Simplex Architecture Performance and Cost

Mike Gagliardi
Theodore Marz
Neal Altman
John Walker

September 2000

TECHNICAL REPORT
CMU/SEI-2000-TR-006
ESC-TR-2000-006

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Simplex Architecture Performance and Cost

CMU/SEI-2000-TR-006
ESC-TR-2000-006

Mike Gagliardi
Theodore Marz
Neal Altman
John Walker

September 2000

Dependable Systems Upgrade Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Norton L Compton

Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Abstract

The Simplex[™] Architecture facilitates the building of dependable and upgradable real-time systems. Before using the technology, potential users want to know more about the costs of adopting the Simplex paradigm compared to the benefits of using it. This paper examines Simplex performance and the costs associated with its use.

[™] Simplex is a trademark of Carnegie Mellon University.

1 Introduction

This paper focuses on the costs and performance issues of the Simplex Architecture.™
For a general introduction to the Simplex architecture, visit http://www.sei.cmu.edu/simplex.

This report addresses the following cost-related concerns:

- How fast is a system based on the Simplex Architecture?
- What resources will a Simplex system use?
- How much will a Simplex system cost to build?

This paper provides information from tests using an existing Simplex prototype. The Simplex Architecture is applicable to a wide range of problems, which the prototype only samples.

This report provides performance data from a system based on Simplex Architecture. To help make this data meaningful, we compare the Simplex systems to comparable systems built using other architectural paradigms that accomplish the same task.

It is important to realize that "comparable" does not mean "functionally identical." The comparisons are always drawn between systems that can reach the minimum specification for the test, but the systems vary in their reliability, efficiency, and adaptability. Features and capabilities are noted for each of the software artifacts tested.

1.1 Test Software

Numerical data for this report were generated by instrumented test programs. An individual test program is the executable object that results from processing a set of source code with a specified set of tools. Once prepared, the program is not altered during a test series.

While we define a test program as a set of source code plus transformation steps, we actually measure an executing program in a larger system that includes the supporting system software: the operating system, firmware, and dynamically bound support libraries. For this report, we chose not to explore how performance changes when the system software changes. Therefore, we held the operating system and hardware constant during the testing period.

1.2 Test Environment: Hardware and System Software

Each test program was exercised on a single system, whose hardware and software was held constant during testing.

™ Simplex is a registered trademark of Carnegie Mellon University.

The test programs were run using a uniprocessor PC-driven real-time control system (see Figure 1). The computer holds an unstable inverted pendulum upright by a feedback control loop.

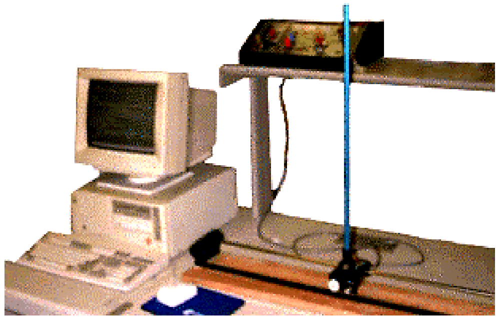

Figure 1: Test Apparatus

The inverted pendulum is on the right, the pendulum power supply is above (behind the pendulum) and the controlling computer on the left.

The test programs were developed under the LynxOS® version 2.4 operating system. The LynxOS® operating system is a POSIX-compliant, real-time operating system (RTOS).

The PC utilizes an Intel 133mHz Pentium processor operating on a 66Mhz bus. The system contains 32 MB of RAM and 256KB of pipeline cache memory. A Data Translation, DT2811PGH data acquisition card is employed for control of the Inverted Pendulum apparatus. The DT2811PGH provides 16 single-ended, Analog-to-Digital (A-D) inputs at 12 bits of resolution. The card also provides two Digital-to-Analog (D-A) outputs also at a resolution of 12 bits. The Inverted Pendulum requires the use of two A-D inputs (one measuring rod angle, one measuring track position), and one D-A output (supplying a DC voltage to the motor). An Alpha Logic Stat! Timer card was selected to provide an accurate time base. The Stat! Timer card is capable of supplying a 32-bit sample timer running at a resolution of 250 nanoseconds.

The pendulum consists of a metal rod, which is attached by a freely swinging hinge to a small cart. The cart is powered and can move horizontally on a track under computer control. The Inverted Pendulum apparatus is a commercially available device consisting of the rod and cart assembly, a 36-inch (91.44 cm) track, and a power supply housing a power op-amp that amplifies the control voltage sent from the PC. Low-pass filters were added to the two A-D input lines to help decrease the amount of signal noise injected into the system by the angle and track potentiometers.

® LynxOS is a registered trademark of LynuxWorks.

2 Data Collection and Analysis Strategy

Data for this report were collected by analysis of the program code and by running instrumented test programs to observe the system during normal (steady-state) system operation.

Two kinds of data were acquired:

1. feature related
2. resource consumption

Feature-related data describe the architectural layout and capabilities of the software under test:

- efficiency
- software engineering features
- dynamic and static upgrade provisions

Resource-consumption data focus on high- level constructs: modules, complete systems, and total resource consumption. The following classes of measurement were taken:

- lines of source code
- time consumed

3 Test Software Summary

For this report, test programs were constructed to examine the overhead associated with interprocess communication. Interprocess communication is a key element of the Simplex Architecture, providing fault isolation and dynamic upgrade capabilities. In a Simplex system, *replacement units* are added and removed during execution without interrupting safe system operation; replacement units are bound dynamically to communications streams and their output sent to other modules. Since the tested programs are real-time applications, the communications used must allow real-time operations. Communication also affects how well a system can be distributed and modified. The test series incrementally adds communications capabilities while holding the other software components as constant as possible.

3.1 Monolithic Test Program

The uniprocessor monolithic test program places all functionality in a single module. Once acquired, data are stored locally and accessed as needed. The monolithic test program provides the minimal functionality necessary to keep the inverted pendulum upright, using a design that should place a light load on the computing hardware. It is intended to establish the performance floor for the test series.

Characteristics of the monolithic test program include

- local memory that provides low latency communication
- fast execution

The uniprocessor monolithic artifact was developed by fusing a Simplex baseline controller with the input/output module portions of the Simplex demonstration software.

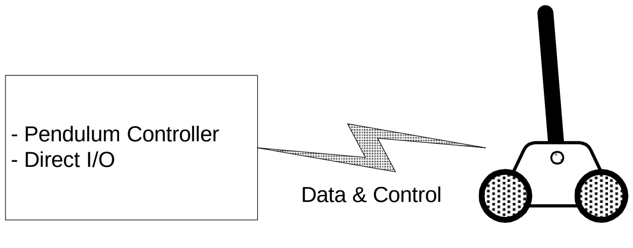

Figure 2: Schematic Diagram of the Monolithic Test Program

3.2 Shared-Memory Test Program

The uniprocessor shared-memory test program introduces modularity and multiple threads of control into the control software. Data input/output and control software were placed in separately executing modules that exchange data through shared memory. This design allows modularization of the software, decreasing the complexity of the individual modules, and permits the modules to execute in separate threads of control. Modularity and separate execution are key enabling technologies for the Simplex Architecture. Shared memory is a straightforward mechanism for interprocess communication with low latency. The shared-memory test program establishes the costs of adding a high-speed but inflexible communications mechanism and modularity.

Characteristics of the shared-memory test program include the following

- Separate threads of execution are possible.
- Data integrity and access control are maintained (when the application program is written correctly).

The shared-memory test program used the baseline controller and input/output module from the Simplex demonstration software. Shared memory replaced the existing communications software.

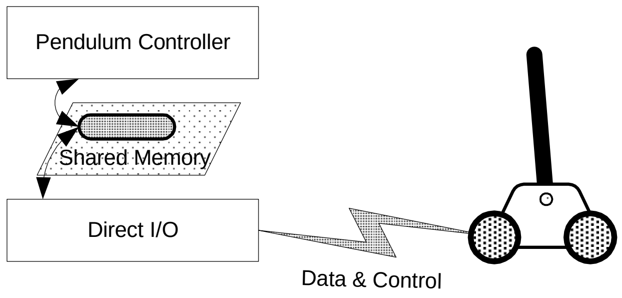

Figure 3: Schematic Diagram of the Shared-Memory Test Program

3.3 POSIX Message Queues Test Program

The uniprocessor POSIX message queues test program alters the interprocess communications to use a queued message system. In the tested implementation, POSIX message queues were used. This artifact examines the costs of using a more flexible communications mechanism as a substitute for shared memory. The Simplex Architecture does not require the use of message-queued communications, but many of the existing Simplex artifacts use this paradigm to implement the system.

Characteristics of the POSIX message queues test program include

- Data synchronization and integrity are provided by communications services.

