Profile Timers} are the third type. They closely resemble the virtual timers but not only do they count down when the process is running in User-Mode, they also do so when the system is running on behalf of the process, i.e. in System-Mode.

If all three types, real, virtual and profile, were set at the same instant and with the same time values, then the real alarm would sound no later than first, the profile would sound no later than second, and the virtual would sound last. It is possible for all three to sound at the same instant if and only if the process runs for the entire time frame with no waiting and no system intervention.

The duplicates make use of the Virtual Timers. They extract a User-Mode CPU value contained within a profile record and then set the interval timer to that amount. Between that instant and the time when the timer alarm sounds, the executable attempts to fulfill all

\textsuperscript{10}These mechanisms are known to be available on current HP-UX and SunOS systems.
of its obligations as spelled out by the profile data on hand. The process must access data, it must address text, and it may even have to read from or write to files\textsuperscript{11}. 

\begin{verbatim}
if ( in the midst of a Task ) then
    while ( next profile record has no User-Mode CPU activity ) do
        Sleep ( Sample Rate Seconds )
    end
else if ( End of Task ) then
    if ( We are NOT a member of The Next Task ) then
        Signal Those in Next Task
        Sleep Indefinitely for a Continuation Signal
    else if ( We are a member of The Next Task ) then
        Signal Those in Next Task
        Continue Immediately
    end if
end if
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure16.png}
\caption{After Interval Timer Decision Algorithm}
\end{figure}

\textsuperscript{11}A duplicate's reliance upon interval timers as a means for determining how much work to perform does present a problem if the duplicates are to be run on another CPU platform. The clock rates on different CPUs can be different. One hundred microseconds of work on one CPU may only take 75 microseconds on another CPU. We address this problem in more detail and present a solution in the final sections of this chapter.
When an interval timer does expire, the executable suspends its actions for a period of time. The process remains idle until it is instructed to proceed once again. The quantity of idle time depends upon what the profile and task data indicate. Recall that each profile record represents a snapshot of the real process’s resource demands, and that each task is used to reflect the temporal ordering relationships that were exhibited among the real processes.

The profile and task information are both used by a duplicate process to determine whether it should continue immediately, pause for a brief span of time, or wait indefinitely for another process to signal the duplicate to proceed once again. An outline of the algorithm appears in Figure 16.

3.5.3 Satisfying Resident Set Requirements: Text and Data

Each profile record has a bit stream incorporated into it. Two bits are required to represent one data page, whereas just one bit is required for the text pages. Data page bits have to indicate 3 states, reference made, no reference made and modified reference. A text page’s bit must only represent two states, reference made, and no reference made. A text page can never be modified.

The duplicate processes have the same number of text and data pages as the real processes, plus a few additional pages for overhead. The bit stream bits correspond to data and text pages from a real process. That process’ corresponding duplicate process assigns the bit stream bits to its own data and text pages. A one to one mapping is the goal.

---

12Because a process’ data segment is immediately followed by its stack segment, we have found it reasonable to count stack segments as part of the data. Future references to a data segment’s size will always imply the inclusion of the corresponding stack segment’s size.
If the duplicate sees a modified data page bit, it modifies the corresponding page. When a reference is made, the duplicate extracts a byte from the data page. If no reference is indicated by the bits, then the duplicate skips the data page.

Text pages are handled similarly, but the procedure is slightly more complicated. Recall from section 3.4.4 that object modules are compiled into the generated duplicates so they can achieve the same maximum resident text sizes that the real processes achieved. Those object modules are called by the duplicate processes depending upon the setting of the reference bits.

3.5.4 Satisfying File I/O Requirements

Conceptually, file reading and writing is by far the simplest of the criteria to satisfy. If the Profile data says that \( X \) bytes were read in or written out, then the duplicate executable attempts to do the same. Unfortunately, it is not so simple. Situations arise when file reads and/or writes start but do not complete before the sounding of the interval timer. The sounding of the interval timer indicates that the process has received all the User-Mode CPU time that it should obtain at this moment. The duplicate process can either postpone the reading and/or writing until the next interval, complete the I/O before servicing the alarm, or try to complete it for an additional span of time before giving up and servicing the signal. The default is to postpone the alarm servicing until the I/O completes. For the sake of flexibility, a run-time parameter can be used to change the default.

Questions about the number of files to read from and/or write to must be addressed. The current implementation allows the user to specify as a parameter the number of files to be used for reading and writing. At least one file is required if any reading or writing is to be done.
3.5.5 Problems with a Time-Based Approach

The approach detailed in the preceding sections produces accurate and realistic results when the duplicate processes are run upon the same machine that the real processes were monitored. The approach detailed to this point does not, however, allow for a change in the underlying CPU architecture. Recall that the methodology relies upon timing information. The duplicate processes, up to this point, have been instructed to execute for some allotment of User-Mode CPU time, 25 milliseconds for example. Unfortunately, twenty-five milliseconds of User-Mode activity on 50 MHz 68030 processor is not equivalent to twenty-five milliseconds on a 20 MHz 68020. The number of instructions which can be executed by each processor is not equivalent during such a span of time. Slower processors require more time to complete the same quantity of work than faster processors.

The time-based difficulties are serious but there is a partial solution. In order for the duplicates to run properly in an altered environment they have to behave in much the same manner as the real processes would. If the real processes were executed on two separate CPUs, the amount of work that they would perform would remain constant. They would still execute the same instructions, address the same memory pages and make the same file I/O requests. The quantity of work done remains a constant even if the elapsed wall-clock time changes, and even if it takes a different number of CPU milliseconds to execute the code.

This property of the real processes must be incorporated into the generated duplicates. To do so, the duplicates must translate the time-based instructions found in the profile records into work-based instructions. This translation requires an initial run of the duplicates in an environment that is identical to the one from which the real profile data was gathered. The duplicates must run, at least once, on the same CPU as the real processes for
the translation to work. It is during that initial run that the time-based instructions are translated into *Quantities of Work*.

When a duplicate process sets an interval timer to go off some amount of time in the future, it also records its *location* within the executable. This is known as an *initial location*. As it runs during that allotment of time, it records *intermediate location* information of interest, such as how many times the process went through a loop. When the interval timer expires and signals the duplicate, the duplicate makes note of the *final location* where it had progressed to when the signal arrived. The amount of time that the duplicate ran has, in effect, been translated into some quantity of work. The quantity is specifiable by the initial, intermediate and final location information.

The translation is complete when this has been done for all intervals contained within the profile records. The duplicates can now be run without the interval timers. They can now use the location information and run from the initial location, through the intermediate locations, and stop when they arrive at the final location. It does not matter if it takes more or less wall clock time to accomplish. It does not matter if CPU speeds have changed. The same Quantity of Work, i.e. number of instructions, is guaranteed to be executed as if the real processes were running.

### 3.6 Chapter Conclusions

The Workload Generator approach is based upon Capture and Play Back. There are four main phases of operation. Phase 0, the identifying of the real workload’s processes, precedes the actual capture phase, i.e. Phase 1. During Phase 1, two monitoring passes are required in order to gather accurate page referencing data and observe the interrelationships that exist between the processes.
Collecting data about the resource demands of processes highlights the Capture phase. This data is then used to fuel the generating of synthetic duplicate processes and to drive the duplicates during the play back phase. Because of its widespread use, the accuracy of the gathered data is a major concern. As will be shown in Chapters IV and V, our data gathering techniques are good, but not perfect.

Phase 2, the data analysis and code generation phase, is responsible for merging the data gathered in the two passes into one Profile data file per process. The synchronization characteristics of the real processes are then determined and represented as Tasks. Also during Phase 2, the Workload Generator creates duplicate code for each of the processes. The code size for each duplicate process is as large as the code for its respective real process.

The execution of the duplicate processes is done during the Playback phase, i.e. Phase 3. The duplicates attempt to emulate the task synchronization, CPU demands, memory referencing patterns, and I/O requests of the real processes. The execution of the duplicates can be driven by time constraints and interval timers, or by quantities of work.

In Chapter IV, a real workload, the tested workload, is presented and a duplicate of that workload is created. A comparison of the real and duplicate workloads is made. We also discuss several current problems with the Workload Generator’s design, such as the problems associated with moving workloads to different CPU platforms.

The tested workload in Chapter IV was hand created to stress all aspects of the Workload Generator. This tested workload allowed us to further refine the Workload Generator’s algorithms. Because it was used to aid in the development and tuning of the Workload Generator, we do not claim that it can be used to predict how well the Workload Generator might perform on other workloads.
We provide the results of testing an additional workload, the *experiment workload*, in Chapter V. We did not use the experiment workload to help tune the Workload Generator.
Chapter IV

A Tested Workload

To tune, test, and debug the Workload Generator, we needed an initial set of workloads to monitor, analyze and emulate. These test workloads were used to help develop and further refine the Workload Generator's routines. Because we would be running many time consuming tests, we required that all our test workloads be easy to start, easy to stop, and easy to adjust. We decided to design our own initial set of test workloads to satisfy those requirements. All of our test workloads were similar in nature. Some workloads consisted of two processes, others were comprised of three processes.

The test workloads, such as the one presented in this chapter, were designed to aid in the development of the Workload Generator. Since these test workloads were used to tune the Workload Generator, they cannot also be used to predict how well the Workload Generator might work on a new workload. In the next chapter, we provide results for an experimental workload. The experimental workload was not used to tune the Workload Generator.

Emulation results from one of our tested workloads is presented in this chapter as well as a detailed description of the workload itself. We also discuss our methods of evaluation.
Unless otherwise specified, the tests in this chapter were run upon HP 9000/400 workstations with 16 MBs of main memory, and file service being provided by one HP 9000/375.

4.1 The Tested Workload

The tested workload utilizes two different C programs and a GNU Emacs editing program. The C programs were designed to work with files, index through their memory segments, and sleep for intervals of time as a means of emulating user thinking times. The GNU Emacs program, built from the Emacs-Lisp programming language, edits files in much the same fashion as do real users. Some of the editing instructions include file visiting, text insertion, text searching, and cutting and pasting sequences. Sleeping does occur between some of the editing operations to emulate user thinking times. See [26] for more details regarding the characteristics of the Emacs-Lisp program.

The two C programs, C1 and C2, and the GNU Emacs program communicate with one another, via signals, so there are sequences of parallel activity and sequences of sequential activity. Our goal was to emulate the case where a process, or processes, were forced to wait for another process, as well as the case where processes were free to execute and progress independently.

The GNU Emacs process has a large text size requirement, has a moderate data segment size, performs a considerable amount of file I/O, but makes very low User-Mode CPU demands. It stresses the system by occupying memory space, not by overloading the CPU. Process C2, on the other hand, makes considerable User-Mode CPU demands as it repeatedly performs floating point and integer calculations. C2 also has large data and text segments that it accesses frequently. Process C1's resource demands are modest. Its CPU acquisition rate is relatively low and its text and data segments are not very large.
4.1.1 Evaluating Separate Runs of the Same Invariant Process

To understand the results presented within this chapter, one must first understand the Collector's sampling intervals and how profile records can be analyzed. Consider the following example, an invariant process, IP, is run twice and monitored twice on the same system. The Collector is used to gather data about process IP at one-second intervals, for a total of N seconds. The value of N is specified as a parameter. Each one-second sample represents an individual profile record for the monitored process, IP. The N resulting profile records from the first-run are each represented by $FP_i$, $i = 1$ to N. Similarly, the N profile records from the second-run are each represented by $SP_i$, $i = 1$ to N.

Even though IP is an invariant process, i.e. runs exactly the same way during each execution, it is not possible to claim that any $FP_i$ equals the corresponding $SP_i$. Instructing the Collector to sample at one-second intervals is no guarantee that it will do so precisely. Contention for resources, clock resolution, and other considerations all work against the precision of the Collector. With a heavily loaded system, we have observed one-second sampling intervals to be off by as much as 2 to 3 tenths of a second.

As a result, we cannot expect the data captured within a given $FP_i$ to be equal to the data within the corresponding $SP_i$. In fact, $FP_i$ may actually correspond to $SP_{i 
eq i}$. Or, as

![Figure 17 Sampling Intervals may not be in Alignment.](image)
is more likely, the data represented within FP\textsubscript{i} may correspond to data spread over two consecutive, second-run profile records, SP\textsubscript{j} and SP\textsubscript{j+1}. Figure 17 reflects such a case. For example, assume that FP\textsubscript{i} indicates that 27 Ticks of User-Mode CPU activity occurred during the \textsuperscript{i-th} sampling interval of the first-run. It is possible for the 27 Ticks of activity to be spread across two second-run profile records, SP\textsubscript{j} and SP\textsubscript{j+1}. SP\textsubscript{j} might indicate that 12 Ticks of activity took place, and SP\textsubscript{j+1} might indicate that 15 Ticks occurred. The same total but spread across two samples.

As a result of this lack of alignment, the graphs used to compare the two runs of the same process may be a bit misleading if viewed by individual samples. The activity spikes often have differing shapes and magnitudes.

Figure 18 graphically portrays the User-Mode CPU values that the Collector reported for the first and second runs. The graphs are similar but not exact. There exist occasional differences in magnitude, i.e. along the Y-axis, and shifts in time along the X-axis.

![Figure 18](image.png)  

**Figure 18** First-Run and Second-Run C2's CPU Activities (Depicted as Spikes)
Any analysis of the first-run and second-run profile records should not be based upon a comparison of an individual first-run profile record to an individual second-run profile record. A better method views the entire set of first-run profile records as a whole and the set of second-run profile records as a whole as well.

Figure 19 shows the same data values contained in Figure 18, but the Y-axis data is displayed in a *Cumulative*\(^1\) fashion. In this manner, we have conceptually treated the entire set of profile records as a whole. Displaying resource consumption data in a cumulative fashion has the added advantage of highlighting the relationship that exists between the consumption of a resource, in this case the CPU, and the progression of time.

---

1 Cumulative fashion means that each new Y-axis value is added to the sum of all preceding Y-axis values.
In the above figure, User-Mode CPU consumption data for two distinct runs of an invariant process is shown. Two runs of the same process indicate the importance of comparing the profile records as whole sets instead of as individual records. It is equally important to compare profile records from runs of different processes as whole sets as well. Therefore, the profile records for the Real processes will be treated as whole sets as will the profile records for their corresponding Duplicate processes.

4.1.2 Evaluating the Generated Duplicates During Playback

Conceptually, the first-run profile records are analogous to a Real process' profile records, and the second-run profiles are analogous to a Duplicate process’ profile records.

4.1.3 Analysis and Meaning of Cumulative Resource Distributions

Cumulative User-Mode CPU distributions for the real and duplicate C2 processes are shown in Figure 20. Cumulative User-Mode CPU time is shown along the Y-axis. As has been previously stated, User-Mode CPU time is a measure of work.

