Measuring Time in Cm*

Thomas H. Kong, Alfred Z. Spector, Daniel P. Siewiorek
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
May 1986

Abstract

A system clock is often used as a time-keeping device for measuring software performance. Obtaining accurate clock reading in a distributed system with only one system clock may be difficult due to communication delays and clock contention. This paper investigates the problems associated with a single time base in multiprocessor systems and uses the Cm* multiprocessor as a research vehicle. First, the accuracy of the reported time was found to be greatly affected by the number of simultaneous clock reads and overall system workload. Second, methods were developed to compensate the clock readings by monitoring the system load during the time measurements. These methods correct readings so that they are within 7μS. of the actual time that the clock read request is issued. Finally, to demonstrate the utility of the methods, an experiment was performed to measure the latency of messages and message-based remote procedure calls.

Technical Report CMU-CS-86-136

This project was supported by NASA Langley Research Center under contract number NAG-1-190, by NSF under contract number MCS-8120270, and by the Department of the Army under contract number DASG-60-80-C-0057.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NASA, NSF, the Department of the Army, or the U.S. Government.

Table of Contents

List of Figures

1 Introduction

Software methods that rely only on a high resolution clock are more commonly used to measure the performance of computer systems than are methods involving the use of special purpose hardware monitors. Certainly, a major motivation for the use of software is cost. However, software methods are also more flexible: they simplify the automation of performance evaluation, from data collection to data reduction, and they permit performance analysis to be done remotely without probing the internals of the machine. The disadvantage of software methods is that they may be prone to inaccuracy because of possible interactions between the measuring and measured software.

Inaccurate clock readings may be particularly prevalent when the system clock is used as a time source. This is because the amount of time required to read the system clock may depend upon various system loads. Therefore, when a time value is returned to the calling program, it may be inaccurate. In a distributed system where communication delay is dependent on system activity, and where a large number of subsystems can attempt to read the clock simultaneously, the time value that is returned to the calling program may be even less accurate.

This problem is serious in the case of the Cm* [Jones 80] clock. A preliminary study showed that the result of a clock read can be in error (i.e., outdated) by as much as 2mS, depending on the system load and the amount of contention for the clock. Thus, we wanted methods that would permit more accurate clock reads for Cm*. More importantly,,we wanted to investigate the general feasibility of measuring the performance of multiprocessor operating systems using software methods.

The next section surveys previous work related to this project and describes the research vehicle, Cm*, and its two operating systems, StarOS and Medusa [Jones 80]. Section 3 discusses the clocks in Cm* and the mechanisms that Cm*'s two operatings systems use to access them. The accuracy of time measurements as a function of system load is presented and shown to be a problem. Section 4 discusses elapsed time measurements on Cm* and presents algorithms that yield more accurate measurements of elapsed times. The accuracy of these methods is illustrated by a few examples.

Section 5 presents measurements of the latency of message passing and remote procedure call mechanisms used on Cm* in order to illustrate the usage of one of the more accurate clock algorithms. Section 6 presents general conclusions.

2 Background

2.1 Related Work

Related work can be divided into the categories of empirical performance evaluation, elapsed time measurement, and performance evaluation of Cm*. With respect to the first category, we refer the reader to Ferrari [Ferrari 78] for a survey.

Obtaining accurate measurements of elasped time can be difficult on multiprocessors and distributed systems. Providing each processor with its own time source is one useful approach, but this can be costly for very large systems and lead to synchronization problems. Lamport [Lamport 78], Ellingson [Ellingson 73], and Marzullo [Marzullo 83] discuss the general issues and present techniques for synchronizing independent time sources. Using a single time source is an alternate approach that eliminates synchronization problems, but it is prone to inaccuracy due to communication and contention delays. Because Cm* does not have an independent clock for each computer module, this work concentrates on the use of a centralized clock.