The POSIX message queues test program reuses the uniprocessor shared-memory artifact's software, but replaces the shared-memory communications with a queued-message communications.

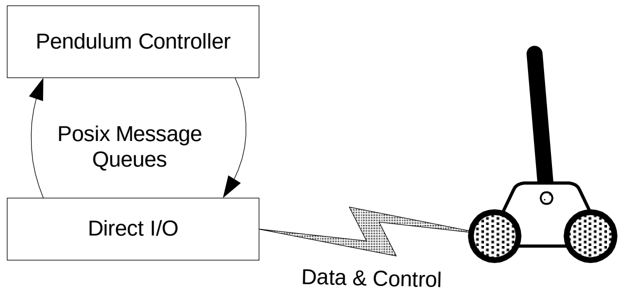

Figure 4: Schematic Diagram of the POSIX Message Queues Test Program

3.4 Data Tagged IPC (Dtag) Test Program

The uniprocessor data tagged IPC[1] test program (Dtag) implements a publish-subscribe communications paradigm. With publish-subscribe techniques, information producers and consumers can communicate without requiring detailed connection information about each other. This facilitates the use of dynamically replaceable components.

Characteristics of the Dtag test program include

- Multicast communications; the sender does not need to know the number of receivers.

The uniprocessor data tagged artifact recasts the uniprocessor message-queued artifact's software to use data-queued message communications.

[1] Interprocess communications.

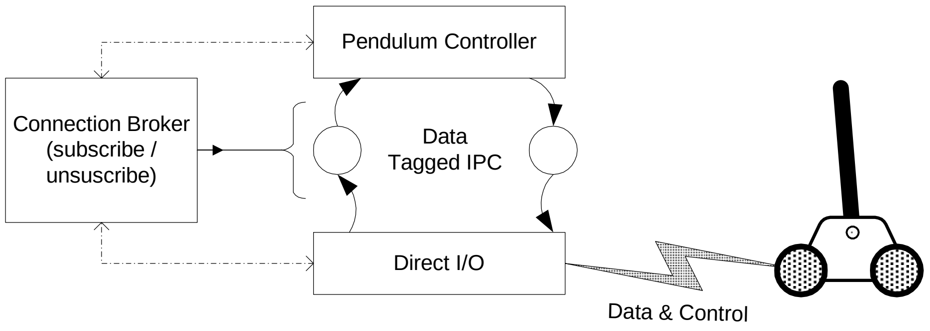

Figure 5: Schematic Diagram of the Data Tagged I/O (Dtag) Test Program

3.5 Simplex Test Program

The uniprocessor analytically redundant artifact contains all the features of a single-CPU system based on the Simplex Architecture. It includes a safety and decision module to automatically detect and clear faults as well as support for replacement controllers.

Characteristics of the Simplex test program include

- functional redundancy
- safety checking and error recovery
- dynamic upgrade capability
- fine grained control over publish/subscribe communications

The uniprocessor Simplex test program is an instrumented test version of the Simplex Singleton demonstration software, with data capture during normal operations. (Dynamic process creation/deletion is not initiated during the observation period.)

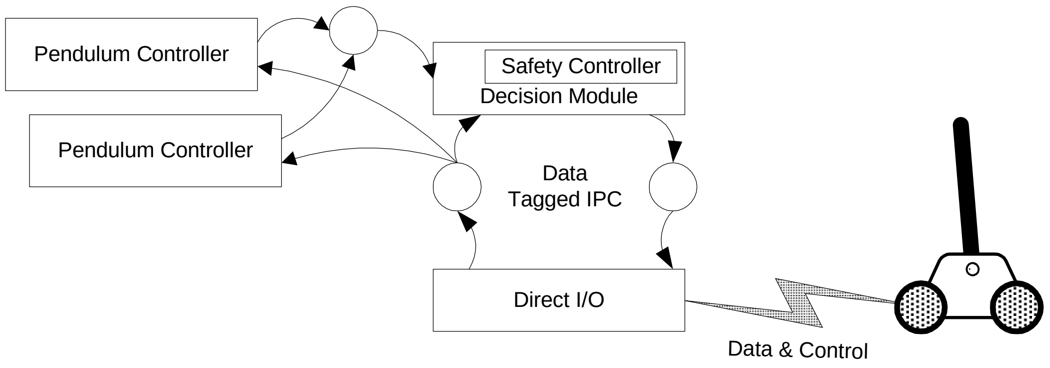

Figure 6: Schematic Diagram of the Simplex Test Program

4 Summary of Results

Table 1: Features Comparison

Feature: *Program:*	***Monolithic***	***Shared Memory***	***POSIX MsgQ***	***Dtag***	***Simplex (1 Controller)***	***Simplex (2 Controllers)***	***Simplex (3 Controllers)***
Application Distributable		Local Only	Yes	Yes	Yes	Yes	Yes
System Data Integrity Services			Yes	Yes	Yes	Yes	Yes
Broadcast Communication				Yes	Yes	Yes	Yes
Safety Checking and Error Recovery					-	Yes	Yes
Dynamic Upgrade Capability					Yes	Yes	Yes

Table 2: Performance Comparison

Observation: *Program:*	***Monolithic***	***Shared Memory***	***POSIX MsgQ***	***Dtag***	***Simplex (1 Controller)***	***Simplex (2 Controllers)***	***Simplex (3 Controllers)***
Total Lines of Source Code[2]	454	839	906	885	2549	2980	3413
Min/Mean/Max Cycle Time (µsec)	178.0 181.6 240.0	451.0 463.7 676.0	523.0 537.5 764.0	681.0 708.9 962.0	1700.0 1939.0 2325.0	2092.0 2709.0 3203.0	2974.0 3337.0 3835.0
Cycle Time Std. Deviation	2.571	7.922	9.293	15.59	145.6	267.1	189.9
# Deadlines Missed	0	0	0	0	0	0	0

2 Simple line count, does not include library files.

5 Detailed Results

The test programs vary in two important ways: communications complexity and degree of redundancy (i.e., fault tolerance). While the test programs add communications complexity incrementally, only the Simplex test program has any built-in fault tolerance. Consequently, the Simplex results are shown separately for non-redundant, two level redundant and three level redundant operation.

The test programs can be ordered by the complexity of the communications showing the overhead cost of communications, leading to the communications support needed to implement Simplex. Figure 7 summarizes the range of execution times for the test programs. Communications complexity increases from left to right in the graph while the remaining elements of the system are held constant. As might be expected, execution time increased as the communications became more sophisticated, about threefold on the tested system. Although communications imposed an overhead cost, the worst-case execution times (top line) did not exceed 150% of the mean value (largest is shared memory, maximum is 145.8% of the mean value).

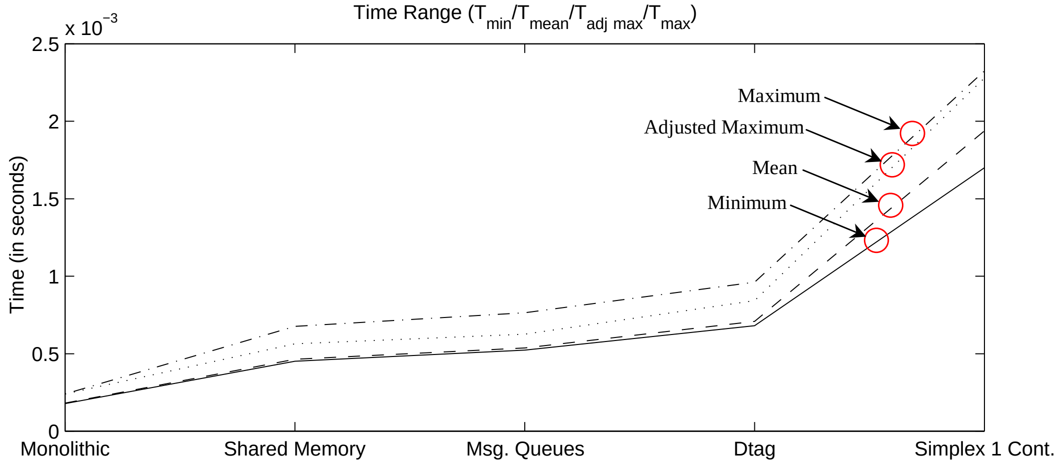

Figure 7: Time Versus Communications Complexity

Communications complexity increases from left to right while the vertical axis shows execution time (in seconds) for a single cycle through the program. The lines show, from bottom to top: minimum, mean, adjusted maximum, and maximum value for all data values for each test series. Data include the timing overhead.

The test runs were started and stopped manually. In many test artifacts, the final observation, taken as the artifact halts, was also the maximum value (see Figure 7 or sections on individual test programs). With this final value removed, worst-case execution times were reduced to a maximum of 135% of the mean value (dotted line, Figure 7).