Figure 20 clearly indicates that process C2 has periods of activity and periods of inactivity. The periods of inactivity appear as regions where User-Mode CPU vs. elapsed

![Figure 20 Real and Duplicate Cumulative CPU consumptions for C2.](image-url)
time has a zero slope, i.e. the flat parts of the graph. The inactive regions of C2, and of processes in general, can be due to any number of conditions. The process may be waiting for user input, it may have instructed itself to sleep for a fixed period, the operating system may have blocked the process for scheduling purposes, the process may be blocked and waiting for another process to complete some task, or the process may be blocked on file or system I/O. Process C2 experiences many of those delay types during its lifetime.

The amount of work done and the rate with which work is performed are two considerations that need to be accounted for when analyzing the accuracy of the duplicate processes. We shall do so by evaluating how close an approximation the duplicate is to the real. In each time interval, a duplicate process attempts to perform the same quantity of work as its real counterpart.

Let $D_r(c) = M_r$ and $D_d(e) = M_d$ be functions that represent the real and duplicate User-Mode CPU distributions. For a given elapsed time value, $e$, $D_r(e)$ determines the corresponding quantity of User-Mode CPU time that the real process accumulated. Similarly, $D_d(e)$ determines the quantity of User-Mode CPU time that the duplicate process accumulated at elapsed time $e$.

Distribution functions $D_r(e)$ and $D_d(e)$ have inverse functions $E_r(u)$ and $E_d(u)$ respectively. The inverse relationship between $D_r(e)$ and $E_r(u)$ is expressed in Equations (4.1) and (4.2) and shown in Figure 21 and Figure 22. The functions $E_r(u)$ and $E_d(u)$ determine the minimum elapsed times that the real and duplicate processes required to accumulate $u$ units of User-Mode CPU time. It is possible for a specific $u$ units of User-Mode CPU time to correspond to many elapsed time values. The situation occurs when the process is idle and not consuming CPU cycles. In such cases, we define $E_r(u)$ to be the
minimum of the corresponding elapsed time values because the minimum represents the amount of elapsed time that it took to acquire the CPU time.

\[ U_R(e) = u, \text{ and } E_R(u) = \text{Minimum (e)} \]  \hspace{2cm} (4.1)

\[ U_R(E_R(u)) = u \]  \hspace{2cm} (4.2)

We determine how well the distribution \( E_D(u) \) approximates the \( E_R(u) \) distribution by applying Equations (4.3), (4.5), and (4.7).

\[ \text{Difference}(u) = |E_R(u) - E_D(u)| \]  \hspace{2cm} (4.3)

Equation (4.3) is used to calculate absolute differences in elapsed time for particular User-Mode CPU time values along the Real and Duplicate distributions. Figure 22 is a representation of Equation (4.3).

When \( u \) units of User-Mode CPU activity have been acquired by both the real and duplicate processes, the distributions differ by the absolute value of \( e_d \) minus \( e_r \). The elapsed time differences can then be normalized with respect to \( E_R(u) \), i.e. the elapsed time distribution of the real process.
Figure 21  User-Mode CPU Time Distributions.

Figure 22  Elapsed Time Distributions.

\[ \text{Difference}(u) = |e_d - e_r| \]
Figure 23  Area of the Real Process' Elapsed Time Distribution.

The shaded region in Figure 23 represents the area of the distribution $\mathcal{E}_R(u)$. The actual area under the curve is expressed in Equation (4.4).

$$\text{Area} (0, N) = \int_0^N \mathcal{E}_R (u) \, du$$  \hspace{1cm} (4.4)

Because we measure at discrete time intervals, we approximate this as with Equation (4.5).

$$\text{ApproxArea} (0, N) = \sum_{i=0}^{N} \mathcal{E}_R (u_i) \times \Delta u_i$$  \hspace{1cm} (4.5)

Similarly, we approximate the difference in area with Equation (4.6).

$$\text{AreaDifference} (0, N) = \sum_{i=0}^{N} \text{Difference} (u_i) \times \Delta u_i$$  \hspace{1cm} (4.6)
Equation (4.5) is used to determine the approximate area of the real process’ distribution curve. That area provides us with a basis to normalize the elapsed time differences from Equation (4.3).

We are concerned with the area of the curve because we wish to compare the elapsed time required for the duplicate and real processes to each acquire the same quantity of User-Mode CPU time.

Accumulated relative accuracy, as represented in Equation (4.7), is calculated by dividing the difference between the $\mathcal{E}_R(u)$ and $\mathcal{E}_D(u)$ distributions by the total area of the real process’ distribution.

Our monitoring tools provide us with discrete User-Mode CPU time values at one Tick intervals. There are 100 Ticks in one second of User-Mode CPU time. In Equations (4.5), (4.6), and (4.7), the change in $u_i$, denoted $\Delta u_i$, is always equal to one Tick. Therefore, all $\Delta u_i$ values in Equations (4.5), (4.6), and (4.7) factor out easily.

\[
\text{Accumulated Relative Accuracy } (0, N) = 100 \times \frac{Area\ Difference \ (0, N)}{Approx\ Area \ (0, N)} \tag{4.7}
\]
4.2 CPU Activity Comparisons

Figure 20 displays the User-Mode CPU consumption distributions for the real and duplicate versions of process C2. The duplicate C2 process makes CPU demands similar to the real C2 process. Applying Equation (4.7) to the real and duplicate distributions for process C2 yields an accumulated relative accuracy value of 98% with N = 3400 ticks.

A similar analysis for the real and duplicate GNU Emacs processes can also be performed. The User-Mode CPU consumption distributions for the real and duplicate GNU Emacs processes are shown in Figure 24. The accumulated relative accuracy value produced by Equation (4.7) with N = 900 ticks is 97.62%.

Figure 25 depicts the User-Mode CPU distributions for the real and duplicate Cl processes. Equation (4.7) indicates that the duplicate's User-Mode CPU distribution is a 96.57% relative accurate representation of the real process' distribution when N = 1100 ticks.
Figure 24  Real and Duplicate Cumulative CPU consumptions for GNU Emacs.

Figure 25  Real and Duplicate Cumulative CPU consumptions for Cl.
4.3 File I/O Comparisons

As mentioned previously, satisfying file I/O requirements is conceptually simple to control. The duplicates just read and/or write the proper number of bytes whenever the profile records indicate that they should. It is possible to gauge the accuracy of the duplicate's file activities by comparing the total input and output operations performed by the duplicates with those performed by the real processes. Table 1 contains the results for corresponding, three and one half minute periods of activity for the real and duplicate processes.

We have chosen a tabular format for file I/O data, instead of a graphical one, because the data matches so well. We are not surprised that the data values are identical. In Table 1, "Read Count" and "Write Count" refer to the quantity of read or write operations that were executed.

Table 1 Input and Output Operations Performed.

<table>
<thead>
<tr>
<th></th>
<th>Read Count</th>
<th>Read Bytes</th>
<th>Write Count</th>
<th>Write Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real C1</td>
<td>3</td>
<td>24,576</td>
<td>18</td>
<td>287,300</td>
</tr>
<tr>
<td>Duplicate C1</td>
<td>3</td>
<td>24,576</td>
<td>18</td>
<td>287,300</td>
</tr>
<tr>
<td>Real C2</td>
<td>10</td>
<td>81,920</td>
<td>33</td>
<td>254,000</td>
</tr>
<tr>
<td>Duplicate C2</td>
<td>10</td>
<td>81,920</td>
<td>33</td>
<td>254,000</td>
</tr>
<tr>
<td>Real GNU Emacs</td>
<td>335</td>
<td>46,616</td>
<td>93</td>
<td>163,840</td>
</tr>
<tr>
<td>Duplicate GNU Emacs</td>
<td>335</td>
<td>46,616</td>
<td>93</td>
<td>163,840</td>
</tr>
</tbody>
</table>
4.4 Memory Page Reference Comparisons

Through the monitoring tools developed, i.e. Reference Set Collector, it is possible for each duplicate executable to determine the pages that its corresponding real process referenced. Page reference data is contained within the profile records. For each duplicate, a one to one relationship is established between the pages that it accesses and the pages that the real process accessed. For every page that exists in the real process’ memory segment, there exists a corresponding page in the duplicate’s memory segment. A duplicate process, however, does have a few additional pages of text and data that do not have corresponding pages in the real process’ memory segment. These overhead pages contain the instructional data and code that the duplicate needs to operate. The referencing of these overhead pages can be detected, and the effects on a duplicate’s accuracy can be ascertained. We can evaluate a duplicate process’ referencing of pages, with respect to the real process, by intrusively monitoring both the duplicate and the real with the Reference Set Collector and Collector tools.

4.4.1 Reference Set Precision

By definition, invariant processes execute the same code during each execution. As they run the same code, they also refer to the same memory pages and the referencing is done at the same point of execution. This property of invariant processes allows us to evaluate and determine the precision of our reference set collection routines. If a process uses the same pages from run to run, then the monitoring tools should capture that same referencing in each of the runs. If the monitoring tools do not consistently report the same page referencing, then that is a reflection of the monitoring precision.

Equation (4.8) states that if a specific set of memory pages is referenced between User-Mode CPU times $i$ and $j$ during an execution of an invariant process, either a real or
duplicate invariant process, then that same set will be referenced during any run, \( r \), of that invariant process. In Equation (4.8), \( M_{ij} \) denotes a bit vector where each set bit, i.e. with a value of 1, represents a page that was referenced between User-Mode CPU times \( i \) and \( j \) for either a real or duplicate process. Also in Equation (4.8), \( rM_{ij} \) is used to denote a bit vector from run \( r \).

\[
(\forall r), rM_{i,j} = M_{i,j}
\]  

Equation (4.8) states that for invariant processes the referencing of pages is independent of run number.

We will refer to the work done between two User-Mode CPU time values, \( i \) and \( j \), as an Interval of Work because User-Mode CPU time is a measure of work. In Figure 26, an interval of work for an invariant process is depicted.

![Figure 26 An Interval of Work for an Invariant Process.](image)
The precision of our reference set monitoring tools can be determined for an interval of work by comparing the reported set of referenced pages, $r^i M_{i,j}$, in each of the $r$ runs of an invariant process. Each set, $r^i M_{i,j}$, contains a series of bits denoting the status of all of the process' pages. Each bit represents a single page. A value of 1 for bit $k$ indicates that the $k^{th}$ logical page was referenced, a zero means that no reference was made to logical page number $k$ during the interval of work.

Recall that a page's reference bit is contained within a page table entry. The Reference Set Collector examines the page table entries sequentially, beginning with entry zero. Page table entries are assigned to invariant processes in precisely the same manner during each run, (i.e. logical page number $k$ is consistently assigned to the $k^{th}$ page table entry). Therefore, it is possible to compare the reported reference sets in a straightforward manner. Figure 27 contains a pseudo-code algorithm for comparing reported reference sets in an interval of work.

```
Num_Different = 0

For k = 0 to Num_Memory_Pages
    if $1^i M_{i,j}(k) \neq 2^i M_{i,j}(k)$ then
        Num_Different = Num_Different + 1
    end if
end for

Percent Difference = ( Num_Different / Num_Memory_Pages ) * 100
```

Figure 27 Percent Difference Algorithm for Reported Page Reference Sets.
The status of each of the memory pages is compared. If both sets show a page as being referenced, or if both show it as not being referenced, then there is no difference. However, if the status bit for a page in one set differs from that in the other, the algorithm updates its difference counter. As discussed in Section 3.3.5, any differences in the bit settings are due to the limitations of our monitoring tools and to the operations of the kernel's page stealer process.

The "Percent Difference" between the two sets, and hence the precision of our tools, is calculated by dividing the difference counter by the number of memory pages. We consider the "Percent Difference" to be a measure of precision because we know that during both runs of the invariant process, \( r = 1 \) and \( r = 2 \), the same pages were referenced during the interval of work. The reported difference is entirely due to the limitations of our tools.

Table 2 contains the average "Percent Difference" values when comparing pairs from the ten runs of the real processes, C1, C2 and GNU Emacs. From the ten runs, all 45 possible set orderings were compared for each of the three processes.

<table>
<thead>
<tr>
<th></th>
<th>Number Runs</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real C1</td>
<td>10</td>
<td>9.7%</td>
</tr>
<tr>
<td>Real C2</td>
<td>10</td>
<td>5.2%</td>
</tr>
<tr>
<td>Real GNU Emacs</td>
<td>10</td>
<td>4.9%</td>
</tr>
</tbody>
</table>
4.4.2 Reference Set Comparisons

What is important is really page faults, not reference set accuracy; but reference set accuracy leads to page fault accuracy. This will be shown in this section.

The reference set accuracy of our duplicate processes can be ascertained by comparing the sets of referenced pages that both the real and the duplicate processes referenced during identical *intervals of work*. However, with the current implementation of the Workload Generator, it is not possible to apply the algorithm in Figure 27 directly.

The comparison algorithm in Figure 27 can only be used for comparing two runs of the same invariant program. As stated previously, page table entries are assigned to an invariant process in the same order during each execution. The comparison algorithm can therefore be used to compare the $k^{th}$ status bit in $2M_{i,j}$ with the $k^{th}$ status bit in $1M_{i,j}$.

Duplicate executables, although similar, are not the same as their real counterparts. Page table entries are not assigned to the duplicates in precisely the same order as they were assigned to the real processes. Therefore, it is not possible to compare the setting of the $k^{th}$ status bit in $R_{M_{i,j}}$ with the $k^{th}$ status bit in $D_{M_{i,j}}$, where $D_{M_{i,j}}$ denotes a run of the duplicate and $R_{M_{i,j}}$ denotes a run of the real.

<table>
<thead>
<tr>
<th>Number Runs</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicate C1</td>
<td>10</td>
</tr>
<tr>
<td>Duplicate C2</td>
<td>10</td>
</tr>
<tr>
<td>Duplicate GNU Emacs</td>
<td>10</td>
</tr>
</tbody>
</table>
It is still possible to compare a synthetic duplicate's page referencing patterns to a real process' referencing patterns. A duplicate process maintains a one to one relationship between a page in the real process' memory segment and a page in its own segment. Therefore, if a real process references 72 pages during an interval of work, then its synthetic duplicate should also reference 72 pages during an equivalent interval. If the duplicate references any amount other than 72, then that is a measure of the error.

Of course, referencing the same quantity of pages is not sufficient. Not only does a real invariant process reference the same quantity, it also references the same pages in its memory segment, (i.e. if page 10 is referenced in one run, it will be referenced in all runs). During repeated runs, a duplicate process should also reference the same pages in its memory segment.

Fortunately, a synthetic duplicate is an invariant process, therefore it references the same pages each time that it is executed. We can apply the algorithm in Figure 27 to show that this property of the real processes has been preserved. Table 3 contains the average "Percent Difference" values for several runs of the duplicate processes, C1, C2 and GNU Emacs.

To exactly match a real process during an interval of work, a duplicate process must satisfy two requirements: it must reference the same quantity of pages as its corresponding real process; and it must refer to the same set of pages in its memory segment during repeated executions. The small percent difference values in Table 3 indicate that the duplicate processes do indeed reference the same set of pages from run to run.