At Carnegie-Mellon, many measurements of both Cm* and C.mmp [Wulf 81] have used specialized hardware. For example, early measurements of Cm* performed by Raskin [Raskin 78] used the Cm* Map-Bus Monitor, logic state analyzers, and hardware counters. Marathe's measurements of the operating system kernel of C.mmp/Hydra used both hardware and software methods. As he correctly observed, measurement tools should match the level of the measurement [Marathe 77]. This paper is specifically concerned with software methods for performance evaluation that require no special hardware other than a high resolution system clock.

2.2 Research Vehicle

2.2.1 Cm* Hardware Structure

Cm* is a multiprocessor consisting of fifty processor-memory pairs made up of Digital Equipment Corporation LSI-11's. Each processor-memory pair is called a *computer module.* These computer modules are grouped into five *clusters*, forming a hierarchical switching structure. The lowest level of the switching hierarchy consists of the *Slocals*, which are switches placed between each processor and its local memory. Their function is to determine if references generated by the processor can be directed to the local memory. If the references cannot be directed to local

memory, the Slocal forwards the address through the *Map Bus* to the *Kmap* of that cluster for further address translation.

The Kmap is a high speed microprogrammable communication controller that permits computer modules (Cm's) of its own cluster to communicate with each other, and cooperates with other Kmaps to service inter-cluster communication requests. All communication between the Kmaps is implemented via packet-switching rather than by circuit-switching to avoid possible deadlock over dedicated circuit-switching paths. In addition, since the Kmap is much faster than the main memory of the LSI-11's, the Kmap is active only for a small fraction of the time of a memory reference. Therefore, packet-switching allows the Kmap to service more than one request concurrently. In addition to their communication functions, some StarOS and Medusa functions are executed by the Kmaps.

2.2.2 StarOS

StarOS is a message based, object oriented operating system for Cm* that is described by Gehringer and Chansler [Gehringer 81] and Jones et al. [Jones 79]. Briefly, all StarOS information, including code and data, are contained in objects. Each object has an associated type and a special set of type-specific operations. Users can define their own abstract object types.

A StarOS object is made accessible via the possession of a capability, which contains the name of the object and a list of rights. Capabilities themselves do not contain the address information of the objects they name. Rather, they contain pointers to the *descriptors* that contain the physical locations of the objects. This way, if an object named by a number of capabilities is to be relocated physically, only its descriptor needs to be updated while all the capabilities remain unchanged.

The StarOS message facility supports the transmission of messages containing one capability or one data word. Thus, messages of size larger than one word must be passed by reference. By reference semantics are possible because names of objects are known system-wide and a capability is sufficient to access an object anywhere in the system.

2.2.3 Medusa

Medusa is another message based operating system for Cm* and is described by Ousterhout et al. [Ousterhout 80a] [Ousterhout 80b]. All Medusa information is stored in *objects* that are addressed through *descriptors* that contain the type, the location, and the size of the objects. Descriptors are stored only in protected objects known as *descriptor lists*. Each Medusa process, known as an *activity*, has two descriptor lists. The *private descriptor list* keeps the descriptors to objects that are private to the process, while the *shared descriptor list* keeps the descriptors to objects that are shared by all processes within a collection of cooperating processes that are performing a given computation.

In Medusa, all objects are defined by the system. The message facility of Medusa supports messages of variable size. Messages are transmitted by value through special objects called *pipes*. These are similar to Unix [Ritchie 74] pipes except that only complete messages can be sent or received from the pipes, and that both the identity of the sender and the size of the message are available to the receiver.

3 Clocks in a multiprocessor

One simple technique for providing access to a global time source is to have a globally readable clock with a communication delay that is small (compared to the clock resolution) and fixed, regardless of system load. Such a clock may require a special bus allowing multiple simultaneous read accesses. An example of such a bus structure is the interprocessor bus of C.mmp [Wulf 81]. In the C.mmp implementation, there is a 56-bit global clock of 4 microseconds resolution. The value of this clock is continually broadcast on the interprocessor bus.