In Figure 8, data from individual test runs are aggregated into the graphs shown below.

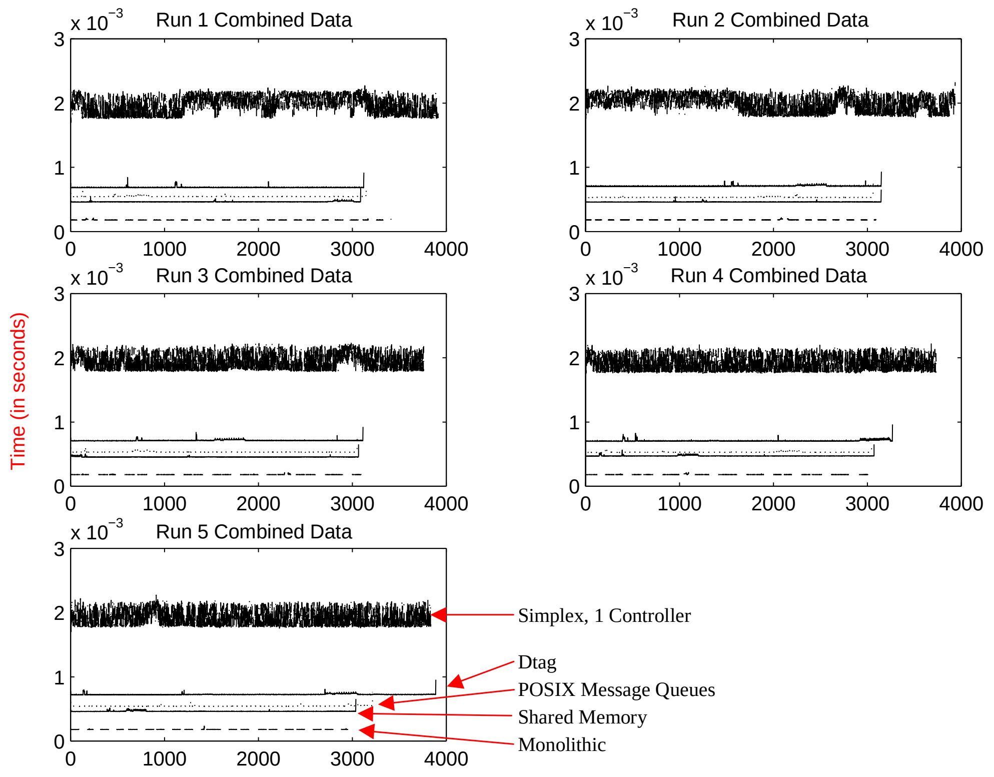

Figure 8: Combined Timing Data, with Five Test Programs per Graph

The horizontal axis shows individual observations, from first to last; the vertical axis is time in seconds. For each graph, from bottom to top, the horizontal lines show monolithic data, shared-memory data, POSIX message queues data, Dtag data, and Simplex data (including one controller).

5.1 Monolithic Test Program

Table 3: Monolithic Test Program: Artifact Summary

Lines of Code	454
Cycle Time	0.02 seconds (50 Hertz)

Table 4: Monolithic Test Program: Time Data Summary

Observation ***Run #:***	***1***	***2***	***3***	***4***	**5**	**Comb.**
Number of Observations	3411	3093	3102	3006	3005	15617
Minimum Cycle Time (µsec)	0.178	0.181	0.178	0.179	0.178	0.178
Mean Cycle Time	0.181	0.183	0.181	0.182	0.180	0.182
Maximum Cycle Time	0.213	0.217	0.215	0.217	0.240	0.240
Cycle Time Std. Dev.	2.35×10^{-6}	2.58×10^{-6}	2.61×10^{-6}	2.37×10^{-6}	1.86×10^{-6}	2.57×10^{-6}
Adjusted Max	0.213	0.217	0.215	0.217	0.240	–
# Deadlines Missed	0	0	0	0	0	0

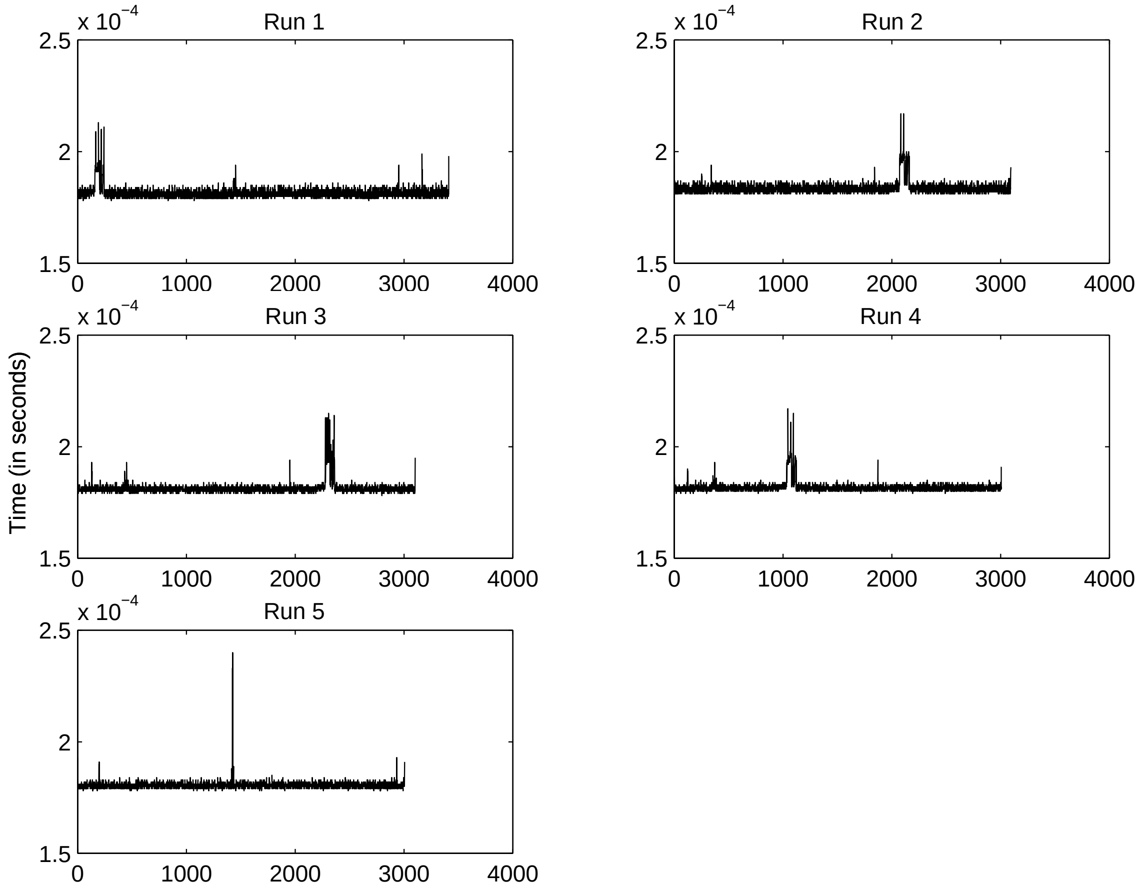

Figure 9: Data Distribution Graphs for the Monolithic Test Program

5.2 Shared-Memory Test Program

Table 5: Shared-Memory Test Program: Artifact Summary

Lines of code	839
Cycle Time	0.02 seconds (50 Hertz)

Table 6: Shared-Memory Test Program: Time Data Summary

Observation Run #:	***1***	***2***	***3***	***4***	***5***	**Comb.**
Number of Observations	3088	3147	3066	3071	3037	15409
Minimum Cycle Time (µsec)	0.458	0.457	0.451	0.467	0.457	0.451
Mean Cycle Time	0.465	0.462	0.456	0.473	0.463	0.464
Maximum Cycle Time	0.676	0.655	0.651	0.652	0.656	0.676
Cycle Time Std. Dev.	$6.36x10^{-6}$	$4.71x10^{-6}$	$5.23x10^{-6}$	$6.09x10^{-6}$	$6.17x10^{-6}$	$7.92x10^{-6}$
Adjusted Max	0.546	0.542	0.505	0.564	0.514	–
# Deadlines Missed	0	0	0	0	0	0