Although the same set of pages are referenced from run to run of the duplicates, each duplicate does not reference the same quantity of pages as their real counterparts. The duplicates have overhead that the real processes do not. Text code and data variables are
required by the duplicate processes if they are to emulate the real processes effectively. Currently, 20 pages of text space is required. As shown in Figure 28, the overhead text pages are easily identified after compilation, they constitute the first 20 pages of the text segment. The one to one relationship between a real text page and a duplicate text page begins after the duplicate’s 20th text page. As there are only 20 overhead pages of text that can be referenced, text segment overhead can never exceed 20 pages.

Although 20 overhead pages of text are present in each duplicate executable, all 20 are not a part of the duplicates working set size. Many of the text pages are not referenced after the first few moments of execution. They are only needed during start up are not referenced afterwards.

![Diagram: Real and Duplicate Text Segment Comparisons](image)

Figure 28 Real and Duplicate Text Segment Comparisons
A duplicate's data segment contains three types of data: profile records which indicate which real data segments were accessed when, data structures and variables used by the duplicate while emulating the real process, and memory space to emulate the data pages accessed by the real process.

The first 13 pages of a duplicate's data segment contain necessary data structures and variables that a duplicate makes use of while emulating a real process. Those 13 pages are distinguishable from the other data segment pages as the overhead text pages are from the other text pages. One to one relationships do not exist between any of the 13 overhead data pages and any of the real data pages.

Unlike the text segment, a duplicate's data segment overhead is not contained within the first pages. As shown in Figure 29, the profile records that the duplicates reference are located elsewhere within the data segment. When references are made to the profile records...

![Figure 29 Real and Duplicate Data Segment Comparisons](image-url)
records, some of the data pages, beyond the 13th page, are touched as a result. Therefore, it is possible for data segment overhead to exceed 13 pages.

Consider the following as an example. At time T, a duplicate executable examines the profile information found within Profile record, R. Profile record, R, happens to be physically located within page 30. The referencing of page 30 constitutes overhead.

The existence of 20 extra text pages, or the existence of 13 or more additional data pages, is not the real problem that overhead pages present to the system. The referencing of those pages is the problem. When a duplicate process emulates a real process’ interval of work, the duplicate must periodically refer to the overhead text and data pages. Therefore, during equivalent intervals of work, a duplicate will reference more pages than the corresponding real process.

An examination of the executions of duplicate processes has shown that the quantity of referenced, overhead pages in the examples is consistently below 30 pages (text and data combined). Table 4 contains statistics about the overhead text and data pages. The table’s

<table>
<thead>
<tr>
<th>Over Head Type</th>
<th>Mean</th>
<th>Median</th>
<th>Standard Deviation</th>
<th>Maximum</th>
<th>Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Pages</td>
<td>14.18</td>
<td>14</td>
<td>0.75</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>Data Pages</td>
<td>7.56</td>
<td>10</td>
<td>4.87</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Text &amp; Data</td>
<td>21.75</td>
<td>21</td>
<td>4.49</td>
<td>30</td>
<td>17</td>
</tr>
</tbody>
</table>
values were compiled from numerous intervals of work for the three distinct sets of real and
duplicate processes, C1, C2, and GNU Emacs. Table 5 also contains text and data page
statistics. The values presented in Table 5 show the real and duplicate page reference set
sizes for the individual processes and their duplicates.

The consistency of the overhead size is not surprising. Each duplicate’s overhead text
pages are comprised of the same executable code and each duplicate’s overhead data pages
are comprised of the same data structures and variables. Slight variations in the quantities
of referenced overhead pages are mainly due to the precision of the reference set collector,
to the quantity of profile records that a duplicate uses during an interval of work, and, most
of all, to the memory page locations of the accessed profile records. If the accessed profile
records are within a data page that the duplicate is instructed to reference during the interval
of work, then there is no overhead penalty for accessing those profile records. For instance,
if profile record R is located within memory page M, and M’s corresponding reference bit
indicates that it should be touched during the interval of work, then the accessing of R and
its contents comes with no overhead expense.

A consistent overhead page quantity of under 30 pages implies that the Workload
Generator cannot be very accurate for processes with “small” reference set sizes. A real
process that references 60 total pages of text and data cannot be emulated well by a
duplicate that references those 60 pages plus an additional 30 overhead pages. Equation
(4.9) expresses the method by which we compare a duplicate’s reference set size to the
reference set size of the real process. In Equation (4.9), \( Q_R \) represents the quantity of
referenced pages that the real process’ reference set contains, and \( Q_D \) has a similar
representation for the duplicate process. For corresponding intervals of work, the relative
accuracy of a duplicate is one minus the difference between the real process' and the
duplicate process' reference set sizes, divided by the size of the real process' reference set.

\[
\text{Relative Accuracy} = 1 - \frac{|Q_R - Q_D|}{Q_R} \quad (4.9)
\]

For a real process that references a total of 60 data and text pages and a duplicate that
references 90 data and text pages, the accuracy of the duplicate, as expressed by Equation
(4.9), is only 0.50, or 50%. As the size of the real process' reference set increases, so too
does the accuracy of the duplicate's reference set size. As shown in Figure 30, a duplicate
is capable of achieving 90% accuracy if the corresponding real process referenced 300
pages. A duplicate's accuracy continues to climb and reaches 96.7% if the real references
900 pages.

Relative Accuracy

![Figure 30 Reference Set Size Comparison curve.](image)
Table 5 contains reference set size data for the three pairs of real and duplicate processes. The computed relative accuracy values are also provided for the total pages. High relative accuracy values are important but must be kept in perspective. For duplicates and real processes alike, the referencing of pages causes page faults to occur when a system is short on memory. The duplicate executables attempt to reference the same pages as the real processes so the duplicate workload will experience a similar level of page faulting activity. Therefore, we need to explore the page faulting characteristics of the real and duplicate workloads and not just their page reference patterns. This can be done by experimenting with different quantities of main memory and will be addressed in the next section.

Table 5  Referenced Set Sizes for Real and Duplicate Processes

<table>
<thead>
<tr>
<th></th>
<th>Data Pages</th>
<th>Text Pages</th>
<th>Total Pages</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real C1</td>
<td>547</td>
<td>135</td>
<td>682</td>
<td>96.48%</td>
</tr>
<tr>
<td>Duplicate C1</td>
<td>557</td>
<td>149</td>
<td>706</td>
<td></td>
</tr>
<tr>
<td>Real C2</td>
<td>391</td>
<td>62</td>
<td>453</td>
<td>94.04%</td>
</tr>
<tr>
<td>Duplicate C2</td>
<td>407</td>
<td>73</td>
<td>480</td>
<td></td>
</tr>
<tr>
<td>Real GNU Emacs</td>
<td>83</td>
<td>91</td>
<td>174</td>
<td>89.66%</td>
</tr>
<tr>
<td>Duplicate GNU Emacs</td>
<td>86</td>
<td>106</td>
<td>192</td>
<td></td>
</tr>
</tbody>
</table>
4.4.2.1 Main Memory Size Changes and Page Faults

A stated goal of our thesis was to create duplicate workloads that would respond to memory changes as the real workloads. By changing the quantity of main memory that a machine can provide to a workload, one can expect subsequent changes to occur in the quantity of page faults. In this section, we explore how the real and duplicate workloads respond to several levels of main memory, 16 MB, 14 MB, 12 MB, 10 MB, and 8 MB.

The page fault statistics in Table 6 have been derived from the aggregate page faults made by the real GNU Emacs process, the real Cl process, and the real C2 process, as the workload ran from one identifiable point to some other identifiable point. The identifiable points can be marked by particular User-Mode CPU values for the processes, by synchronization points, or by any other events that can be readily identified. We used User-Mode CPU times as our identifiable points.

We shall term the work done by a workload between two identifiable points as an Interval of Activity. Identifying intervals of activity is necessary to ensure that we collect page fault activity from the same portion of the workload during the many runs of the workload. Table 7 contains the page fault statistics from the duplicate workload. The interval of activity is the same for the duplicate and real workloads. Elapsed time statistics for the interval of activity are respectively provided in Table 8 and Table 9. As the amount of main memory is changed, so too is the elapsed time required to complete the interval of activity.

The page fault data show there exists a high degree of consistency between the two workloads, real and duplicate, when the system has between 10 and 16 Mega Bytes of main memory. The mean page faults for the duplicate workload are consistently within one standard deviation of the real workload's mean values. In fact, with 99% confidence we
cannot reject the hypothesis that the real and duplicate page fault means are from the same distribution. However, with 8 MBs of main memory the real workload experiences an average of 5,204.1 page faults to the duplicates 3,866.2 faults.

Table 6  Real Workload Page Fault Statistics (Pages)

<table>
<thead>
<tr>
<th>Client Memory</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 MB</td>
<td>8</td>
<td>52.7</td>
<td>51</td>
<td>7.5</td>
<td>47</td>
<td>68</td>
</tr>
<tr>
<td>14 MB</td>
<td>10</td>
<td>1104.9</td>
<td>1131.5</td>
<td>115.83</td>
<td>818</td>
<td>1225</td>
</tr>
<tr>
<td>12 MB</td>
<td>10</td>
<td>1378.6</td>
<td>1323.5</td>
<td>174.5</td>
<td>1226</td>
<td>1814</td>
</tr>
<tr>
<td>10 MB</td>
<td>11</td>
<td>2282.1</td>
<td>2279</td>
<td>99.35</td>
<td>2128</td>
<td>2504</td>
</tr>
<tr>
<td>8 MB</td>
<td>8</td>
<td>5204.1</td>
<td>5097</td>
<td>994.5</td>
<td>4060</td>
<td>6755</td>
</tr>
</tbody>
</table>

Table 7  Duplicate Workload Page Fault Statistics (Pages)

<table>
<thead>
<tr>
<th>Client Memory</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 MB</td>
<td>6</td>
<td>54.67</td>
<td>61.5</td>
<td>24.1</td>
<td>14</td>
<td>76</td>
</tr>
<tr>
<td>14 MB</td>
<td>8</td>
<td>1127.5</td>
<td>1092</td>
<td>107.3</td>
<td>1000</td>
<td>1312</td>
</tr>
<tr>
<td>12 MB</td>
<td>11</td>
<td>1525.5</td>
<td>1556</td>
<td>130.6</td>
<td>1267</td>
<td>1756</td>
</tr>
<tr>
<td>10 MB</td>
<td>10</td>
<td>2224.5</td>
<td>2214.5</td>
<td>86.8</td>
<td>2122</td>
<td>2365</td>
</tr>
<tr>
<td>8 MB</td>
<td>10</td>
<td>3866.2</td>
<td>3891</td>
<td>182.3</td>
<td>3517</td>
<td>4189</td>
</tr>
</tbody>
</table>
### Table 8  Real Workload Interval of Activity Durations (Seconds)

<table>
<thead>
<tr>
<th>Client Memory</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 MB</td>
<td>8</td>
<td>141.3</td>
<td>141</td>
<td>2.96</td>
<td>136</td>
<td>145</td>
</tr>
<tr>
<td>14 MB</td>
<td>10</td>
<td>234.0</td>
<td>235</td>
<td>21.8</td>
<td>195</td>
<td>280</td>
</tr>
<tr>
<td>12 MB</td>
<td>10</td>
<td>260.4</td>
<td>255.5</td>
<td>18.4</td>
<td>240</td>
<td>296</td>
</tr>
<tr>
<td>10 MB</td>
<td>11</td>
<td>365.7</td>
<td>362.5</td>
<td>36.2</td>
<td>341</td>
<td>401</td>
</tr>
<tr>
<td>8 MB</td>
<td>8</td>
<td>786.0</td>
<td>790</td>
<td>84.0</td>
<td>620</td>
<td>930</td>
</tr>
</tbody>
</table>

### Table 9  Duplicate Workload Interval of Activity Durations (Seconds)

<table>
<thead>
<tr>
<th>Client Memory</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 MB</td>
<td>6</td>
<td>138.6</td>
<td>140</td>
<td>2.4</td>
<td>135</td>
<td>140</td>
</tr>
<tr>
<td>14 MB</td>
<td>8</td>
<td>262.3</td>
<td>265</td>
<td>20.0</td>
<td>240</td>
<td>298</td>
</tr>
<tr>
<td>12 MB</td>
<td>11</td>
<td>274.5</td>
<td>265</td>
<td>36.5</td>
<td>240</td>
<td>380</td>
</tr>
<tr>
<td>10 MB</td>
<td>10</td>
<td>369.2</td>
<td>373</td>
<td>17.6</td>
<td>333</td>
<td>395</td>
</tr>
<tr>
<td>8 MB</td>
<td>10</td>
<td>846.1</td>
<td>843.5</td>
<td>92.4</td>
<td>700</td>
<td>1010</td>
</tr>
</tbody>
</table>
We consider 8 MBs to be the limit of our duplicate workload's abilities for this real workload. The real workload pages and swaps almost continuously with just 8 MBs of memory and it is not likely that anyone would wish to run a workload under those thrashing conditions.

Although it is unlikely that anyone would want to operate this workload with only 8 MBs of memory, we can still explain the reasons why the duplicate workload experiences far fewer page faults than the real workload. Each duplicate process attempts to touch pages as it emulates a real process. If a reference bit is set, then the duplicate references a page. Because we cannot distinguish between a single reference to a page and multiple references to a page, the algorithm was designed so the duplicate process would reference the page only once before moving on to the next profile record. The real process may have referenced the page more than once during that interval of time. Under thrashing conditions, it is likely that the second, third, or any later reference to the page will result in page faults. So, the repeated referencing of pages by the real processes can result in additional paging activity when the system is thrashing.

4.5 Multiple Client Workload Emulations

One of the goals for the Workload Generator project was to handle both single machine and multiple machine emulations. We designed the Workload Generator so it could monitor real workloads on a single of a file server's clients, and then generate duplicates capable of being run on any number of the file server's clients. The challenge for the duplicate workloads is to exhibit resource demand patterns, across the full range of $I$ to $N$ clients, that match those made by the real workloads when run on the same cluster of clients. The duplicate executables should not be provided with any knowledge regarding the behavior of the real workload when there was more than one active client.
Of course, with more than 1 client in operation, the quantity of jobs in the file server’s queues is increased. That translates into increased waiting times for all I/O requests directed towards the server. Consequently, a real workload’s resource demand patterns, when there are no other active clients, differs from its demand patterns when there are \( N \) active clients. For the class of invariant process, these differences have to do with the timing of requests, not with the actual requests themselves. The work done by invariant processes remains constant although the elapsed time to do the work increases.

From the file server’s perspective, each diskless client is just a black box that makes requests. If the real processes are emulated properly by the duplicates, then the file server should not be able to distinguish between a black box running duplicates and a black box running the real processes. That is the ideal case.