However, in a large and more loosely coupled system, it may not be feasible to devote a special purpose bus to the global clock because of the amount of cabling involved as well as the problems associated with long signal delays and bus arbitration. Also, broadcasting the clock value on the general purpose bus would use a large portion of the cycles available on the bus. For these two reasons, broadcasting the clock value is generally not done in large distributed systems. In Cm*, when a subsystem needs to know the system time, it establishes a connection with the clock and then reads its value. This way, communication occurs only when necessary. However, because the time required to establish a connection depends on bus activity, and the transmission delay depends

on the physical location of the subsystem, the total delay is unpredictable. When multiple requests for the system time arrive simultaneously, bus contention results and a queue is formed. The wait time in this queue adds further uncertainty to the total delay.

In this section, the problems in reading a central clock will be examined, and their effect on the performance of the Cm* clock will be studied. Schemes that yield more accurate clock readings will be proposed.

3.1 Cm* clocks

Cm* currently provides three 32-bit real time clocks for time measurements, only one of which is used to provide the system time. They are connected as peripherals to Cm3 on cluster 1, Cm4 on cluster 2, and Cm14 on cluster 5. These clocks have a quartz crystal time-base with an adjustable resolution. The maximum resolution is 0.5 microseconds, but they are currently set for 2 microseconds. This yields a maximum range of $2^{32} * 2\mu S = 2.386$ hours. The clocks can be zeroed under program control for interval measurements.

Since the LSI-11 uses memory mapped I/O, reading the clock is a simple read to a specific location in the I/O page (page fifteen) of the LSI-11 address space. For both StarOS and Medusa operating systems, reading the system clock is implemented via remote memory references.

3.2 Clock reading routines and their performance

In both StarOS and Medusa, clock reading is performed using procedure calls rather than by an LSI-11 "MOV" instruction. This is because the clock is 32 bits while the data bus is only 16 bits wide. Thus, to read the full clock requires at least two memory references. Since the clock is always running, there is no guarantee that the high and low order words read correspond to the same 32-bit clock word. This is because after reading the first word, the low order word may overflow and wrap around at a clock tick, invalidating the first word read.

When this project began, both StarOS and Medusa provided a standard routine for reading the clock. For future reference, this algorithm is named "Varying-Read" algorithm because the clock register is read either three or four times depending on the value of the clock. Below is the pseudo-code for this routine:

```
Varying-Read:

FirstHi  = Read high order word of clock;
FirstLow = Read low order word of clock;
SecondHi = Read high order word of clock;
if SecondHi > FirstHi then begin
        SecondLow = Read low order word of clock;
        return SecondHi and SecondLow as the clock result;
end
else begin
        return FirstHi and FirstLow as the clock result;
end;
```

When SecondHi is greater than FirstHi, the low order word must have wrapped around between the first and second read of the high order word. Since it is not known whether FirstLow was read before or after the wrap around occurred, the low order word must be read a second time.

A preliminary experiment was set up to evaluate the performance of the Varying-Read clock routine of Medusa. The objectives were to determine the average execution time of the routine and to see how the accuracy was affected by the system load. The experiment measured the elapsed time between two successive clock read procedure calls. This elapsed time was identical to the execution time of the routine including all the remote memory references.

The experiment was performed with eight Cm's distributed between clusters 2 through 5 reading the clock in cluster 1. The experiment was then repeated with thirty Cm's, also distributed between clusters 2 through 5. The results for all processors are summarized in Figure 1, which plots the elapsed time between two clock reads against the time elapsed since the beginning of the experiment. The total intercluster memory reference rate is approximately 90 thousand references per second in the case of 30 Cm's and approximately 53 thousand references per second in the case of 8 Cm's.

Figure 1 reveals periodic peaks and troughs in the 30 Cm's curve. The peaks occurred when the low order word of the clock wrapped around during the execution of the second clock read routine, causing the second clock reading routine to read the low order word a second time. In this instance, there were 4 clock reads between the significant low order reads. The troughs occurred when the low order word wrapped around during the execution of the first clock read routine, causing the low order word to be read again. In this instance, there was only 1 clock read between the significant low order reads. (For a more detailed explanation of the reason for the peaks and