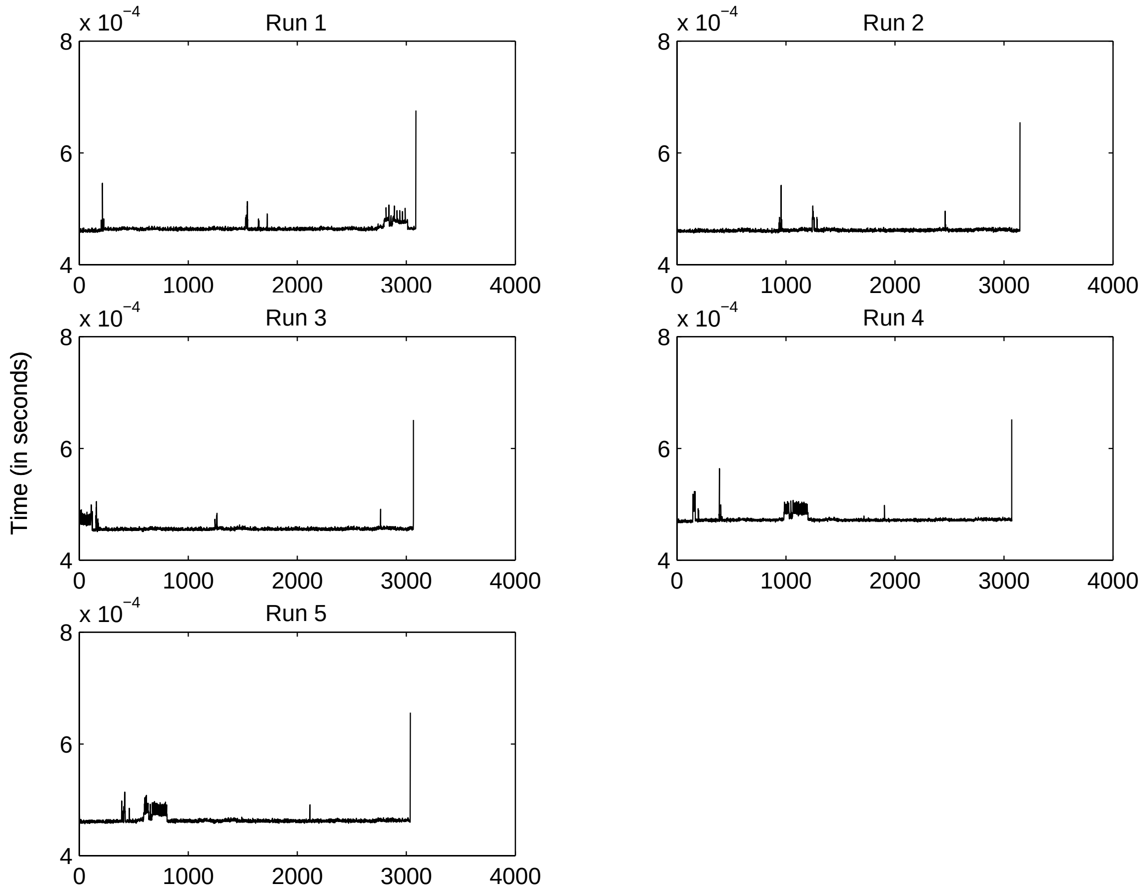

Figure 10: Data Distribution Graphs for the Shared-Memory Test Program

5.3 POSIX Message Queues Test Program

Table 7: POSIX MsgQ Test Program: Artifact Summary

Lines of Code	906
Cycle Time	0.02 seconds (50 Hertz)

Table 8: POSIX MsgQ Test Program: Time Data Summary

Observation ***Run #:***	***1***	***2***	***3***	***4***	**5**	**Comb.**
Number of Observations	3148	3062	3051	3046	3215	15522
Minimum Cycle Time (µsec)	0.540	0.527	0.527	0.523	0.538	0.523
Mean Cycle Time	0.546	0.533	0.533	0.531	0.544	0.538
Maximum Cycle Time	0.742	0.751	0.735	0.750	0.764	0.764
Cycle Time Std. Dev.	6.54×10^{-6}	7.24×10^{-6}	6.47×10^{-6}	6.96×10^{-6}	6.45×10^{-6}	9.29×10^{-6}
Adjusted Max	0.626	0.592	0.600	0.607	0.606	–
# Deadlines Missed	0	0	0	0	0	0

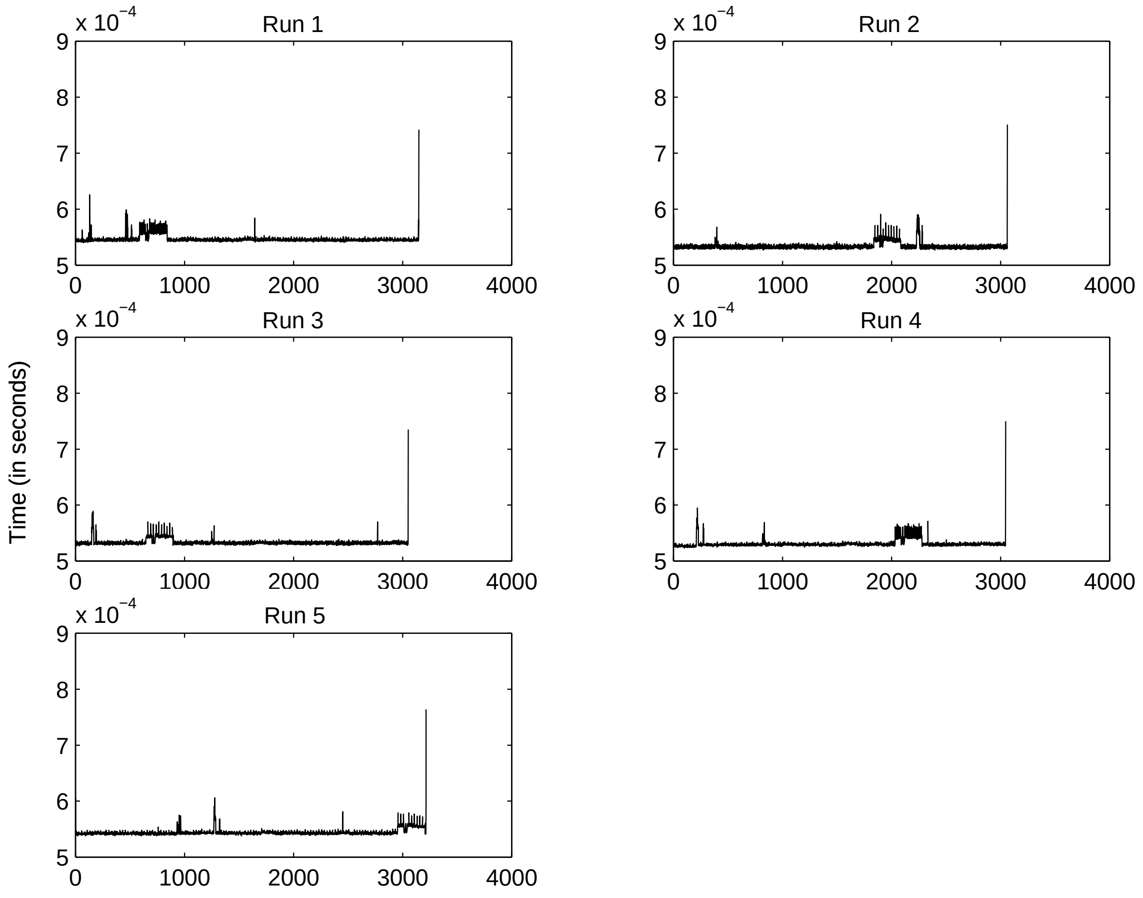

Figure 11: Data Distribution Graphs for the POSIX Message Queues Test Program

5.4 Data Tagged I/O Test Program

Table 9: Dtag Test Program: Artifact Summary

Lines of Code	885
Cycle Time	0.02 seconds (50 Hertz)

Table 10: Dtag Test Program: Time Data Summary

Observation ***Run #:***	***1***	***2***	***3***	***4***	**5**	**Comb.**
Number of Observations	3122	3148	3113	3267	3888	16538
Minimum Cycle Time (µsec)	0.681	0.699	0.704	0.696	0.716	0.681
Mean Cycle Time	0.687	0.708	0.713	0.707	0.726	0.709
Maximum Cycle Time	0.918	0.933	0.925	0.962	0.955	0.962
Cycle Time Std. Dev.	$8.15x10^{-6}$	$9.38x10^{-6}$	$8.36x10^{-6}$	$10.12x10^{-6}$	$7.93x10^{-6}$	$15.59x10^{-6}$
Adjusted Max	0.843	0.791	0.838	0.826	0.808	–
# Deadlines Missed	0	0	0	0	0	0