We can gauge the success of our duplicate workload by examining the resources of the file server, i.e. its CPU and disks, as the number of active clients changes from 1 to \( N \). As an additional test, response time measures taken from the clients are also provided.

4.5.1 Evaluating Multiple Client Workload Emulations

The evaluation of our emulations of multiple clients is based upon response time measures and file server resource statistics. We examine the utilization of the file server’s resources to make sure that the duplicate and real processes stress the system to a similar extent. Response time measures are taken as a means for backing up the server data. Response times should change as the usage of the file server’s resources changes.

Before examining specific multiple client results for our tested workload, we must detail some difficulties associated with the testing of clusters of workstations. A typical workload experiences cycles of relative high and low activity, with respect to the resource demands that it makes. When workloads are run on multiple machines there inevitably are
occurrences when the majority of the workloads all undergo their high and low activities together. This type of synchronization leads to a high degree of variance in response times, and in server resource utilizations.

In all of our tables, we report the standard deviations with the mean values. For our duplicate workloads to emulate the real workload accurately, it is vital that similar mean values be accompanied with similar standard deviations. If the means are similar but the standard deviations are not, then the duplicate emulations are not modeling the behavior of the real processes consistently.

4.5.2 File Server CPU Utilization

Table 10 and Table 11 contain file server busy percentages when a cluster of 10 MB diskless clients are executing either the real or the duplicate workloads. The busy percentages represent 5 second intervals. With two to eight clients running either a real or duplicate workload, the duplicate's average file server busy percentages remain within one standard deviation of the real's busy percentages. Also, the standard deviations decrease as the number of clients is increased from two to eight. The inverse relationship between the standard deviations and the quantity of clients can be observed in the data regarding all file server resources. With fewer clients operating, we can expect the periods of high and low activities to more drastically affect the server's resources.

The data shows that the file server's CPU is not the bottleneck. The mean busy percentages never exceed 50%. Figure 31 contains a graph of the mean file server busy percentages.
Table 10  Real Workload Server CPU Busy Percentages (5 Second Samples).

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>200</td>
<td>48.0</td>
<td>47.8</td>
<td>5.6</td>
<td>33.4</td>
<td>68.2</td>
</tr>
<tr>
<td>6</td>
<td>175</td>
<td>47.3</td>
<td>47.2</td>
<td>8.8</td>
<td>10.2</td>
<td>69.4</td>
</tr>
<tr>
<td>4</td>
<td>175</td>
<td>39.1</td>
<td>41.2</td>
<td>15.7</td>
<td>3.2</td>
<td>67.2</td>
</tr>
<tr>
<td>2</td>
<td>175</td>
<td>26.8</td>
<td>29.6</td>
<td>14.7</td>
<td>2.6</td>
<td>55.6</td>
</tr>
</tbody>
</table>

Table 11  Duplicate Workload Server CPU Busy Percentages (5 Second Samples).

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>200</td>
<td>44.8</td>
<td>44.6</td>
<td>3.8</td>
<td>31.0</td>
<td>56.0</td>
</tr>
<tr>
<td>6</td>
<td>175</td>
<td>43.3</td>
<td>44.2</td>
<td>6.0</td>
<td>16.8</td>
<td>53.6</td>
</tr>
<tr>
<td>4</td>
<td>175</td>
<td>38.9</td>
<td>41.8</td>
<td>8.9</td>
<td>10.2</td>
<td>51.8</td>
</tr>
<tr>
<td>2</td>
<td>175</td>
<td>26.4</td>
<td>26.2</td>
<td>13.7</td>
<td>3.4</td>
<td>53.8</td>
</tr>
</tbody>
</table>
4.5.3 Disk I/O

Figure 32 shows the total disk I/O statistics, reading plus writing, for the real and duplicate workloads. Breakdowns of the reading and writing activities for both workloads are located in Table 12, Table 13, Table 14, and Table 15. All samples are 5 seconds in duration.

In all cases, the duplicate workload averages are within one standard deviation of the real workload averages. In fact, with 99% confidence we cannot reject the hypothesis that the real and duplicate disk reading and writing means are from the same distribution. Also, the real and duplicate standard deviations are similar. With just two clients, the standard deviations are the largest. Once again, large variances are expected with a few clients in operation.

The real and duplicate disk writing statistics in Table 14 and Table 15 indicate a drop in the average number of kilobytes written to disk when the quantity of clients increases from six clients to eight clients. However, the disk reading data shows an increase in the number of kilobytes read. We believe that this phenomena is due to the file server’s disk cache. Disk reads are avoided if the information is found to be in the file server’s disk cache. Because lightly loaded systems have file server cache hit ratios which are higher than the hit ratios of heavily loaded systems, the number of disk reads required to perform a quantity of work can be greater on heavily loaded systems. When the disk subsystem’s performance peaks, the increase in the required disk reads reduces the number of write operations that can be performed in a five second sample.
Figure 31  File Server CPU Utilization: 10 MB Clients.

Figure 32  Total Reading plus Writing Disk I/O: 10 MB Clients (Kilobytes).
Table 12  Real Workload Disk Reading Statistics on 10 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>184</td>
<td>789.3</td>
<td>782.5</td>
<td>170.6</td>
<td>291</td>
<td>1255</td>
</tr>
<tr>
<td>6</td>
<td>161</td>
<td>695.5</td>
<td>708</td>
<td>230.7</td>
<td>52</td>
<td>1311</td>
</tr>
<tr>
<td>4</td>
<td>161</td>
<td>625.9</td>
<td>648</td>
<td>327.5</td>
<td>9</td>
<td>1484</td>
</tr>
<tr>
<td>2</td>
<td>322</td>
<td>437.5</td>
<td>396</td>
<td>346.7</td>
<td>0</td>
<td>1364</td>
</tr>
</tbody>
</table>

Table 13  Duplicate Workload Disk Reading Statistics on 10 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>184</td>
<td>818.8</td>
<td>800.5</td>
<td>185.8</td>
<td>236</td>
<td>1415</td>
</tr>
<tr>
<td>6</td>
<td>161</td>
<td>680.7</td>
<td>664</td>
<td>232.6</td>
<td>31</td>
<td>1324</td>
</tr>
<tr>
<td>4</td>
<td>161</td>
<td>687.5</td>
<td>648</td>
<td>356.5</td>
<td>52</td>
<td>1527</td>
</tr>
<tr>
<td>2</td>
<td>322</td>
<td>451.3</td>
<td>337</td>
<td>387.0</td>
<td>0</td>
<td>1499</td>
</tr>
</tbody>
</table>
Table 14  Real Workload Disk Writing Statistics on 10 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>184</td>
<td>673.8</td>
<td>653.5</td>
<td>201.9</td>
<td>196</td>
<td>1245</td>
</tr>
<tr>
<td>6</td>
<td>161</td>
<td>725.6</td>
<td>726</td>
<td>248.4</td>
<td>104</td>
<td>1469</td>
</tr>
<tr>
<td>4</td>
<td>161</td>
<td>527.4</td>
<td>492</td>
<td>365.9</td>
<td>0</td>
<td>1664</td>
</tr>
<tr>
<td>2</td>
<td>322</td>
<td>351.9</td>
<td>233.5</td>
<td>346.1</td>
<td>0</td>
<td>1389</td>
</tr>
</tbody>
</table>

Table 15  Duplicate Workload Disk Writing Statistics on 10 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>184</td>
<td>649.8</td>
<td>652</td>
<td>180.5</td>
<td>124</td>
<td>1087</td>
</tr>
<tr>
<td>6</td>
<td>161</td>
<td>711.8</td>
<td>700</td>
<td>228.0</td>
<td>112</td>
<td>1440</td>
</tr>
<tr>
<td>4</td>
<td>161</td>
<td>560.7</td>
<td>523</td>
<td>298.1</td>
<td>73</td>
<td>1364</td>
</tr>
<tr>
<td>2</td>
<td>322</td>
<td>363.4</td>
<td>176</td>
<td>386.0</td>
<td>0</td>
<td>1465</td>
</tr>
</tbody>
</table>
4.5.4 Response Time and Main Memory Variations

In this section we present response time measures for multiple client configurations as the amount of client main memory is varied. Response time variations are a good means for evaluating levels of performance because they represent the system from the user’s perspective. Response time measures help give meaning to other system measures, such as disk I/O, server CPU utilization, and throughput. It is often difficult for a user to understand the impact of a 20% jump in file server CPU utilization, without providing response time variations as well. A doubling of response time can more easily be understood. In other words, it is the user’s sense of performance that matters most.

In a cluster of 10 MB diskless workstations running either the real or duplicate workloads, Figure 33 shows how the response time of compiling a 4 Kilobyte C program increases as the number of diskless clients is increased. The X-axis values, between 0 and 8, refer to the number of additional diskless clients operating during the timing of the compiles. Figure 34 shows how the response time increases as the quantity of 8 MB diskless workstations increases from 0 active clients to 8 active clients. The term active client is used to indicate that the client is executing either a duplicate or real workload.

Response time data for the real and duplicate workloads, for clusters with 12 MB, 10 MB and 8 MB diskless clients are provided in the following tables. In all circumstances, the mean response times for the duplicate workloads are within one standard deviation of the real workload’s mean times. For all but two cases, there exists a 99% confidence interval indicating that we cannot reject the hypothesis that the response times from the real and duplicate distributions are from the same distribution. The two exceptions occur with 4 and 6 active 12 MB client workstations. We have 90% confidence for those two cases.
Figure 33  Compile Response Times: 10 MB Clients

Figure 34  Compile Response Times: 8 MB Clients
Table 16  Real Workload Response Times on 12 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>8.08</td>
<td>8.0</td>
<td>1.52</td>
<td>5.7</td>
<td>11.9</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>6.69</td>
<td>6.4</td>
<td>0.88</td>
<td>5.7</td>
<td>9.1</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>6.59</td>
<td>6.2</td>
<td>1.02</td>
<td>5.4</td>
<td>8.7</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>5.86</td>
<td>5.8</td>
<td>0.52</td>
<td>5.3</td>
<td>7.7</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 17  Duplicate Workload Response Times on 12 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>8.95</td>
<td>8.65</td>
<td>1.67</td>
<td>6.2</td>
<td>12.8</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>6.10</td>
<td>5.9</td>
<td>0.61</td>
<td>5.4</td>
<td>7.6</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>5.86</td>
<td>5.6</td>
<td>0.56</td>
<td>5.4</td>
<td>7.8</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>5.72</td>
<td>5.6</td>
<td>0.50</td>
<td>5.3</td>
<td>7.8</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>
Table 18  Real Workload Response Times on 10 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>27.55</td>
<td>27.2</td>
<td>7.5</td>
<td>15.3</td>
<td>40.7</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>18.67</td>
<td>18.6</td>
<td>4.6</td>
<td>12.0</td>
<td>28.9</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>12.09</td>
<td>11.7</td>
<td>2.3</td>
<td>7.6</td>
<td>18.1</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>7.45</td>
<td>7.3</td>
<td>1.4</td>
<td>5.3</td>
<td>10.3</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 19  Duplicate Workload Response Times on 10 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>50</td>
<td>28.61</td>
<td>27.15</td>
<td>10.72</td>
<td>11.0</td>
<td>57.9</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>18.66</td>
<td>17.9</td>
<td>8.06</td>
<td>6.8</td>
<td>39.0</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>11.97</td>
<td>10.7</td>
<td>3.76</td>
<td>8.5</td>
<td>22.6</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>8.08</td>
<td>7.9</td>
<td>1.92</td>
<td>5.4</td>
<td>11.4</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>
### Table 20  Real Workload Response Times on 8 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>38.5</td>
<td>41.9</td>
<td>8.91</td>
<td>17.9</td>
<td>56.5</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>27.04</td>
<td>26.0</td>
<td>5.12</td>
<td>19.3</td>
<td>43.6</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>15.46</td>
<td>15.0</td>
<td>3.65</td>
<td>11.4</td>
<td>27.3</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>9.34</td>
<td>9.6</td>
<td>1.92</td>
<td>5.7</td>
<td>14.3</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

### Table 21  Duplicate Workload Response Times on 8 MB Workstations. (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>42.1</td>
<td>42.4</td>
<td>6.11</td>
<td>30.3</td>
<td>56.0</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
<td>27.99</td>
<td>29.0</td>
<td>5.28</td>
<td>15.0</td>
<td>35.8</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>16.22</td>
<td>15.4</td>
<td>3.64</td>
<td>9.6</td>
<td>24.7</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>8.94</td>
<td>8.9</td>
<td>2.4</td>
<td>5.8</td>
<td>14.9</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>
It is worth noting that the response time standard deviations do not behave as the standard deviations of the file server's resources. With few exceptions, rather than an inverse relationship, there exists a direct relationship between the quantity of active clients and the response time standard deviations. Saturation of the disk drives may be the cause. With many clients in operation, the disks occasionally become saturated. The service queues grow in length and response time variations increase. As shown in Sections 4.5.2 and 4.5.3, the utilization of the file server's resources do not, however, experience increases in variance. When a resource, such as the disks, becomes saturated they perform at a consistent, peak utilization. And, the utilization of other resources levels off as well.

4.6 Differing CPU Platforms

One of the stated goals of the Workload Generator project was to produce duplicate workloads capable of being utilized on different CPU platforms. In particular, we aspired to monitor a real workload on CPU A, and play back a duplicate of that workload on CPU B, without requiring any knowledge of how the real workload would operate on CPU B. In this section we will discuss the Workload Generator’s abilities to create duplicates that can run on different CPUs with the same instruction set. We will also detail the two main problems with our approach. The first involves the proportion of integer to floating point calculations, the second involves the rate at which the duplicates progress.

4.6.1 Converting User-Mode CPU Time to A Quantity of Work

In this section we explain why we must convert duplicates that operate on User-Mode CPU time basis into duplicates that operate on a Quantities of Work-basis. Recall that the time-based approach for emulating real processes involved following the profile record instructions for consuming User-Mode CPU ticks, performing file I/O, and referencing memory pages. After consuming the proper amount of the User-Mode CPU time, the
duplicate then sleeps for the remainder of the period. Upon awakening, the duplicate can proceed to the next profile record.

Our goal is to monitor a real workload on CPU A, and play back a duplicate of that workload on CPU B. Since CPU A and CPU B cannot be assumed to operate at the same speeds, our reliance upon User-Mode CPU time to approximate how much work to perform has to be replaced with another method. If the two CPUs operate at different speeds, then differing numbers of instructions can be performed in the same amount of User-Mode CPU time on the two CPUs. Thus, the workload being run upon the two will not be the same if User-Mode CPU time is used as the measure of work. Since we must have equivalent workloads to cross CPU platforms properly, User-Mode CPU time cannot be relied upon for this.