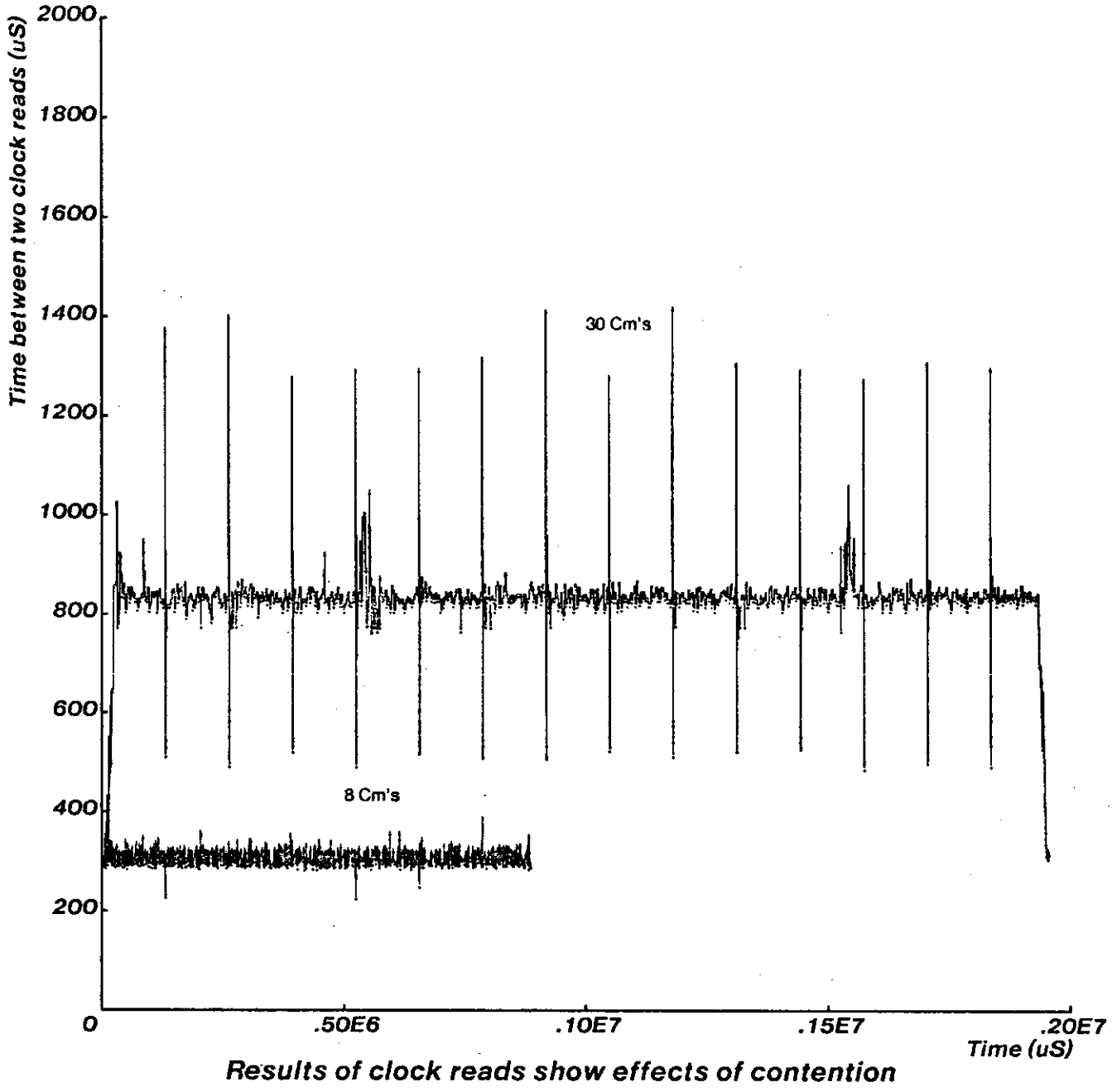

Figure 1: Performance of Medusa Varying-Read clock routine

troughs, refer to Kong's Masters Project Description [Kong 82].) Note that in the 8 Cm's case, there were fewer Cm's reading the clock and the probability of reading the clock while its low order word wraps around was much lower; hence the peaks and troughs did not appear regularly.