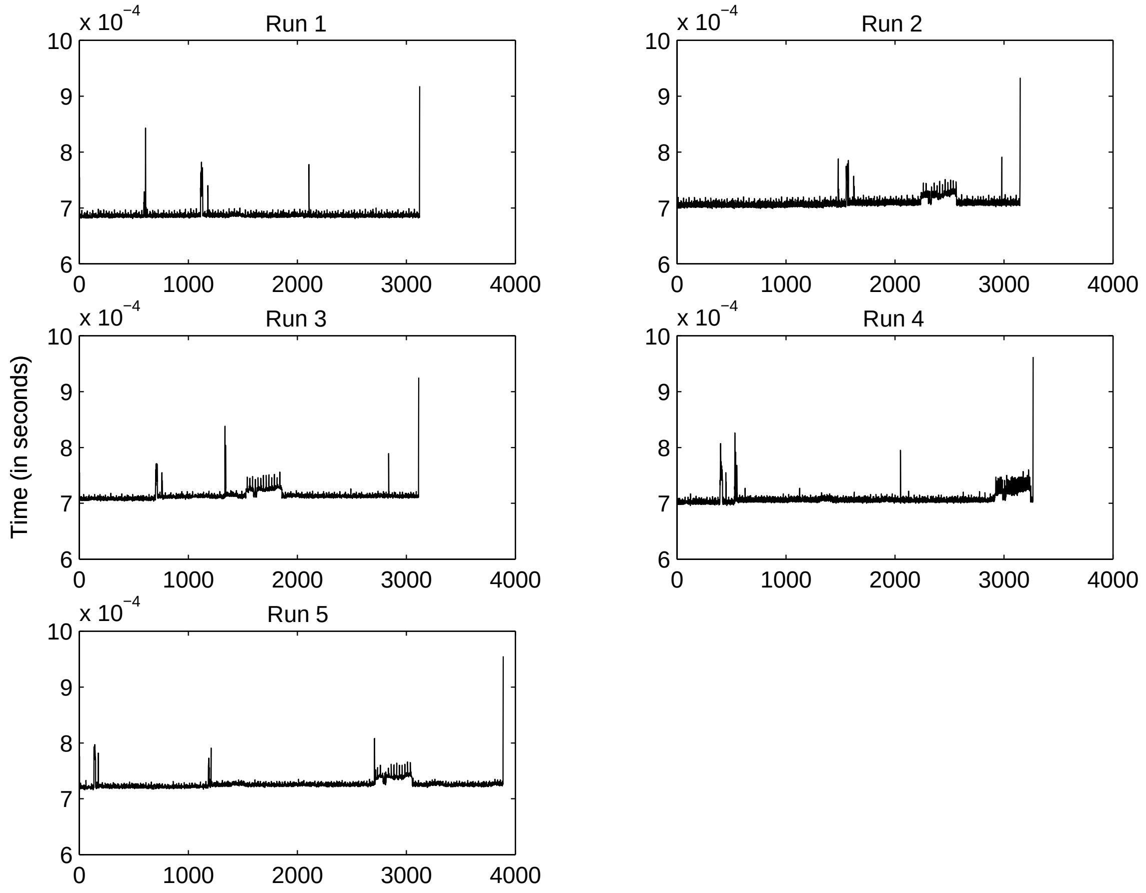

Figure 12: Data Distribution Graphs for the Dtag Test Program

5.5 Simplex Test Program

Table 11: Simplex Test Program: Artifact Summary

Lines of Code	3413 (3 controllers)
Cycle Time	0.02 seconds (50 Hertz)

Table 12: Simplex Test Program: Time Data Summary, One Controller Included

Observation Run #:	***1***	***2***	***3***	***4***	**5**	**Comb.**
Number of Observations	3914	3934	3760	3732	3836	19176
Minimum Cycle Time (μsec)	1.700	1.722	1.778	1.712	1.702	1.700
Mean Cycle Time	1.974	2.003	1.921	1.895	1.8945	1.939
Maximum Cycle Time	2.270	2.325	2.279	2.228	2.272	2.325
Cycle Time Std. Dev.	1.50×10^{-4}	1.46×10^{-4}	1.35×10^{-4}	1.29×10^{-4}	1.31×10^{-4}	1.46×10^{-4}
Adjusted Max	2.270	2.272	2.279	2.228	2.272	–
# Deadlines Missed	0	0	0	0	0	0

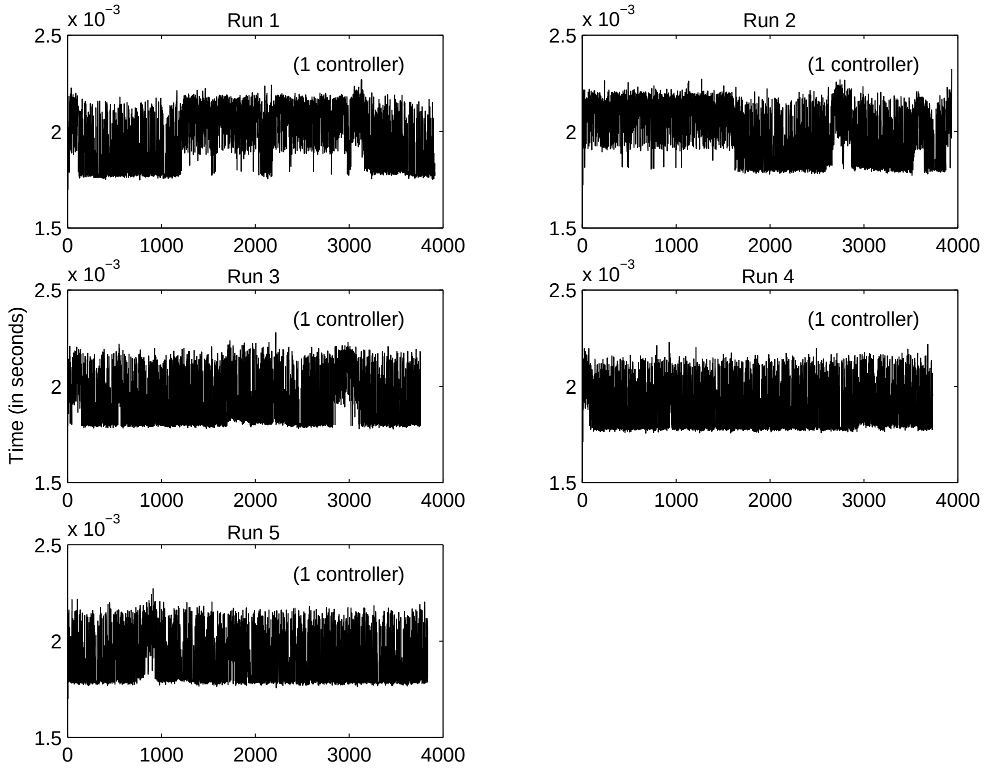

Figure 13: Data Distribution Graphs for the Simplex Test Program: One Controller Included

Table 13: Simplex Test Program: Time Data Summary, Two Controllers Included

Observation *Run #:*	***1***	***2***	***3***	***4***	***5***	**Comb.**
Number of Observations	3914	3934	3760	3732	3836	19176
Minimum Cycle Time (µsec)	2.092	2.172	2.174	2.160	2.174	2.092
Mean Cycle Time	2.728	2.812	2.705	2.577	2.717	2.709
Maximum Cycle Time	3.125	3.203	3.113	3.128	3.153	3.203
Cycle Time Std. Dev.	$3.05\text{x}10^{-4}$	$2.27\text{x}10^{-4}$	$2.25\text{x}10^{-4}$	$3.12\text{x}10^{-4}$	$1.89\text{x}10^{-4}$	$2.67\text{x}10^{-4}$
Adjusted Max	3.125	3.203	3.113	3.128	3.153	–
# Deadlines Missed	0	0	0	0	0	0