If we ran the real processes on two CPU platforms, then the same amount of work, i.e. the same instructions (assuming the same instruction set), would always be executed on both platforms. That characteristic must be preserved by the duplicate processes. We can partially accomplish that by translating the instruction sets for the duplicate processes from a time based format into a Quantities of Work-based format. Instead of running for a particular number of CPU ticks and sleeping for the remainder of the period, the duplicates should run until they have reached a certain destination in the duplicate's code. The real processes run that way. The duplicates should run that way as well.

The translation from the time format to the work format is done by initially running the duplicate workload on CPU A (the same on which the real was monitored), before running on CPU B. During that initial run on CPU A, whenever the processes are instructed to run for a number of ticks, they record their initial location in the program code; record intermediate locations as they progress; and then record their final position when the
interval timer sounds. These location beacons can then be used in any additional runs, on this CPU or on others, in place of the interval timers. No matter the platform, the duplicates will execute the same instructions on each, as the real would do.

The duplicate processes must also execute instructions at the same rate as the real processes. Therefore, in addition to recording the initial, intermediate, and final locations, a duplicate also records how long it was instructed to pause before continuing. Recall that during the time-based phase, a duplicate is instructed to run for a number of ticks. After acquiring the specified quantity of ticks, the duplicate then pauses for the remainder of the profile record's sampling interval, i.e. 1 second. If no time remains in the interval, then no sleeping is required. During the time-based phase, the periodic sleeping keeps the duplicates progressing at the proper rate.

Because the time to pause is particular to one CPU, the rates of the duplicates will differ from the rates of the real processes if they are run on a different CPU. The pause time prevents the duplicates from acquiring CPU time. When a pause is issued, the duplicate is placed on the wait queue by the operating system's scheduling process. Thus, the duplicates cannot receive additional CPU time until the pause time expires. The real processes might not be prevented from obtaining the CPU because they may not be in the wait queue. On occasion, the real processes can proceed when the duplicates cannot. Therefore, when run on a different CPU the real processes can progress at a faster rate than the duplicate processes.

It is also possible for the duplicates to progress at a faster rate than the real processes. If the real processes require much screen I/O, such as in the moving and displaying of graphical objects, then the CPU's processing speed can be tempered by the screen I/O. Screen I/O can keep a process from progressing as fast as the CPU would otherwise allow.
Our duplicate processes do not emulate screen I/O behavior because at this time we cannot monitor screen I/O. Thus, if the duplicates are run on a faster CPU and the real processes require considerable screen I/O operations, then the duplicates can proceed at a faster rate than the real processes.

In Figure 35 we have provided an overview of the two ways that the duplicate processes can have their rates adversely affected. Whether we move from a faster CPU to a slower CPU, or vice versa, a real process that performs CPU actions and very little screen I/O will progress at a faster rate than the duplicate processes. The pauses keep the duplicates from progressing when the real processes can proceed.

The situation becomes more complex for real processes that require screen I/O. The pause values still affect the duplicates in the same manner and tend to slow them down.

<table>
<thead>
<tr>
<th>Move From Faster to Slower CPU</th>
<th>Move From Slower to Faster CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU Only Processes</strong></td>
<td><strong>CPU Only Processes</strong></td>
</tr>
<tr>
<td>Pauses Slow Duplicates</td>
<td>Pauses Slow Duplicates</td>
</tr>
<tr>
<td><strong>Processes With Screen I/O</strong></td>
<td><strong>Processes With Screen I/O</strong></td>
</tr>
<tr>
<td>Pauses Slow Duplicates AND</td>
<td>Pauses Slow Duplicates BUT</td>
</tr>
<tr>
<td>Screen I/O Affects Decrease</td>
<td>Screen I/O Affects Increase</td>
</tr>
<tr>
<td>So</td>
<td>So</td>
</tr>
<tr>
<td>Reals Go Even Faster than</td>
<td>Reals Slow Down</td>
</tr>
<tr>
<td>the Duplicates</td>
<td>BUT Duplicates do Not Slow</td>
</tr>
</tbody>
</table>

Figure 35  A Duplicate’s progression rate can differ for two reasons.
However, the screen I/O affects on the real processes are less clear. Faster CPUs are slowed more by screen I/O than slower CPUs\(^1\). When moving from a faster CPU to a slower CPU, the screen I/O affects fade. Just the opposite occurs when moving from a slower CPU to a faster CPU. The screen I/O is more of a factor on the faster CPU than on the slower. The potential work that the real processes could perform is reduced by the screen I/O influences. The duplicates, which have no screen I/O emulation abilities, can more fully realize their potential. Thus, the real processes are slowed down more than the duplicates.

In three of the four cases in Figure 35, the real processes progress at a faster rate than the duplicates. Only when we move from a slower CPU to a faster CPU and the real processes require screen I/O is there a chance for the duplicates to proceed at a faster pace. If the screen I/O's affects are greater than the pause value’s affects, then the real processes will progress slower than the duplicates.

The tested workload presented in this chapter performs very little screen I/O. A few lines of text are printed periodically. The effects of screen I/O should not present themselves for this workload. Therefore, we expect the duplicate processes to progress slower than the real processes. The effects of screen I/O can be observed on the CAD/CAM workload presented in the next chapter.

### 4.6.2 Processing Speed - Floating Point and Fixed Point

A CPU performs both floating point and fixed point (or integer) operations. The relative floating point and integer processing capabilities of two CPUs may differ significantly. In this section we examine how the floating point and integer processing

\(^1\) The amount of time spent waiting on the screen I/O represents more of a loss in potential work to the faster CPU than it does to the slower CPU.
speed differences of a new CPU reduces the accuracy of the Quantities of Work-based approach.

To understand the performance of two or more CPUs on a particular workload, or process, it is important to understand the floating point and integer demands of the workload, as well as the floating point and integer processing capabilities of the CPUs. For three CPUs, Table 22 contains the normalized User-Mode CPU times\(^1\) required to complete two distinct processes. The first process, Integer process, performs a sequence of integer computations. The second process, Floating Point process, performs floating point calculations.

For any machine M, let FP(M) and INT(M) denote the floating point and integer processing speeds of machine M. FP(M) is calculated by dividing 1 by the User-Mode CPU time required to execute the Floating Point process. Likewise, INT(M) is calculated by dividing 1 by the User-Mode CPU time required to complete the Integer process.

Table 22 Integer and Floating Point User-Mode CPU Times: Relative to CPU A.

<table>
<thead>
<tr>
<th>Type of Workload</th>
<th>CPU A</th>
<th>CPU B</th>
<th>CPU C</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 MHz 68040</td>
<td>50 MHz 68030 50 MHz 68882</td>
<td>25 MHz 68030 25 MHz 68882</td>
<td></td>
</tr>
<tr>
<td>Integer(^1)</td>
<td>1.0 (12.9 secs)</td>
<td>1.1783</td>
<td>1.7287</td>
</tr>
<tr>
<td>Floating Point(^1)</td>
<td>1.0 (18.2 secs)</td>
<td>4.489</td>
<td>6.912</td>
</tr>
</tbody>
</table>

\(^1\) The normalized times are taken with respect to the 12.9 seconds of User-Mode CPU time that CPU A required to complete the Integer process and the 18.2 seconds to complete the Floating Point process. Therefore, CPU C took 6.912 x 18.2 seconds, or 125.79 seconds of User-Mode CPU time to complete the Floating Point process.
For both floating point and integer, CPU A is faster than CPU B which is faster than CPU C. Let FP(A) be the floating point processing speed of processor A, and INT(A) be the integer processing speed of CPU A. Also, FP(B), FP(C), INT(B), and INT(C) have similar representations for processors B and C. The ratios of the floating point and integer processing speeds are expressed in Equations (4.10), and (4.11).

\[
\frac{FP(A)}{FP(B)} \gg \frac{INT(A)}{INT(B)} > 1 \quad (4.10)
\]

\[
\frac{FP(B)}{FP(C)} \sim \frac{INT(B)}{INT(C)} \sim 1 \quad (4.11)
\]

CPU A’s floating point capabilities have improved over CPU B’s by a larger proportion than the integer capabilities. Although only by a small margin, the floating point processing abilities of CPU B have improved over CPU C’s by a larger proportion than the integer capabilities.

Because the floating point to integer ratios can differ from one CPU to another, floating point intensive processes will receive a different level of speed up, or slow down, than integer intensive processes if run on a different CPU. Currently, we have not implemented a monitoring means to determine the proportion of integer and floating point operations of a real process. Therefore, we expect our generated duplicate processes to misrepresent the proportion of integer and floating point operations to perform. Furthermore, an incorrect proportion should lead to differences in the CPU times.

4.6.3 Processing Speed - From a Faster CPU to Slower CPU

In this section we examine how the Quantities of Work-based duplicates created on a faster CPU perform when executed on a slower CPU. CPUs B and C from Table 22 were
used for these tests. The data shown within this section highlight the two major problems with our current implementation: the floating point and integer proportion problem, and the rate of progress problem. As will be shown, the floating point and integer proportion problem is not particularly significant for these two CPUs. The rate of progress problem is more significant. The rate of progress problem causes shifts to the left and right in graphs. The floating point and integer proportion problems causes shifts up and down in the graphs.

We tested the Quantities of Work-based approach by monitoring the real workload on CPU B, an HP 400t workstation equipped with a 50 MHz Motorola 68030 and a 50 MHz Motorola 68882 floating point co-processor. We played back the Quantities of Work-based duplicate workload on CPU C, an HP 370 equipped with a 25 MHz Motorola 68030 and a 25 MHz Motorola 68882 floating point co-processor. As shown in Table 22, the HP 400t is slightly faster than the HP 370 in floating point and integer calculations.

We also ran the real workload on the slower CPU C, i.e. the HP 370, so comparisons between the actual workload and the work-based duplicate could be made. We provide User-Mode CPU consumption graphs for the real C1, C2, and GNU Emacs processes in Figure 36, Figure 37, and Figure 38 respectively. Also within those figures, we provide User-Mode CPU consumption graphs for “Integer-Based” duplicates and “Floating Point-Based” duplicates. Although the “Integer-Based”, or integer, duplicates consume CPU ticks by performing integer calculations, they are not 100% integer. Some floating point is required. The “Floating Point-Based”, or float, duplicates mainly perform floating point operations to consume CPU cycles. Of course, they are not entirely floating point and do perform some integer calculations.
4.6.3.1 The Floating Point and Integer Proportion Problem

We have provided data for the integer and floating point duplicates to highlight the significance of the slight floating point and integer performance differences of CPUs B and C. The figures show that the floating point and integer intensive duplicates consume CPU cycles in a similar manner to the real processes. However, the differences are larger than they were when the duplicates and real processes were both run on the same CPU. There are differences in the rates that the real and duplicate processes progress and there are differences in the amounts of User-Mode CPU times that they consume. In this section we will concentrate on the differences in User-Mode CPU time.

In Figure 36 and Figure 37, differences in the quantities of consumed User-Mode CPU ticks are evident. The integer C1 and C2 processes consume fewer ticks of User-Mode CPU time than their real counterparts. The floating point intensive C1 and C2 duplicate processes are a closer approximation to the real processes than are the integer duplicates. However, as shown in Figure 38, the integer GNU Emacs duplicate is clearly a better representation of the real GNU Emacs process. The floating point intensive duplicate consumes more CPU ticks than both the real or the integer GNU Emacs. In all cases, we attribute the differences in consumed User-Mode CPU ticks to the differences in floating point and integer processing capabilities of the two CPUs. Additional statements regarding the graphs follows the figures.
Figure 36   CPU Consumption: Real C1 vs. Duplicate C1s (CPU C).
Figure 37  CPU Consumption: Real C2 vs. Duplicate C2s (CPU C).

Figure 38  CPU Consumption: Real GNU Emacs vs. Duplicates (CPU C).
As stated above, the floating point and integer aspects of the real processes cannot as of yet be distinguished. Duplicate processes attempt to utilize the CPU to the same degree as the real processes, but they do not attempt to have the same floating point to integer processing ratio. As a result, moving to a different CPU might yield improvements in the real processes that the duplicates cannot approximate. Or, the real processes might perform worse than the duplicates.

The floating point and integer processing speed ratios, expressed in Equation (4.11), indicate that floating point intensive processes will be more detrimentally affected than integer intensive processes if a transition is made from CPU B to CPU C. Although only slightly, the floating point performance drops off more than the integer performance. Therefore, processes that require more floating point processing will require more CPU time than integer intensive processes.

The real C1 and C2 perform a considerable quantity of floating point math. On CPU C, the real C1 and real C2 processes require more CPU time than the integer intensive duplicate C1 and C2 processes. For the most part, the integer duplicate C1 and C2 processes acquire CPU time by performing integer math. They require only a small amount of floating point processing to operate. Thus, the real C1 and C2 processes require more CPU ticks than the integer intensive duplicate C1 and C2 processes. In terms of CPU consumption, the floating point intensive C1 and C2 duplicates are better representations of the real C1 and C2.

Conversely, the real GNU Emacs process requires less CPU time than the integer intensive duplicate GNU Emacs. This has to do with the proportion of floating point to integer operations. The integer intensive duplicate GNU Emacs performs a small quantity of floating point math as it executes. The real GNU Emacs process performs string
operations as it edits files. String operations require little, if any, floating point processing. Since the integer duplicate does perform some floating point, it requires slightly more CPU time than the real. The floating point intensive duplicate GNU Emacs process requires even more CPU time than the real.

4.6.3.2 The Rate of Progress Problem

In Figure 36, Figure 37, and Figure 38, the real processes consistently progress at a faster rate than do the integer or floating point intensive duplicates. As discussed in Section 4.6.1, the rate differences are due to the pause values which were determined on CPU B. Because the real and duplicate workloads perform very little screen I/O, screen I/O is not a factor in the rate differences.

The pause values are responsible for the slower rate of the duplicate workload. The pause values are particular to CPU B. On CPU C, the pause values derived from CPU B are used to keep the duplicate processes from progressing at times when the real processes might be able to proceed. Thus, a reduced rate of progress is expected from the duplicate processes.

4.6.4 Processing Speed - From a Slower CPU to a Faster CPU

In this section we examine how the Quantities of Work-Based duplicates created on a slower CPU perform when executed on a faster CPU. CPUs A and B, as identified in Section 4.6.2, were used for these tests. Also within this section, we address the two major problems with our current implementation: the floating point and integer proportion problem, and the rate of progress problem. As will be shown, the floating point and integer proportion problem is much more of an issue with these two CPUs than it was in our previous example.
We tested the Quantities of Work-based approach by monitoring the real workload on CPU B, an HP 400t workstation equipped with a 50 MHz Motorola 68030 and a 50 MHz Motorola 68882 floating point co-processor. We played back the Quantities of Work-based duplicate workload on CPU A, an HP425t equipped with a 25 MHz Motorola 68040. The Motorola 68040 performs floating point processing on chip. It does not need a floating point co-processor. As shown in Table 22, the HP 400t is faster than the HP 370 in floating point and integer calculations.