Figure 1 also shows for the 30 Cm's case that the elapsed time rose at the beginning of the

experiment from approximately 300μS to over 800μS and then fell from 800μS to approximately 300μS at the end of the experiment. This was because not all the Cm's started and finished simultaneously. Hence, there was less system load and contention at both the beginning and the end of the experiment. In the 8 Cm's case, the average value was approximately 300μS and no rise or fall was seen; With only 8 Cm's reading the clock, there was insufficient traffic to slow the clock read routines.

Because of the Varying-Read clock routine's erratic behavior when the low order word of the clock flips, two new clock reading routines were written. The first one was a modification to the original Varying-Read routine. It reads both the high order and the low order word twice, and has the property that it always returns the first low order word read as the low order word of the clock. Its execution time is essentially independent of the value of the clock readings, as shown in Figure 2. In Figure 2 the rate of remote memory references was 96 thousand per second for the 30 Cm's case and 64 thousand per second for the 8 Cm's case.[1] Below is the pseudo-code for the routine:

```
4-Read:

FirstHigh  = read high word of clock;
FirstLow   = read low word of clock;
SecondHigh = read high word of clock;
SecondLow  = read low word of clock;
IF SecondHigh > FirstHigh THEN          /* clock flipped */
        IF SecondLow > FirstLow THEN    /* flip was before FirstLow */
                return FirstLow and SecondHigh as result
        ELSE    /* flip was after FirstLow read */
                return FirstLow and FirstHigh as result
ELSE    /* no flip occurred between FirstHigh and SecondHigh */
        return FirstLow and FirstHigh as result;
```

The second routine reads only the low order word of the clock and computes the value of the high order word. It assumes the clock is read by each Cm at least once in every interval T_f, where T_f is the time between two low word flips and is equal to $2^{16} * R$, where R is the number of seconds between a clock tick. With the present R of 2μS, T_f equals 0.131 seconds.

The routine makes use of two static variables $OldLow and $Hi on each Cm. During a clock reset, these variables are zeroed. Every time the routine is called, the low order word of the clock is

---

[1]These numbers are higher than for the previous clock reading routines because 4 remote memery references are done for each clock read.