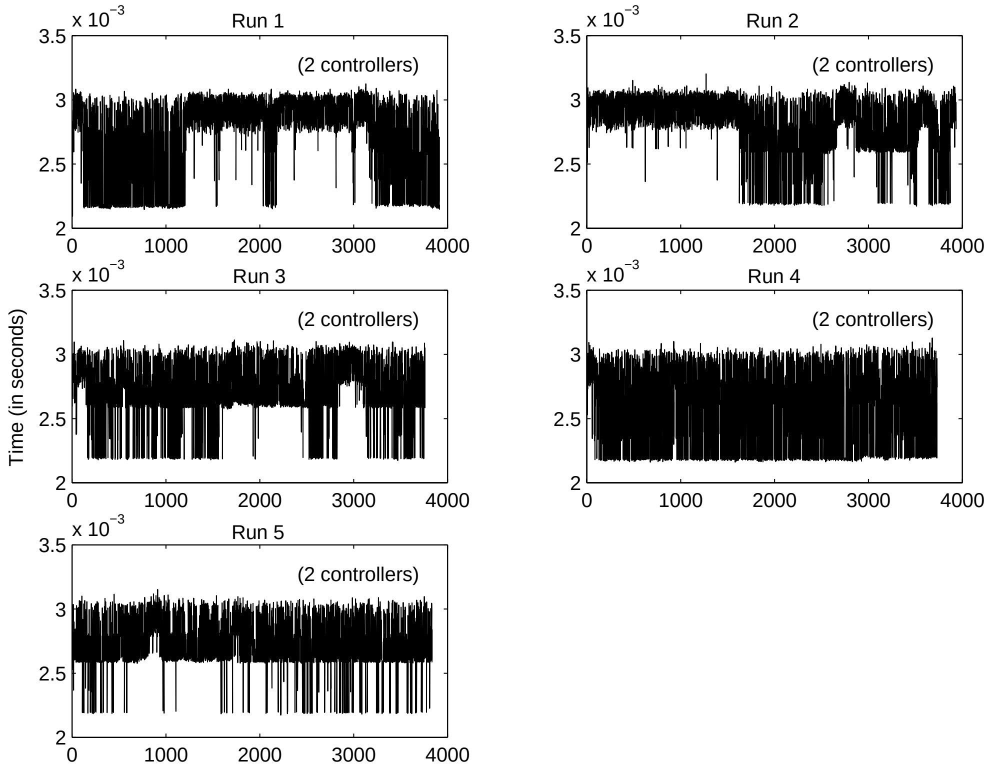

Figure 14: Data Distribution Graphs for the Simplex Test Program: Two Controllers Included

Table 14: Simplex Test Program: Time Data Summary, Three Controllers Included

Observation Run #:	***1***	***2***	***3***	***4***	**5**	**Comb.**
Number of Observations	3914	3934	3760	3732	3836	19176
Minimum Cycle Time (µsec)	2.974	3.028	3.083	3.003	3.019	2.974
Mean Cycle Time	3.378	3.411	3.292	3.300	3.298	3.337
Maximum Cycle Time	3.720	3.835	3.729	3.745	3.743	3.835
Cycle Time Std. Dev.	1.96×10^{-6}	1.88×10^{-6}	1.81×10^{-6}	1.75×10^{-6}	1.75×10^{-6}	1.90×10^{-6}
Adjusted Max	3.720	3.835	3.729	3.745	3.743	–
# Deadlines Missed	0	0	0	0	0	0

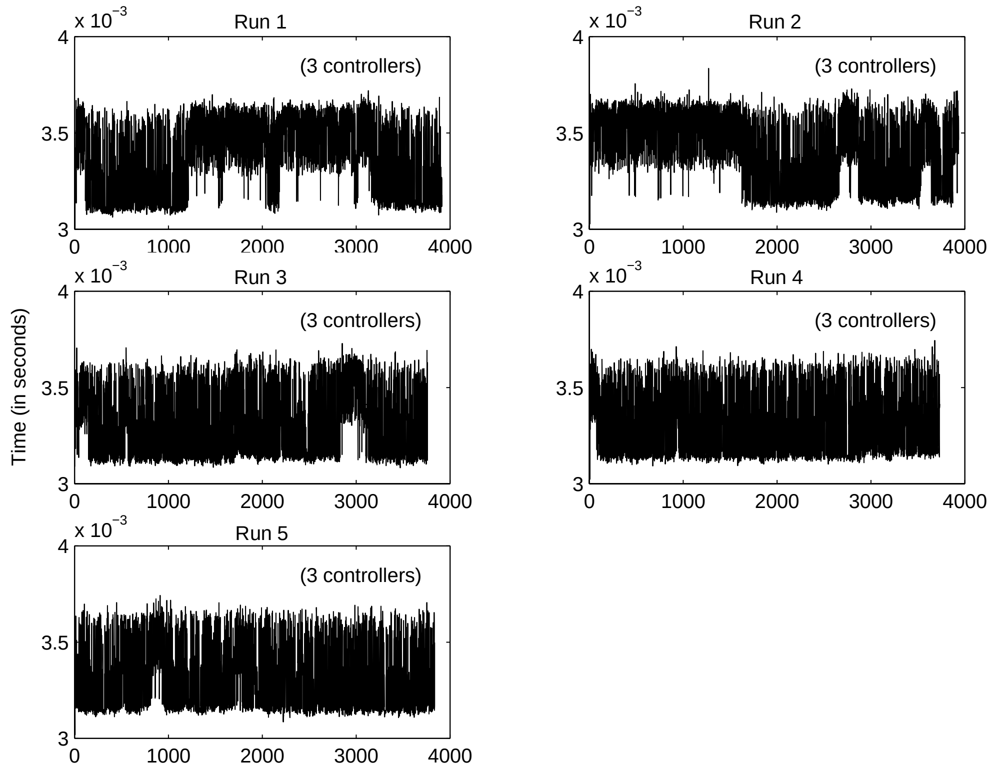

Figure 15: Data Distribution Graphs for the Simplex Test Program: Three Controllers Included

6 Testing and Data Analysis Methodology

The systems tested for this report run in an open-ended fashion, performing an open-loop control task repetitively until execution is terminated. For such systems, the total time of execution is of little interest or merit in describing the system, since the program runs for an arbitrary length of time.

The execution of the open-loop control systems tested can be characterized as a series of relatively discrete and repetitive cycles. In the course of a single cycle, the system reads sensor data from the controlled hardware, then performs calculations using the collected data, yielding control values that are output to the hardware. The test programs execute a series of program threads as a single cycle, whose specific constituents vary according to the scenario. Resource consumption can be characterized by observing each cycle repetitively.

Using the cycle as the primary unit observation, the following dynamic measures are of interest:

- computation time used in the cycle
- timeliness of cycle completion

In addition, a static measure is significant:

- creation effort

Benchmarks often aggregate observations at data collection times automatically by timing multiple executions of a module in a test loop. This approach avoids the need for specialized testing equipment by increasing the observation duration. The loop testing approach is insufficient for deadline-critical, real-time software since extreme values, particularly the worst-case execution times, are of considerable interest.

Software tested for this report is structured in modules, which execute within independent threads of control. The execution order of the threads is made predictable by the priorities assigned to the individual threads. Measurements were collected for individual iterations at selected points in the cycle. The measurement points are selected to allow the computation of the cycle execution time as a whole and for the separate constituent threads. The observations represent single cycles (or portions of individual cycles). The observations were logged and aggregated after the test run. Then a statistical program (Matlab) was used to generate standard descriptive measures to characterize average, range, and extreme values.

6.1 Computation Time

Time values were recorded using a high-resolution timer. The timer was used as a stopwatch to time individual passes through code segments. Timing stamps were read and logged, typically at the beginning and end of individual code modules.

The timing logic for the monolithic test program is extremely straightforward, with time-stamps taken at the actuation and completion of each cycle. Duration of execution is computed by a simple subtraction.

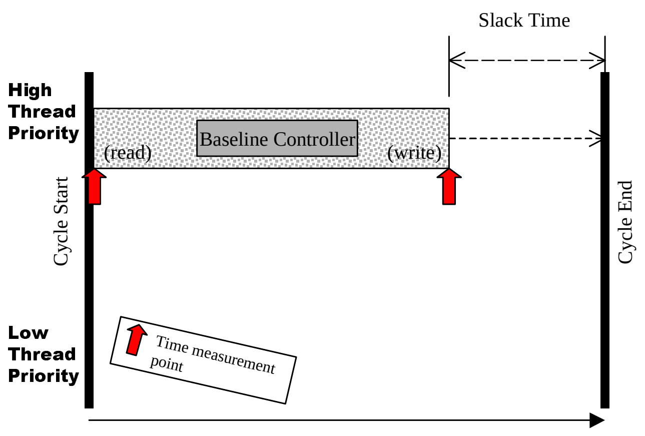

Figure 16: Timing Points for a Typical Cycle of the Monolithic Test Program
A timer reading is logged at each of the vertical arrows.

The shared-memory, POSIX message queues, and Dtag test programs split the functional code into two processes that interchange data. Timing is taken at the actuation and completion of each thread. For analysis purposes, the end-to-end cycle time (last time minus first time) is used, as the communications overhead is of interest. Excluded from time calculations are the slack times, when no task is active.

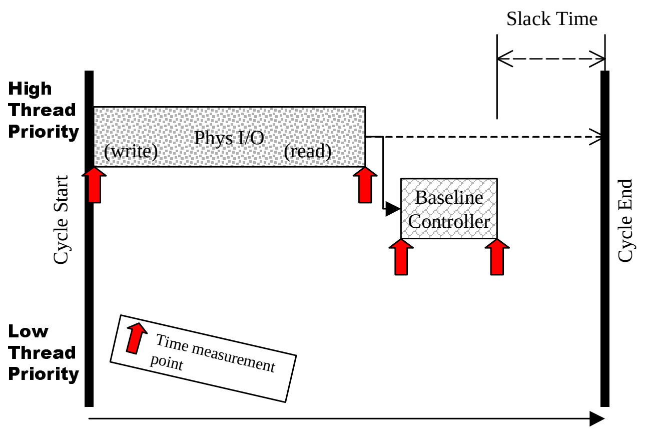

Figure 17: Timing Points for a Typical Cycle of the Shared Memory, POSIX Message Queues and Dtag Test Programs
A timer reading is logged at each of the vertical arrows. Note that the Phys I/O process first outputs the control value previously computed.