To make comparisons between the real workload and the work-based duplicates, we also ran the real workload on the faster processor, CPU A. We provide User-Mode CPU consumption graphs for the real, the integer duplicate, and the floating point intensive duplicate C1, C2, and GNU Emacs processes in Figure 39, Figure 40, and Figure 41 respectively.

4.6.4.1 The Floating Point and Integer Proportion Problem

Equation (4.10) expresses the floating point and integer performance ratios of CPU A and CPU B. CPU A is superior to CPU B in integer performance but even more superior in floating point performance. According to Equation (4.10), if processes are moved from CPU B to CPU A, then floating point intensive processes will experience more of a speed up, i.e. require less CPU time, than integer intensive processes. Thus, we expect the floating point duplicates to require fewer CPU ticks than the integer intensive duplicates. The figures corroborate our expectations.

The integer and floating point differences of the real and duplicate C1 processes can be observed in Figure 39. The integer intensive duplicate C1 requires more CPU ticks than the real C1 process. The real C1 process is floating point intensive. Of the three, the floating point intensive duplicate C1 requires the least CPU time.
In Figure 40, the differences in consumed User-Mode CPU time for the real and duplicate C2 processes are evident. Once again, the integer intensive duplicate C2 consumes more CPU ticks than the floating point intensive real C2. As expected, the floating point intensive duplicate requires fewer CPU ticks than the other two.

In Figure 41, the real GNU Emacs process is shown to require more CPU time than the integer and floating point duplicates. The integer intensive duplicate GNU Emacs requires a small amount of floating point processing to operate. The real GNU Emacs processes executes string manipulations, and therefore, has little, if any, floating point processing to perform. Thus, the integer duplicate requires fewer CPU ticks on CPU A than the real GNU Emacs. Once again, the floating point intensive duplicate requires the least CPU activity.

![CPU Consumption: Real C1 vs. Duplicate C1s (CPU A).](image)
Figure 40  CPU Consumption: Real C2 vs. Duplicate C2s (CPU A).

Figure 41  CPU Consumption: Real Emacs vs. Duplicate Emacs (CPU A).
4.6.4.2 The Rate of Progress Problem

In the above figures, the real processes consistently progress at a faster rate than do the integer or floating point intensive duplicates. Screen I/O is not a factor with this workload. The real workload performs very little screen I/O and the duplicates also perform very little.

As noted in Section 4.6.1, the rate differences are due to the pause values which were determined on CPU B. The pause values are particular to CPU B. On CPU A, the pause values keep the duplicate processes from progressing at times when the real processes might proceed. Thus, a reduced rate of progress is expected from the duplicate processes.

4.6.5 Differing CPU Platforms: Conclusions

In this section we discussed the Quantities of Work-based approach and how it can be used to approximate the consumption of CPU time on different CPUs. Our time-based approach is limited to one CPU because the time to execute instructions can be different on different CPUs. The Quantities of Work-based approach represents work as instructions to be executed. On different CPUs, the same instructions are always executed if the instruction sets are the same. The work to be done remains constant. Unfortunately, the current implementation has two faults.

First, the rate of progress is governed by a pause value that is particular to one CPU and screen I/O is not modeled by our duplicates. On a different CPU, the pause values prevent the duplicates from progressing at times when the real processes might proceed. By not emulating screen I/O, our duplicates cannot properly emulate the rates of real processes that perform lots of screen I/O.

The second fault involves the floating point and integer processing speeds of the CPUs. Duplicates that represent work as integer instructions cannot approximate the floating point
CPU consumption of real processes. Currently, our tools provide no monitoring means for determining the floating point and integer processing ratios of real processes. Thus, we cannot expect duplicate processes to approximate the floating point and integer processing proportions of real processes.

4.7 Chapter Conclusions

This chapter has provided an initial assessment of the Workload Generator's capabilities. The tested workload detailed within the chapter provided us with an opportunity to refine the techniques and routines used to emulate the real processes. Synchronization patterns, resource demands, multiple client testing and other aspects were all explored with the help of the tested workload. This chapter has also been used to identify our means of evaluating the duplicate processes, with respect to the real processes.

The conclusion of this chapter also marks the end of our testing phase. The following chapter discusses the final experiment. It is the final experiment that will determine the degree of our success with the Workload Generator. Refinements were permissible when working with the tested workloads, not so with the experimental workload. Without making adjustments to the Workload Generator, the system will be put to the test by examining the CAD/CAM package, I-DEAS[50].
Chapter V

The Experiment Workload

In Chapter IV, the Workload Generator's ability to create a duplicate of one of our tested workloads was evaluated. The purpose of our tested workloads was to help in the design and tuning phases. The tested workload presented in Chapter IV tested all relevant aspects of the Workload Generator. Because the tested workloads were created by ourselves, a further confirmation with a separate workload is warranted. We refer to this separate workload as our experiment workload. In this chapter we describe an experiment workload, provide data illustrating how well our duplicate processes emulate the experiment workload, and detail how much easier the duplicate workload is to use than the experiment workload.

5.1 The Experiment Workload

We chose to use I-DEAS V from the Structural Dynamics Research Corporation, or SDRC, as our experiment workload. I-DEAS is a CAD/CAM package providing geometric modeling, solid modeling, drafting, finite element modeling, and other functions. I-DEAS also provides a playback mechanism allowing a user to record his/her actions and subsequently play them back. A controlling playback script was created that incorporated
looping constructs and separated various I-DEAS instructions with sleeping commands to emulate user thinking times.

The controlling playback script was designed to allow us to execute precisely the same I-DEAS instructions over and over again, thereby facilitating the testing of the real workload. Unfortunately, the playback script did not work out as well as we had hoped. After considerable testing time had been expended, the data began to suggest a problem. The real I-DEAS workload did not behave consistently across iterations of the playback script's loops. The data segment grew in size as the script continued its execution. Apparently, the I-DEAS workload maintains state, or history, information as it progresses. The amount of state information to maintain increases with each additional statement that is executed. The result is an ever increasing data segment. Therefore, we abandoned the looping aspects of the playback script and opted for a more labor intensive testing method. We completely stopped and re-started the I-DEAS software between tests. We had to spend many hours starting and stopping I-DEAS, another reason why working with real workloads is so difficult. In section 5.7 we compare the ease of use of the duplicate workload with that of the real workload.

Our I-DEAS workload, i.e. the real workload, is comprised of three processes, X Windows, Geomod.exe, and Envdd8.exe. The Envdd8.exe executable is responsible for driving the windowing display functions. Envdd8.exe, or ENV, consists of approximately 2 MBs of text and 11.3 MBs of data. Geomod.exe, or GEO, is the geometric modeling process. It executes with a large virtual memory demand of 11 MBs of text and 15.5 MBs of data. X Windows, or X, requires 1.5 MBs of text and 4.7 MBs of data virtual space. The virtual memory demands of these three processes are extremely large. SDRC recommends
that I-DEAS be run on Hewlett-Packard workstations with at least 16 MBs of main memory.

We found that our I-DEAS workload, once started, operated well with less than 16 MBs. However, we would never recommend any amount smaller than 16 MBs. The processes comprising the I-DEAS workload are very large. They require an enormous amount of time to start with 16 MBs of main memory. The start-up time increases dramatically as memory is reduced. Additional start up details are provided in Section 5.7.

In the next sections of this chapter we provide data for the real and duplicate workloads. First, single machine CPU, I/O, and paging comparisons are made between the real and duplicate workloads. We then examine how the real and duplicate workloads expand to multiple client configurations and how the file server’s resources, disks and CPU, are affected. Response times for compiling a C program are also provided.

5.2 CPU Activity Comparisons

As can be seen in Figure 42, Figure 43, and Figure 44, the duplicate executables created by the Workload Generator are indeed capable of emulating the CPU demands of their real counterparts. A striking feature can be seen in both the real and duplicate ENV User-Mode CPU consumption graphs shown in Figure 43. For a considerable period the ENV process consumes CPU cycles at a relatively slow rate until suddenly the rate increases drastically. The skyrocketing of the CPU consumption rate corresponds to the issuing of object shading commands. The real I-DEAS workload manipulates objects, such as cones, rectangles, and spheres. These objects are intersected with one another, regions of objects are cut out, and the objects are moved around frequently. Those activities require a low amount of CPU activity on the part of the ENV process. On the other hand, the
shading of objects requires a tremendous amount of CPU activity. Two shading operations were performed by the real workload and emulated by the duplicate.

Applying Equation (4.7) to the CPU data for the three pairs of real and duplicate GEO, ENV, and X processes, respectively yields accuracy values of 94.2%, 98.6%, and 95.2% for values of N=1300, 6000, and 2000. Even the peculiar CPU spikes of the ENV process were emulated well.

For comparison purposes, we ran the real workload five different times and calculated all ten possible accuracy values. For the three real processes, GEO, ENV and X, the accuracy value means from the comparisons were calculated to be 96.58%, 97.9%, and 96.55% respectively. The respective standard deviations are 1.7%, 1.2%, and 1.9%. Thus, our duplicate workload shows about as much difference from the real workload as does the real workload from other runs of the real workload.

![CPU Consumption](image_url)

**Figure 42** CPU Consumption for Real and Duplicate GEO Processes.
Figure 43  CPU Consumption for Real and Duplicate ENV Processes.

Figure 44  CPU Consumption for Real and Duplicate X Processes.
5.3 File I/O Comparisons

In this section we provide file I/O comparisons between the real and duplicate processes. As can be seen in Table 23, both the ENV and GEO processes perform a considerable amount of reading and writing. X Windows, on the other hand, does very little reading and no writing whatsoever. As with the tested workload in Chapter IV, the duplicate processes for this workload were capable of matching the total file I/O behavior of the real processes with near perfection.

Of course, reading and writing the proper quantities is only a partial responsibility of the duplicates. They must perform the I/O operations at the proper moments as well. Recall that a duplicate process emulates a real process by following the operational directions within the profile records. Some profile records contain file reading and writing instructions. The reading and writing action of the real process can be paired with the

<table>
<thead>
<tr>
<th></th>
<th>Read Count</th>
<th>Read Bytes</th>
<th>Write Count</th>
<th>Write Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Real ENV</strong></td>
<td>144</td>
<td>589,824</td>
<td>141</td>
<td>569,346</td>
</tr>
<tr>
<td><strong>Duplicate ENV</strong></td>
<td>144</td>
<td>589,824</td>
<td>141</td>
<td>569,346</td>
</tr>
<tr>
<td><strong>Real GEO</strong></td>
<td>582</td>
<td>2,383,872</td>
<td>205</td>
<td>831,490</td>
</tr>
<tr>
<td><strong>Duplicate GEO</strong></td>
<td>582</td>
<td>2,383,872</td>
<td>205</td>
<td>831,490</td>
</tr>
<tr>
<td><strong>Real X Windows</strong></td>
<td>23</td>
<td>304</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Duplicate X Windows</strong></td>
<td>23</td>
<td>304</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
cumulative User-Mode CPU ticks as a means of estimating "when" the reading and writing took place. We can evaluate the duplicate executables by comparing "when", in terms of cumulative User-Mode CPU time, they execute their read/write actions with the "when" times of the real processes.

In Table 24 we provide data from 234 distinct reading and writing operations performed by the three duplicate and real processes. The table contains data about the differences in cumulative User-Mode CPU ticks between the duplicate processes and the real processes. For instance, if the real X process performed a read at User-Mode CPU time 236, and the duplicate performed the read at time 242, then the difference in ticks is 242 - 236, or 6 ticks. Recall that there are 100 ticks in a second.

The Table 24 data indicates that the differences are quite small. Most of the differences are between 0 and 5 ticks. The few larger differences can be attributed to our emulation technique. Occasionally, a duplicate process acquires all of the User-Mode CPU time for a profile record before beginning the file I/O. When that occurs, the duplicate waits to perform the I/O until it begins operating on the next profile record.

Table 24  Tick Differences for Reading and Writing Operations.

<table>
<thead>
<tr>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>234</td>
<td>5.2</td>
<td>5</td>
<td>3.5</td>
<td>0</td>
<td>23</td>
</tr>
</tbody>
</table>
5.4 Main Memory Size Changes and Page Faults

A stated goal of our thesis was to create duplicate workloads capable of responding to main memory changes as the real workloads does. By changing the main memory of a workstation, one can expect changes in the quantity of page faults. In this section, we explore how the real and duplicate workloads respond to several levels of main memory, 16 MB, 14 MB, 12 MB, and 10 MB.

The page fault statistics in Table 25 have been derived from the aggregate page faults made by the real and duplicate processes as the workloads ran from one identifiable point to some other identifiable point. The identifiable points can be marked by particular User-Mode CPU values for the processes, by synchronization points, or by any other events that can be readily identified. We used User-Mode CPU times as our identifiable points.

In Section 4.4.2.1, we defined the work done by a workload between two identifiable points as an Interval of Activity. Recall that it is necessary to identify intervals of activity to insure that we collect page fault activity from the same portion of the workload during the many runs of the workload. We arbitrarily chose a 2 to 3 minute interval of activity from the I-DEAS workload. The work required approximately 2 minutes to complete on 16 MB clients and approximately 3 minutes on 12 MB clients. With 10 MB clients, the time exceeded 10 minutes for the real workload and 6 minutes for the duplicate.

The data indicates that the duplicate processes fail to emulate the real processes at the 10 MB level. We consider 10 MBs to be the breaking point for this workload. The real workload pages and swaps almost continuously with just 10 MBs of memory and it is not likely that anyone would wish to run a workload under those thrashing conditions.

Although it is unlikely that anyone would want to operate this workload with only 10 MBs of memory, we can still explain the reasons why the duplicate workload experiences
far fewer page faults than the real workload. Each duplicate process attempts to touch pages as it emulates a real process. If a reference bit is set, then the duplicate references a page. The algorithm was designed so the duplicate process would reference the page only once before moving on to the next profile record. The real process may have referenced the page more than once during that interval of time. Under thrashing conditions, it is likely that the second, third, or any later reference to the page will result in page faults. So, the repeated referencing of pages by the real processes can result in additional paging activity when the system is thrashing.

<table>
<thead>
<tr>
<th>Client Memory</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 MB</td>
<td>Real</td>
<td>5</td>
<td>203</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>Duplicate</td>
<td>5</td>
<td>221</td>
<td>190</td>
</tr>
<tr>
<td>14 MB</td>
<td>Real</td>
<td>7</td>
<td>339</td>
<td>339</td>
</tr>
<tr>
<td></td>
<td>Duplicate</td>
<td>5</td>
<td>386</td>
<td>391</td>
</tr>
<tr>
<td>12 MB</td>
<td>Real</td>
<td>5</td>
<td>790</td>
<td>754</td>
</tr>
<tr>
<td></td>
<td>Duplicate</td>
<td>7</td>
<td>818</td>
<td>767</td>
</tr>
<tr>
<td>10 MB</td>
<td>Real</td>
<td>5</td>
<td>4488</td>
<td>4321</td>
</tr>
<tr>
<td></td>
<td>Duplicate</td>
<td>5</td>
<td>2593</td>
<td>2667</td>
</tr>
</tbody>
</table>

Table 25  Workload Paging Statistics (Pages)
The page fault statistics provide the reader with an understanding of just one region, or interval of activity, of the workload. The paging activity during other intervals is different. These statistics are not an attempt to convince the reader that the duplicate workload emulates the paging activity over all intervals. This data must be used in conjunction with other data, such as disk I/O data, to understand the paging activity properly.

In section 5.5.2, we provide disk I/O data from tests with multiple 10 MB clients. The 10 MB clients require paging, file reading, and file writing activities as they execute the real and duplicate workloads. If the paging behavior of the duplicates was not a close approximation of the real workload's paging behavior, then the disk I/O statistics would reflect the differences. Fortunately, the data shows only small differences.

5.5 Multiple Client Workload Emulations

As stated previously, one of the goals of the Workload Generator project was to handle single machine and multiple machine emulations. We designed the Workload Generator so it could monitor real workloads on a single of a file server's clients, and then generate duplicates capable of being executed on any number of the file server's clients. Increasing the number of clients from 0 to N has a magnification effect. Errors that cannot be noticed on a single machine get compounded and become more apparent in multiple machine testing.

The real and duplicate I-DEAS workloads were executed on a cluster of 14 MB, 12 MB, and 10 MB clients. The file server's CPU utilization, disk usage, and client response times are presented within this section. Due to swap space limitations, we were only able to run up to 6 diskless clients with I-DEAS. Coincidentally, our I-DEAS licensing agreement also had a limitation of 6 simultaneous users.
5.5.1 File Server CPU Utilization

Table 26 and Table 27 contain file server busy percentages for a cluster of 12 MB diskless clients executing the real and duplicate workloads. All samples are 5 seconds in duration. The means and medians are similar for the real and duplicate workloads. In all cases, the data shows that the file server's CPU is not the bottleneck. The file server's mean CPU utilization remains below 50%. With 99% confidence we cannot reject the hypothesis that the 12 MB mean server CPU busy percentages for the real and duplicate workloads are from the same distribution.

Table 26  Real Workload Server CPU Busy Percentages on 12 MB clients

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>160</td>
<td>42.8</td>
<td>43.4</td>
<td>9.0</td>
<td>6.8</td>
<td>65.0</td>
</tr>
<tr>
<td>4</td>
<td>160</td>
<td>32.9</td>
<td>39.3</td>
<td>15.7</td>
<td>3.8</td>
<td>69.0</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>21.6</td>
<td>13.8</td>
<td>17.4</td>
<td>3.2</td>
<td>62.4</td>
</tr>
</tbody>
</table>

Table 27  Duplicate Workload Server CPU Busy Percentages on 12 MB clients

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>160</td>
<td>43.3</td>
<td>43.0</td>
<td>7.1</td>
<td>10.4</td>
<td>65.6</td>
</tr>
<tr>
<td>4</td>
<td>160</td>
<td>35.2</td>
<td>39.3</td>
<td>11.9</td>
<td>5.6</td>
<td>59.4</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>18.4</td>
<td>12.3</td>
<td>14.4</td>
<td>3.4</td>
<td>49.8</td>
</tr>
</tbody>
</table>
5.5.2 Disk I/O

Table 28 contains the disk reading statistics for the real workload and Table 29 contains the duplicate’s disk reading values. The samples are five seconds in duration. For both the real and duplicate workloads, the means increase as the number of clients increases, but the standard deviation’s decrease. This is typical for multiple client testing. As previously stated, we expect high variances with just a few clients in operation and lower variances as the number of clients increases. Also, with two clients operating there are times when no disk I/O occurs in a given sample. As a result we observe large differences between the mean and the median values. With 99% confidence we cannot reject the hypothesis that the

Table 28 Real Workload Disk Reading Statistics on 12 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>150</td>
<td>730.1</td>
<td>729.6</td>
<td>154.0</td>
<td>151.6</td>
<td>1,115.6</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>520.1</td>
<td>588.8</td>
<td>295.9</td>
<td>0</td>
<td>1,118.7</td>
</tr>
<tr>
<td>2</td>
<td>151</td>
<td>279.1</td>
<td>94.3</td>
<td>329.3</td>
<td>0</td>
<td>1,203.2</td>
</tr>
</tbody>
</table>

Table 29 Duplicate Workload Disk Reading Statistics on 12 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>150</td>
<td>745.7</td>
<td>760.8</td>
<td>238.7</td>
<td>97.3</td>
<td>1,410.0</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>553.5</td>
<td>557.1</td>
<td>293.4</td>
<td>28.7</td>
<td>1,226.3</td>
</tr>
<tr>
<td>2</td>
<td>151</td>
<td>283.6</td>
<td>94.2</td>
<td>349.4</td>
<td>0</td>
<td>1,134.1</td>
</tr>
</tbody>
</table>
12 MB mean disk reading statistics for the real and duplicate workloads are from the same distribution.

Table 30 and Table 31 hold the disk writing statistics for the real and duplicate workloads. Across the range of 2 to 6 clients, the duplicate workload’s disk writing means and medians agree with the real workload’s values. With 99% confidence we cannot reject the hypothesis that the 12 MB mean disk writing statistics for the real and duplicate workloads, with either two or four active clients, are from the same distribution. With six active clients, we have less than 95% confidence that the means are from the same distribution.

Table 30  Real Workload Disk Writing Statistics on 12 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>150</td>
<td>712.2</td>
<td>703.5</td>
<td>192.6</td>
<td>318.5</td>
<td>1,298.4</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>518.6</td>
<td>534.5</td>
<td>315.7</td>
<td>0</td>
<td>1,359.8</td>
</tr>
<tr>
<td>2</td>
<td>151</td>
<td>222.1</td>
<td>73.7</td>
<td>296.7</td>
<td>0</td>
<td>1,343.5</td>
</tr>
</tbody>
</table>

Table 31  Duplicate Workload Disk Writing Statistics on 12 MB clients (KiloBytes)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Samples</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>150</td>
<td>623.9</td>
<td>618.4</td>
<td>153.5</td>
<td>69.6</td>
<td>1,093.6</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>508.4</td>
<td>540.7</td>
<td>267.3</td>
<td>28.7</td>
<td>1,105.9</td>
</tr>
<tr>
<td>2</td>
<td>151</td>
<td>183.2</td>
<td>60.4</td>
<td>234.0</td>
<td>0</td>
<td>880.6</td>
</tr>
</tbody>
</table>
In Figure 45, the total disk reading and writing statistics are graphed. We see from the figure that the disk subsystem has not yet reached its peak level of performance. In a five second period, the disk subsystem has a maximum level of performance between 1,400 and 1,600 kilobytes of reading and writing. The level was observed with the tested workload in Chapter IV.

![Graph showing disk I/O statistics](image)

Figure 45 Total Reading plus Writing Disk I/O: 12 MB Active Clients.
5.5.3 Response Time and Main Memory Variations

In this section we show how response time changes for the real and duplicate workloads as the quantity of client main memory changes. The response times reported in this chapter are wall clock compile times for a 4 Kilobyte C program. As stated in the previous chapter, response time variations are a good means for evaluating the levels of performance. Response times represent the system from the user’s perspective.

We present response time data for clusters of 14 MB, 12 MB, and 10 MB diskless clients. The response times are measured on one client, which is otherwise inactive, while the remaining clients actively emulate users. Table 32 and Table 33 contain the real and duplicate 14 MB client data. Figure 46 is a graph of the mean values. With 99% confidence we cannot reject the hypothesis that the 14 MB mean response times for the real and duplicate workloads are from the same distribution.

Although the variances are quite large with six active clients, the means and medians indicate that 14 MB clients can accommodate this workload. The workload shifts between making heavy demands on the file server and making low file service demands which results in the large variances. Large response time variances indicate that the system will provide erratic service to the users.
Table 32  Real Workload Response Times on 14 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>32</td>
<td>10.3</td>
<td>6.95</td>
<td>9.6</td>
<td>5.6</td>
<td>52.0</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>6.9</td>
<td>6.1</td>
<td>1.95</td>
<td>5.5</td>
<td>13.7</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>6.7</td>
<td>6.0</td>
<td>2.4</td>
<td>5.4</td>
<td>15.6</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 33  Duplicate Workload Response Times on 14 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>32</td>
<td>10.95</td>
<td>8.4</td>
<td>6.5</td>
<td>5.9</td>
<td>34.5</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>8.1</td>
<td>7.35</td>
<td>2.3</td>
<td>6.0</td>
<td>15.0</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>6.0</td>
<td>5.85</td>
<td>0.61</td>
<td>5.4</td>
<td>7.9</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Figure 46  Compile Response Times: 14 MB Clients
Table 34  Real Workload Response Times on 12 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>48</td>
<td>16.8</td>
<td>7.6</td>
<td>15.2</td>
<td>5.2</td>
<td>61.1</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>13.7</td>
<td>10.1</td>
<td>9.9</td>
<td>5.0</td>
<td>53.2</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>8.9</td>
<td>8.1</td>
<td>3.15</td>
<td>5.3</td>
<td>17.3</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 35  Duplicate Workload Response Times on 12 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>48</td>
<td>16.8</td>
<td>12.5</td>
<td>11.1</td>
<td>5.6</td>
<td>43.5</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>13.1</td>
<td>10.9</td>
<td>6.8</td>
<td>5.7</td>
<td>31.6</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>9.2</td>
<td>8.9</td>
<td>2.1</td>
<td>6.0</td>
<td>15.2</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Figure 47  Compile Response Times: 12 MB Clients
Table 36  Real Workload Response Times on 10 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>42</td>
<td>24.7</td>
<td>23.65</td>
<td>12.1</td>
<td>6.3</td>
<td>55.4</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>17.8</td>
<td>16.9</td>
<td>9.5</td>
<td>5.9</td>
<td>56.6</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>9.4</td>
<td>8.4</td>
<td>3.6</td>
<td>5.4</td>
<td>17.2</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 37  Duplicate Workload Response Times on 10 MB Workstations (Seconds)

<table>
<thead>
<tr>
<th>Clients</th>
<th>Timings</th>
<th>Mean</th>
<th>Median</th>
<th>St. Dev.</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>42</td>
<td>25.8</td>
<td>23.7</td>
<td>16.1</td>
<td>5.8</td>
<td>69.6</td>
</tr>
<tr>
<td>4</td>
<td>48</td>
<td>16.9</td>
<td>17.25</td>
<td>7.0</td>
<td>5.9</td>
<td>35.1</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>9.8</td>
<td>9.2</td>
<td>3.1</td>
<td>5.7</td>
<td>24.6</td>
</tr>
<tr>
<td>0</td>
<td>25</td>
<td>5.48</td>
<td>5.3</td>
<td>0.49</td>
<td>5.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Figure 48  Compile Response Times: 10 MB Clients
Table 34 and Table 35 contain the real and duplicate 12 MB client response time data. With 99% confidence we cannot reject the hypothesis that the 12 MB mean response times for the real and duplicate workloads are from the same distribution. The 12 MB client response time means and standard deviations increase as the number of clients increases to six. A close examination of the data shows that with six clients in operation, the real workload’s median response time of 7.6 seconds does not agree with the mean time of 16.8 seconds. Because the 7.6 second median is less than both the four client and two client median response times, we suspect that the six clients were synchronized in such a way that they simultaneously experienced periods of high activity and periods of low activity.

The real and duplicate 10 MB client response time data are contained within Table 36 and Table 37. The means, medians, and standard deviations of the real and duplicate processes are in agreement. The question to ask is, should the duplicate and real workloads show such agreement in response time measurements when the real and duplicate processes page at different levels?

As shown in Table 25, the real workload pages much more than the duplicate workload. The different paging levels are the result of the real workload performing multiple references to pages at times when the duplicate references the pages just once. Thus, while thrashing at the 10 MB memory level, the real workload pages more than the duplicate.

The question remains, why should the duplicate and real workloads show such agreement in response time measurements when they clearly page at different rates? Because both the real and duplicate workloads have placed the 10 MB system in a thrashing state, the clients page and swap at as fast a rate as they can. The real and duplicate workloads force the system to page at that same, maximum rate. Although the real
workload requires more paging activity in an interval of activity, it cannot proceed any faster than the maximum rate. The real workload requires more elapsed time to complete the interval of activity because the maximum paging rate remains fixed. Hence, the load on the file server and the response time measures are nearly identical for both workloads.

5.6 Differing CPU Platforms

One of the stated goals of the Workload Generator project was to produce duplicate workloads capable of being utilized on different CPU platforms. In particular, we aspired to monitor a real workload on one CPU, and play back a duplicate of that workload on a second CPU, without having any knowledge of how the real workload would operate on the second CPU. Since the two CPUs cannot be assumed to operate at the same speeds, our reliance upon User-Mode CPU time to approximate how much work to perform had to be replaced with the Quantities of Work-based approach. See Sections 3.5 and 4.6 for more details concerning the Quantities of Work-based approach.

In this section we review the two main problems with our approach. The first involves the proportion of floating point and integer calculations, the second involves the rate at which the duplicates progress. Also in this section, we provide results detailing the effectiveness of our technique when moving from a slower CPU to a faster CPU.

5.6.1 Processing Speed - From a Slower CPU to a Faster CPU

In this section we examine how duplicates created on a slower CPU perform when executed on a faster CPU. We tested the Quantities of Work-based approach by monitoring the real workload on CPU B, an HP 400t workstation equipped with a 50 MHz Motorola 68030 and a 50 MHz Motorola 68882 floating point co-processor. We played back the Quantities of Work-based duplicate workload on CPU A, an HP 425t equipped with a 25 MHz Motorola 68040.
As the data in Table 22 in Chapter IV indicates, CPU A, the HP 425t, is faster than CPU B, the HP 400t, in both floating point and integer calculations. However, the speed differences are not proportionate. CPU A’s floating point speed up over CPU B is much larger than CPU A’s integer speed up. Therefore, we expect floating point intensive processes to be affected more by the change in CPU than integer intensive processes.

We ran the real workload on the faster CPU A, i.e. the HP 425t, so comparisons between the real workload and two work-based duplicates, integer intensive and floating point intensive, could be made. We provide User-Mode CPU consumption graphs for the real, integer intensive duplicate, and floating point intensive duplicate ENV, GEO, and X processes in Figure 49, Figure 50, and Figure 51 respectively.

5.6.1.1 The Floating Point and Integer Proportion Problem

We provide data for the integer and floating point intensive duplicates to highlight the differences in CPU A’s and CPU B’s capabilities. The duplicate and real processes show differences in the rates that they progress and differences in the quantities of CPU ticks that they acquire. The latter can be attributed to the changes in the floating point and integer performance ratios expressed in Equation (4.10). In this section we concentrate on the differences in User-Mode CPU times.