Then fresh data is read from the pendulum. This differs from the Monolith and Simplex test programs, where timing begins with the reading of data from the pendulum.

Figure 18 shows time measurement points for a representative cycle in the Simplex test program. Note that there are five major processes active during the test period. The two threads with the highest priorities temporarily suspend themselves on a hardware timer temporarily to allow the lower priority threads a chance to execute. If they awake before the lower priority threads complete, the lower priority threads are marked as failing to meet their deadlines. Note also that the safety controller is embedded in the decision module rather than being placed in a separate thread.

System overhead, including the context switch between processes, is included in module execution time. Excluded in time calculations are the slack times, when no thread is active. Not shown are several auxiliary threads that manage the communications links during dynamic replacement of processes. (They are inactive during normal operation.)

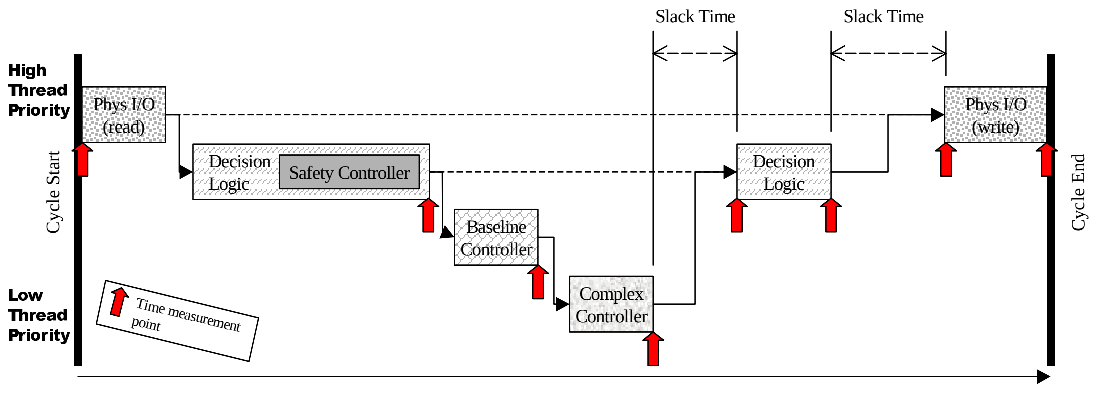

Figure 18: Timing Points for a Typical Cycle of the Simplex Test Program

A timer reading is logged at each of the vertical arrows. Task priority is used to regulate the order of execution of the individual tasks that make up the Simplex Test Program. The operating system will not interrupt a high-priority task for a low-priority task, but-high-priority tasks can suspend themselves for a period of time, allowing lower priority tasks to run. When suspension period is complete, the operating system will interrupt lower priority tasks, providing a simple mechanism to terminate a runaway task.

Since the time values for all tests are collected from a separate hardware timer, they do not depend on the operating system maintaining the system clock accurately.

6.2 Timing Overhead

Time observations require reading the real-time clock and storing the time value in memory. An event logging facility, previously written for debugging Simplex applications, was employed for this purpose.[3] As a check on the overhead imposed by this process, each test included a simple timing calibration test. A pair of timing statements was inserted back-to-back, without any intervening statements. The difference between the observed times gives a rough idea of timing overhead. However, the overhead values are not constant, as can be seen from Figure 19. Instead, the values varied from observation to observation and from run to run.

[3] Timing values are read from memory and formatted for disk storage after the observation period. This post-processing does not contribute to the estimated timing overhead.

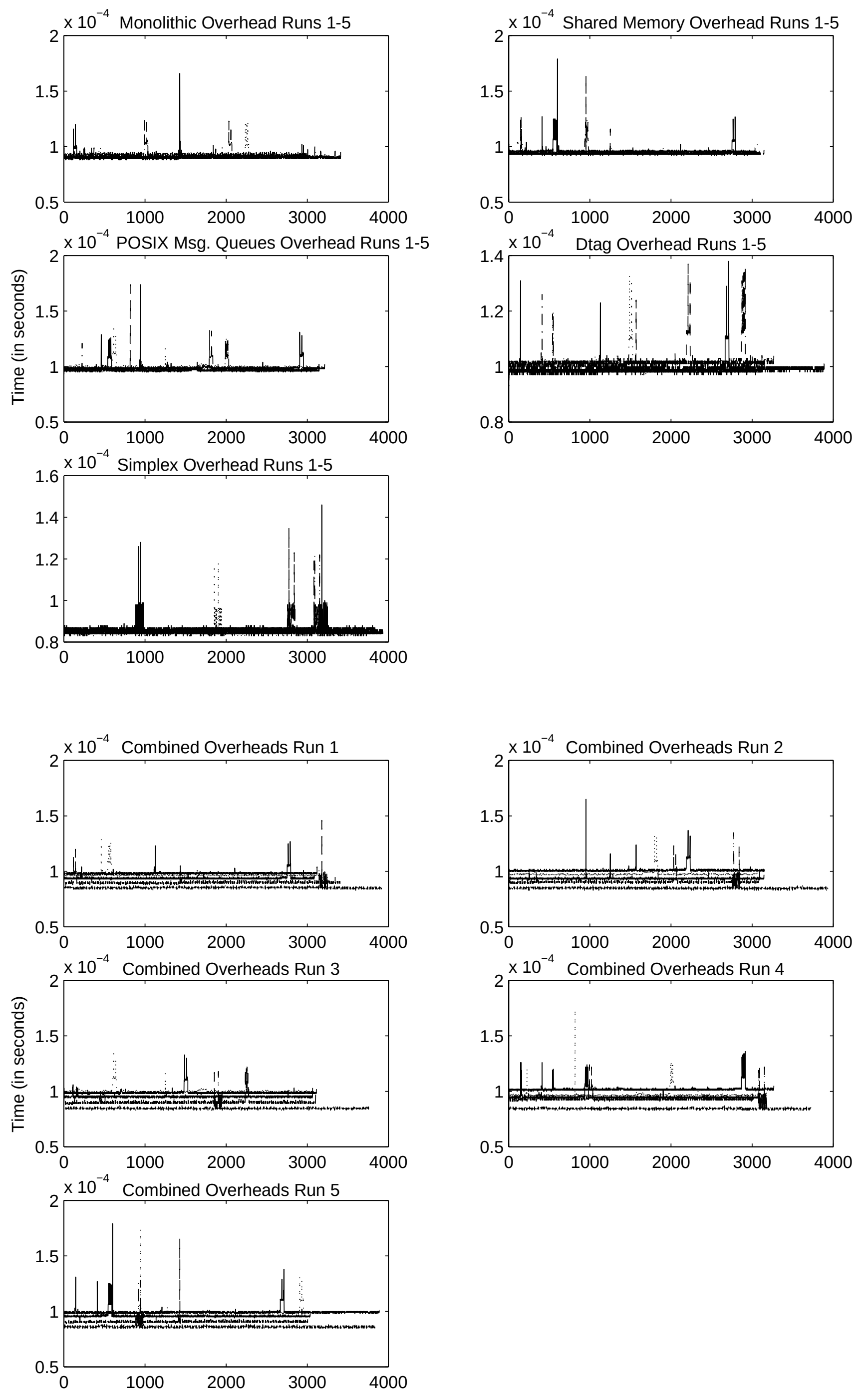

Figure 19: Timing Overhead Data

At the top, timing overhead for each data set from the five test programs is overlaid, showing the variability among test programs. Below, the data for all five runs of each test program are overlaid to show variability among individual test runs.

As shown in Table 15, timing overhead is a substantial proportion of the total observed time for most test series. Since most of the test series contained extra timing statements to capture process times, the base timing figure is multiplied by the number of timing observations made.[4]

Table 15: Timing Overhead as a Percentage of Total Time

Observation Run	*Monolithic*	*Shared Memory*	*Msg. Queues*	*Dtag*	*Simplex (1 Cont.)*
Mean Value (μsec)	181.6	463.7	537.6	708.9	1939.0
Overhead Mean Value	90.77	94.64	97.99	99.85	85.19
Overhead %	49.99	20.41	18.23	14.08	4.39
Overhead % x 3		61.23	54.69	42.26	13.18
Timing Overhead %	49.99	61.23	54.69	42.26	13.18

Figure 20 shows the relationship between timing overhead and the total observed times for the first data set for each of the test runs. When more than a single pair of timing statements was included in a test run, a third line shows the adjusted values (three times the observed value).