The floating point performance of the Motorola 68040, i.e. CPU A, is many times superior to the floating point performance of CPU B’s Motorola 68882. The integer performance difference between the two CPUs is not so dramatic. We discuss potential solutions to the floating point and integer processing speed problem in Chapter VI.

For the ENV and GEO processes, Figure 49 and Figure 50 indicate that the real processes have a higher percentage of floating point instructions than the integer intensive duplicates. Because they have more floating point to perform, the real GEO and ENV
processes require fewer CPU ticks than their integer intensive duplicate counterparts. The floating point intensive duplicates require even less CPU time than the real GEO and ENV processes.

Figure 49   CPU Consumption for Real, Integer and Float ENV Processes (CPU A).
Figure 50  CPU Consumption for Real, Integer, and Float GEO Processes (CPU A).

Figure 51  CPU Consumption for Real and Duplicate X Processes (CPU A).
In Figure 51, the CPU consumption graphs for the real and duplicate X processes are shown. Examining the data closely, we see that the integer intensive duplicate requires fewer CPU clock ticks than its real counterpart. We know that the real X process is an integer intensive process. Since our integer duplicate X performs some floating point, it experiences a slightly greater speed up than the real X process. Of course, the floating point intensive duplicate requires the least CPU time.

5.6.1.2 The Rate of Progress Problem

In addition to having differences in the quantities of consumed CPU ticks, the real and duplicate processes progress at different rates. The duplicate processes, and hence the duplicate workloads, progress at a faster rate than the real processes. In Chapter IV, the opposite was true. The duplicates were slower for the tested workload in Chapter IV. We attributed the relative slowness of the Chapter IV duplicates to pause values. The question is, “why does the duplicate I-DEAS workload proceed faster than the real I-DEAS workload?”

The quicker pace of the duplicate I-DEAS workload and the relative slowness of the real I-DEAS workload can be attributed to screen I/O. The real I-DEAS workload displays many objects and images as it executes. On any CPU, screen I/O slows down the rate that a process may progress. On faster CPUs, the slow down is more noticeable than on slower CPUs. Because we do not yet have a screen I/O monitoring tool, our duplicates were not designed to reproduce the screen I/O of the real processes. Rather, the slow down affects of screen I/O become a part of the pause values that the duplicates use to govern their rates of progress. Because screen I/O has a larger impact on CPU A than it does on CPU B, we expect that the pause values calculated from an execution on CPU B will be too small for an execution on CPU A.
A potential solution to the rate of progress problem caused by screen I/O is discussed in Chapter VI.

5.7 Ease of Use Issues

One claim of our thesis is that the Workload Generator can create duplicate workloads that are easier to use than the real workloads. In this section we estimate the time and effort required to set up and perform experiments with the real and duplicate I-DEAS workloads.

Initially, we spent approximately two days installing I-DEAS on our test cluster of workstations. We experienced difficulties with our version of X Windows and with the HP-UX operating system. After expending considerable time and effort, it was necessary for us to acquire a different version of X Windows. I-DEAS was the first and only application to give us X Windows problems. The operating system troubles were not as dramatic.

One could argue that we would have had to install I-DEAS to build the duplicates anyway, that the I-DEAS installation time should not be considered an ease of use issue. A good point, but not entirely true. We could have monitored a user running I-DEAS on a machine other than one of the machines on our testing cluster.

The duplicate workloads require initial monitoring phases that the real I-DEAS workload does not. The duplicates need to monitor a user running I-DEAS on a normal system and once again on a stressed system. The two phased monitoring approach is required so accurate memory page reference data can be collected. The two phases and subsequent data processing can take between two to three hours to complete.

After installation, we spent two weeks getting familiar with I-DEAS and developing a robust playback script. The playback script included looping constructs, I-DEAS instructions, and sleeping commands to emulate user thinking times. Much of this work
was in vain. As detailed earlier, the looping construct had to be abandoned. We discovered that the memory demands of I-DEAS were not consistent across iterations of the playback script’s loops. The data segment grew in size as the script continued its execution.

We abandoned the looping construct and opted for another, although more labor intensive, testing method. We completely stopped and re-started the I-DEAS workload between tests. To complicate matters, we were unable to automate the starting of I-DEAS. I-DEAS is comprised of a number of processes. Parent processes spawn child process in such a manner that we could not direct input to the child processes.\footnote{For many UNIX processes, it is easy to direct input from a file to the process. One can then automatically send input to a process and not have to type it explicitly.} We were forced to type instructions and operate menus by hand each time we started I-DEAS. Hands-on testing is particularly cumbersome when testing multiple client configurations. Depending on the amount of client memory, the starting of six clients generally took between 20 and 30 minutes of hands-on activity. The starting of four clients averaged about 10 to 12 minutes, and two client starting averaged approximately eight minutes.

To insure the validity of the data, we executed about five separate runs with six clients, four clients, and two clients each. We did this with 14 MB clients, 12 MB clients, and 10 MB clients. A rough calculation indicates that we spent over 11 hours of hands-on time performing the tests. The 11 hours does not include the time required for the tests to run to completion. It only includes the starting and stopping times.

The process of start and restart can be completely automated with the duplicates. Duplicate processes can be started on six different clients by executing one command. In fact, one command can be made to start a series of six, four, or two client tests, one after
the other. In other words, the entire series of tests using the duplicate workloads can be started in the time it takes to type one command.

Of course, the development time to create the command script has not been accounted for yet. I estimate 1 to 3 hours of time, depending on the proficiency of the programmer.

5.8 Conclusions

In this chapter we have shown how the Workload Generator is capable of producing duplicate processes for the I-DEAS software package. Our duplicate processes were very able to emulate the real I-DEAS processes in terms of memory page referencing, file I/O, file server CPU utilization, disk I/O, and response times. The duplicates were successful in emulating the client CPU usage on one CPU, but were less capable of emulating the three I-DEAS processes on a different CPU. Without monitoring the integer to floating point proportion or the screen I/O requirements of the real processes, the duplicate processes are unable to emulate the real processes on different CPUs. However, integer intensive and floating point intensive duplicate processes can be used to identify the range of User-Mode CPU time that the real processes may require on a different CPU.

After reviewing the duplicate’s capabilities, we can now identify how the duplicate workload can be used, instead of the real workload, to make configuration recommendations. In essence, using the duplicate workload as a substitute for the real workload was the main objective of this thesis. The data has shown that our duplicate workload can be used to indicate the following performance characteristics of the I-DEAS workload.

• Client Memory is an issue for the workload. Multiple client performance degrades considerably with less than 14 MBs of client memory.
• The file server's CPU is not a concern. It is usually less than 50% busy.

• The file server's disks become the bottleneck. They are largely responsible for the multiple client performance degradation.

In Chapter VI we outline the main contributions of this research and review areas for possible research.
Chapter VI

Conclusions

In Chapters I and II, we stated the goals of this research. In general, our goal was to
study the problems associated with the automatic generation of synthetic workloads and
provide insight into how they can and cannot be used in experimental studies of
performance. We have shown how duplicate workloads can be used as a replacement for
real workloads for the testing of configuration changes, such as a change to a machine's
main memory, changes to a machine's CPU, or changes to the quantity of diskless file
server clients. In this chapter we discuss the extent to which each of our goals have been
met, and describe improvements and future research associated with the automatic
generation of synthetic workloads.

6.1 Synthetic Workload Overview

An evaluator using real, or natural, workloads can expect to achieve a degree of
accuracy that may not be matched by synthetic workloads. However, the use of real
workloads in experimental performance evaluation studies is cumbersome and time
consuming. Also, the testing of multiple machines, or of configuration changes may not be
possible due to time constraints. Furthermore, the evaluator must learn many intricacies
associated with the real workload to properly examine it. Despite the advantage of higher accuracy, real workloads are seldom used because of their high costs.

Synthetic workloads can be an excellent alternative. A well designed synthetic workload incorporates test control features allowing for the easy testing of multiple machines, and of configuration changes. Synthetic workloads can be hardware and software independent, many real workloads cannot. Ease of use is the main advantage that synthetic workloads have over real workloads. The chief drawback is accuracy.

A synthetic workload can be poorly designed and bear little resemblance to the real workload. Or, with a large investment of time and work in the design stages, a synthetic workload can operate and stress the system similar to the real workload. Well designed synthetic workloads can be very accurate and useful tools. Many synthetic workloads are not accurate because the evaluator was unwilling, or unable, to invest the proper amount of work in the design stages. Constructing synthetic workloads that utilize the memory, the CPU, and the other system resources as a real workload can be an enormous undertaking if done by hand. Reducing the design work and time is one of the objectives of the Workload Generator project.

6.2 Contributions of the Dissertation

The main contributions of this research are listed below.

- Developed monitoring abilities to collect process CPU, input/output, and page faulting information efficiently.

- Developed monitoring abilities to intrusively gather memory page referencing data and created post processing tools to merge data with other, less intrusively collected data.
- Designed and implemented a means for creating synthetic executables based on monitored data from real processes. Synthetic duplicates are similar in size and behavior to the real processes. We have shown how the synthetic duplicates can emulate the real processes in the following tasks:
  - CPU Usage.
  - File and Disk I/O.
  - Main Memory Changes.
  - Multiple Client Testing, include response time measurements.

6.3 Extensions to Our Research

The research extensions identified in this section involve enhancing the Workload Generator's abilities by developing additional monitoring capabilities. We present three potential monitoring improvements as extensions to our research. The first monitoring extension regards memory page references. The second extension has to do with floating point and integer operations. The third extension concerns screen I/O monitoring.

6.3.1 Memory Page References

The key to the Workload Generator's effectiveness is the efficient collection of accurate data. If the data is flawed or incomplete then the generated duplicates cannot behave properly. Our current monitoring techniques are able to efficiently collect CPU and file I/O data with sufficient accuracy. However, our current mechanisms for collecting memory page referencing data are limited. Memory page referencing data must be intrusively gathered and are only an approximation of the true referencing patterns of the processes.
A change to the kernel's memory management routines might allow for a more efficient and accurate collection of page reference data. Our current implementation forces the kernel to act by changing some of the page replacement algorithm's variables. The variable changes place an added stress on the system. A modification to the page replacement algorithm would be more efficient.

6.3.2 Floating Point and Integer Proportions

Our synthetic duplicates do not reflect the floating point and integer proportions of the real workloads. When we execute a duplicate process on a CPU, other than the one on which the real process was monitored, we find that the duplicate and real processes do not consume similar quantities of CPU time. Because different CPUs are likely to have different floating point and integer performance ratios, processes with different floating point to integer instruction ratios will not execute in the same amount of CPU time. Thus, our duplicate processes do not span to different CPUs as well as they could if the floating point and integer proportions were correct.

A monitoring tool capable of ascertaining the floating point and integer proportions of a workload is desired. Such a tool can be built for the Motorola 68000 family of processors[47]. As a program development aid, each 68000 based processor has a Trace bit in its processor status register. When tracing is enabled, an exception handling routine is called after every 68000 instruction. The exception handling routine can include code to count floating point instructions and integer instructions. With proper counts, the proportion of floating point and integer instructions to execute can be incorporated into the code of the synthetic duplicate processes.
6.3.3 Screen I/O Monitoring

We currently have no means for monitoring the screen I/O aspects of a real workload without modifying the screen device driver. Consequently, our synthetic duplicates cannot reflect the screen I/O portions of a real workload. As has been shown, this presents a problem when attempting to play back a synthetic duplicate on a different CPU platform. The new CPU platform can be affected more, or less, by screen I/O than the first platform. The rate with which a real workload progresses is thereby affected by the screen I/O on the new CPU. The real workload may progress faster on the new CPU, or it may progress slower on the new CPU. The synthetic duplicates cannot approximate the rate changes without some rudimentary knowledge of the real workload’s screen I/O demands.

Monitoring the screen I/O demands of the real workload might be possible by augmenting the screen’s device driver program. Line drawing, and other screen I/O operations, are performed by the screen’s device driver program. The first step towards monitoring the screen I/O demands of a workload would involve enhancing the device driver so it maintains detailed data on the requests made to the device driver. That data could then be used by the duplicate processes, allowing the duplicates to emulate the screen I/O demands of the real processes.

6.4 Final Statement

We have shown how a Workload Generator can be designed and built, and how generated workloads can be used to speed the evaluation of clusters of workstations.
Appendix A

Hardware & Software Configurations

Unless otherwise specified the hardware used for the tests were HP 9000/400t series workstations. The 400t workstations have a 50 MHz Motorola 68030 processor, a 50 MHz Motorola 68882 floating point co-processor, and 16MBs of main memory. We identify tests in which other memory quantities were used. All of our tests were run in a diskless configuration.

An HP 9000/375 workstation was used as the file server for all tests. The HP 375 has a 50 MHz Motorola 68030 processor. Three 660 MB SCSI disk drives were attached to the file server.

In Chapters IV and V, tests with different CPU platforms were performed. One of the different CPU platforms was an HP 9000/370 equipped with a 25 MHz Motorola 68030 and 16 MBs of main memory. The HP 370 also has a 25 MHz Motorola 68882 floating point co-processor.

The final CPU platform was an HP 9000/425t workstation. Our HP 425t was equipped with 16 MBs of main memory and a 25 MHz Motorola 68040 processor. The 68040 has a built in floating point co-processor.
The HP 9000/400t and 9000/370 ran the HP-UX 7.0 operating system. The 9000/425t workstation ran HP-UX 7.05. The HP-UX 7.05 and HP-UX 7.0 releases are very similar. The major difference is that HP-UX 7.05 release supports the Motorola 68040 processor but HP-UX 7.0 does not.
Appendix B
Collectible Data Items

The Collector, a Hewlett-Packard software product, can report more than 400 separate data items covering nearly every detail regarding the system's performance. In this appendix we present a portion of the collectible data items which concern processes.

- Process Identification Number
- Process Name
- Creation Time .................................. Time when process was started.
- Death Time...................................... Time when process terminated.
- User-Mode CPU Time ...................... Accumulated User-Mode CPU time.
- System-Mode CPU Time .................. Accumulated System-Mode CPU time.
- Maximum Resident Set Size............... Maximum number of pages in memory.
- Data Resident Set Size..................... Current number of resident data pages.
- Text Resident Set Size .................... Current number of resident text pages.
- Virtual Data Segment Size............ Total number of virtual data segment bytes.
- Virtual Text Segment Size ............. Total number of virtual text segment bytes.
- Page Faults.................................... Number of Page Faults.
• Swap Out Requests ......................... Number of times process is swapped out.
• Swap In Actions ............................... Number of times process is swapped in.
• Received Signals ............................. Number of signals received by the process.
• File Opens .................................... Total number of file opens.
• File Closes .................................... Total number of file closes.
• File Read Count .............................. Total number of read operations.
• File Read Bytes .............................. Total number of read bytes.
• File Write Count ............................. Total number of write operations.
• File Write Bytes .............................. Total number of write bytes.
Bibliography