[4] Timing statements either form bookends at the ends of the observed period (if they are used to compute the elapsed time) or are embedded inside the code. Embedded timing statements are considered to consume the average time for a timing observation while bookend statements are considered to consume one half the average time (since part of the timing statement execution falls outside the observed period). Timing statements outside the observed period are ignored.

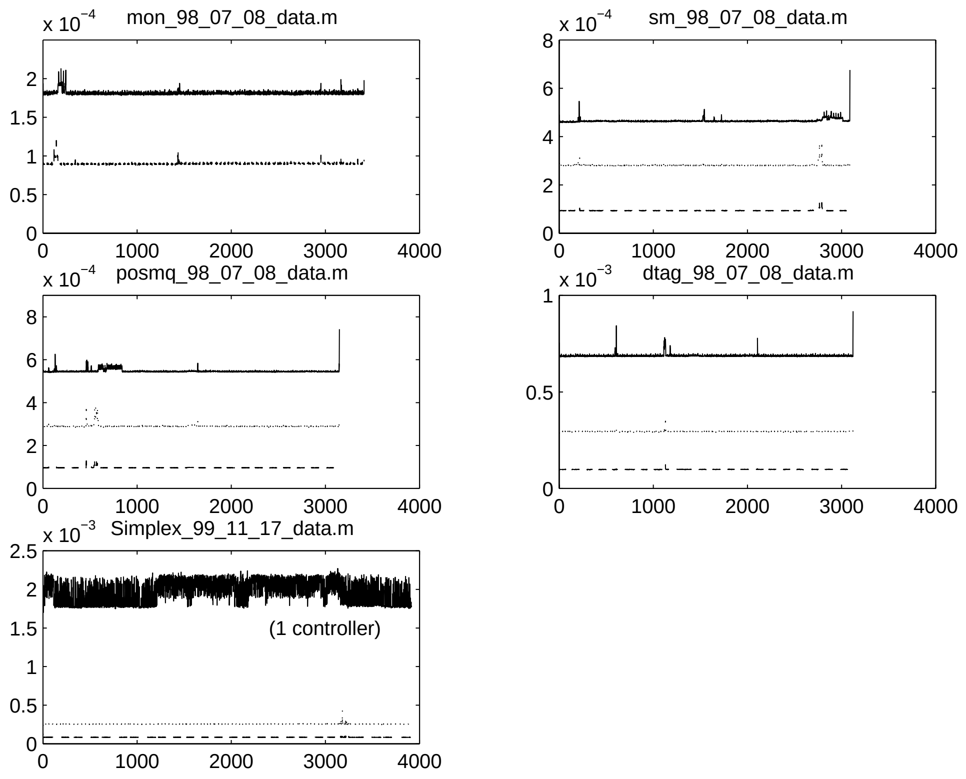

Figure 20: Timing Overhead as a Proportion of the Observed Data for the First Run of Each Test Program

The observed time from back-to-back timing statements is graphed with the corresponding raw data. The lower line is the timing overhead while the top line is the (unadjusted) raw time. The middle line, where present, is three times the timing overhead value and represents the number of timing values in the test run. Note that for the Simplex data, the timing overhead is computed using the end time from the prior cycle, so the overhead cannot be computed for the first cycle.

If the average timing overhead is subtracted from the data, Figure 7 can be redrawn with the revised data as Figure 21:

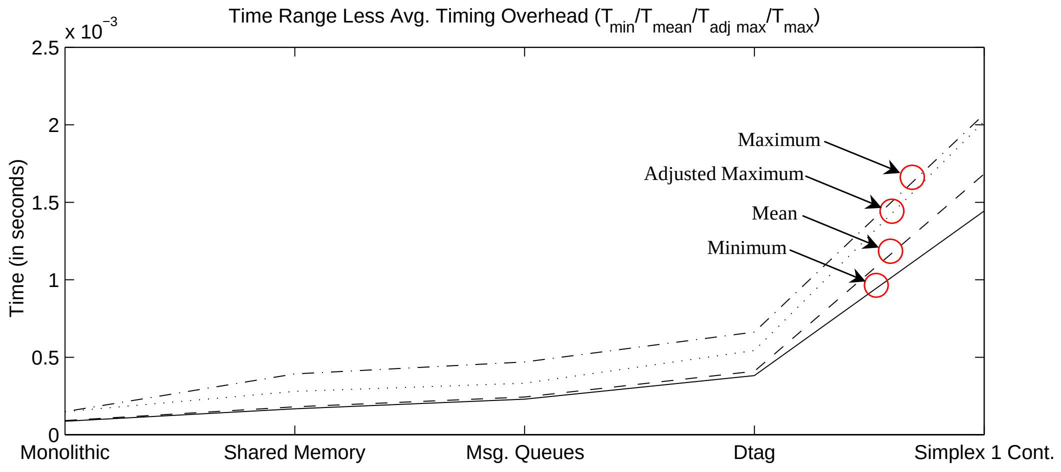

Figure 21: Time Versus Communications Complexity Adjusted for Timing Overhead

Communications complexity increases from left to right while the vertical axis shows time. The lines show, from bottom to top: minimum, mean, adjusted maximum, and maximum value for all data values for each test series.

Since the estimated timing overhead is a relatively smaller proportion of the Simplex test series, the adjusted graph shown in Figure 21 shows that Simplex has a somewhat higher overhead when the cost of observing the program is factored from the data. However there is no way to prove that the overhead computed from the back-to-back timing observations accurately reflects the timing overhead incurred during the operational portion of the program. Instead, experience suggests that these two values may be significantly different. For instance, in other benchmarking tests (run on different hardware) the overhead value obtained by benchmarks using back-to-back timing observations depended critically on the location of code in memory.[5]

6.3 Timely Completion

Real-time software must meet specific timing goals; an overrun deadline represents a serious error. The tested software checks for failure to meet deadlines and logs any deadline overruns. No deadline overruns were observed.

[5] Altman, Neal & Weiderman, Nelson. "Timing Variation in Dual Loop Benchmarks." *Ada Letters VIII*, 3, (May/June 1988): 98-106.

6.4 Creation Effort

There are many ways to measure software development effort. The test software used existing source code where development effort was not specifically recorded. Since the Simplex source was developed originally to test Simplex concepts and methods, time spent in development would not accurately reflect a normal engineering effort. Instead, the code artifact itself was used to measure creation effort.

Line-of-code measures were taken on the tested code. Code for this study was written in C and C++; only code written specifically to implement the Simplex and Comparison systems were included in the counts. Library and support code was included in the count when written specifically for the Simplex software. Standard and purchased library code was not included. The count differentiates between white space (blank lines), comments, and executable code. Executable code includes declarations and statements, and the code was counted by line and by terminator (semicolon).

For purposes of summary reporting, we used a measure defined as the total count of code lines and comments, excluding blank lines. Comments were included in the code count using the logic that their generation is an important part of properly structured programs and that the original creators did not receive any incentive to increase code size through large numbers of comments.

7 Discussion and Conclusion

In this paper, an evolutionary sequence of programs was tested to benchmark the overhead required to support the application of the Simplex paradigm to an open-loop control problem. In a sense, the test series was constructed by devolution, since a working Simplex artifact was simplified by the reduction of communications capabilities to provide the less complex programs. Since the programs were dependent on system software services as well as the underlying hardware for many features, the specific time values observed apply only to the tested systems. What is more interesting is that the flexibility that Simplex provides can be implemented at a cost (in the test series) of an order of magnitude. Comparing the simplest monolithic test program, Simplex consumes 10.7 times more time to perform the control task. This appears to be high, but bear in mind that the control task undertaken in the test was a simple one. For more complex applications, where the program performs significant processing, the Simplex overhead will be a much smaller proportion of the total processing time.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (LEAVE BLANK)	**2.** REPORT DATE September 2000		**3.** REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE *Simplex Architecture Performance and Cost*			**5.** FUNDING NUMBERS F19628-00-C-0003
6. AUTHOR(S) Mike Gagliardi, Theodore Marz, Neal Altman, John Walker			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			**8.** PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2000-TR-006
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116			**10.** SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2000-006
11. SUPPLEMENTARY NOTES			
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			**12.B** DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) The Simplex™ Architecture facilitates the building of dependable and upgradable real-time systems. Before using the technology, potential users want to know more about the costs of adopting the Simplex paradigm compared to the benefits gained by using it. This paper examines Simplex performance and the costs associated with its use.			
14. SUBJECT TERMS benchmark, performance, real-time control, Simplex Architecture			**15.** NUMBER OF PAGES 43
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18.** SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	**19.** SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	**20.** LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102