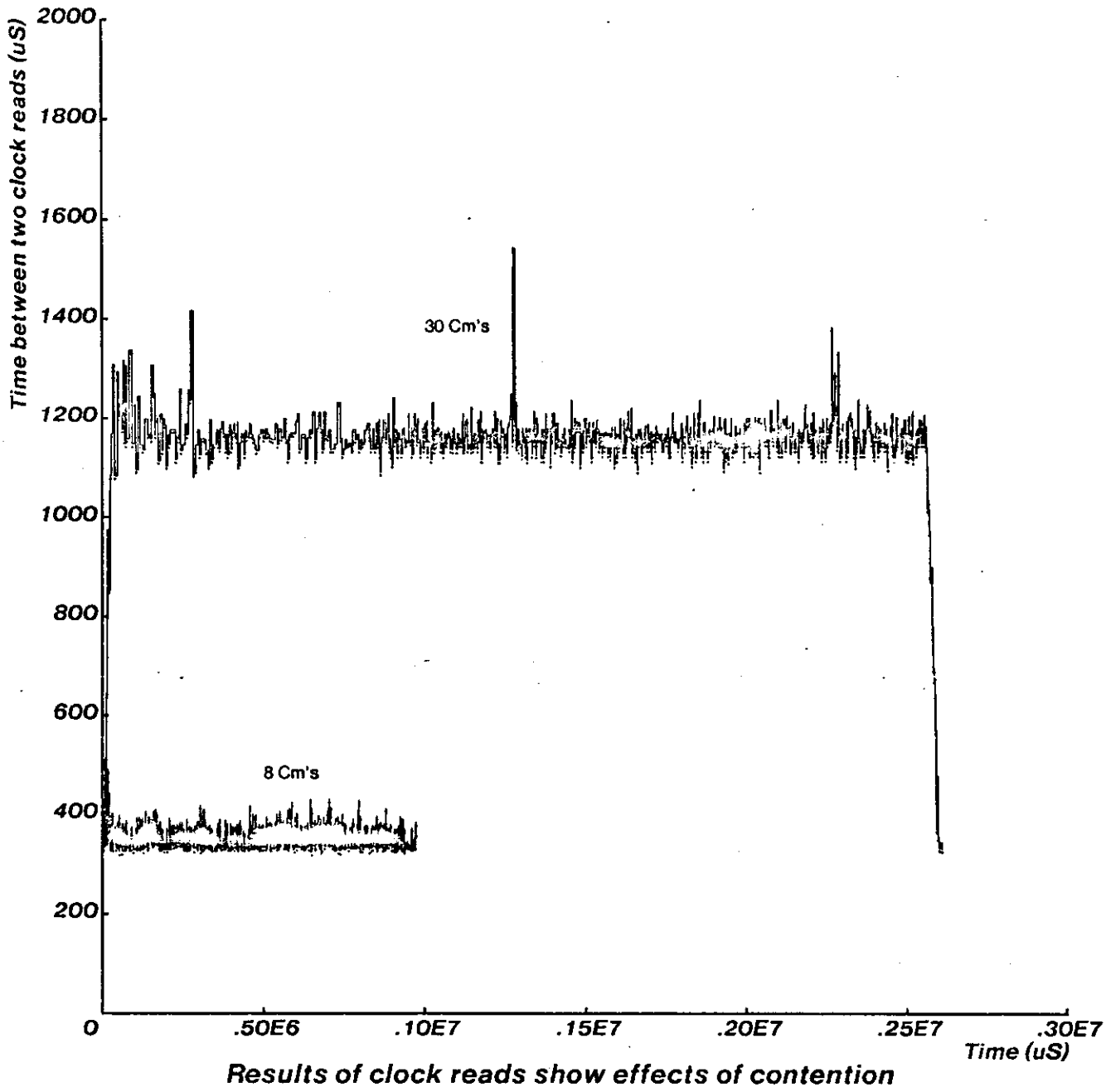

Figure 2: Performance of 4-Read clock routine running under Medusa

read and is compared with the value of $OldLow. If the value of $OldLow is higher than the current value of the low word, a flip must have occurred. The variable $Hi is then incremented. If the value of $OldLow is lower than that of the low word of the clock, no flip has occurred and the value of $Hi remains unchanged. Below is the pseudo-code for the routine:

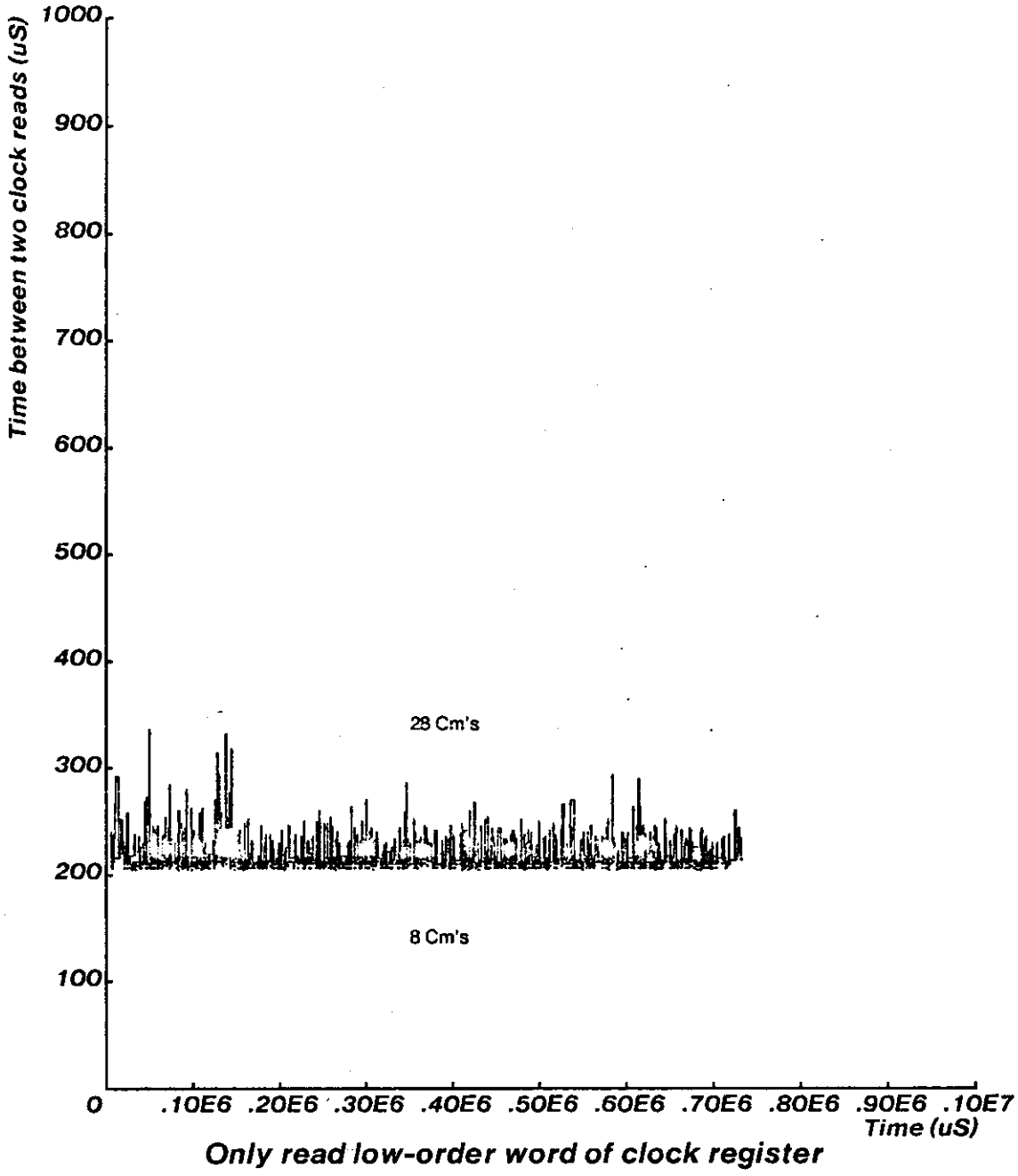

Figure 3: Performance of Medusa 1-Read clock routine

```
1-Read:

STATIC $OldLow;
STATIC $Hi;

Low = Read low word of clock;
IF $OldLow >= Low THEN
        $Hi = $Hi + 1;
$OldLow = Low;
return Low and $Hi as the result;
```

This routine has a constant execution time. The performance of this 1-Read clock routine is summarized in Figure 3. Here the remote memory reference rate was 76 thousand per second for the 28 Cm's case, and 23 thousand per second for the 8 Cm's case. This lower rate of remote memory reference makes the routine execution time quite insensitive to the increasing number of Cm's reading the clock. Therefore, the differences between the 28 Cm's curve and the 8 Cm's curve were so small that the two curves are visually indistinguishable.

Since the 1-Read and the 4-Read routines were to be used, a new set of experiments were set up to test the performance of both routines as a function of load under both StarOS and Medusa. The experiments involved the measurement of elapsed time between two clock reads as a function of the number of Cm's reading the clock.

3.2.1 StarOS results

Both the average execution time and the standard deviation of the 4-Read clock routine increases with the number of participating Cm's. This data is graphically presented in the histograms of Figure 4, which are normalized to give the same area under the curve. The distributions of all the curves appear to be Rayleigh with a lower bound of 310μS, which is the minimum time required to execute the 4-Read clock routine. Though it cannot be seen in the figure, a careful study of the data shows there is a small peak between 630μS to 760μS in addition to the main peaks. This is due to 60 hz line clock interrupts occurring between two clock reads.

The average execution time of the 1-Read routine is quite insensitive to the increasing number of Cm's reading the clock. This is because the load generated by this clock routine is low enough for the Kmap to handle without saturating. Figure 5 is the summary of all the histograms normalized to give the same area under the curve. Since the average execution time varies very little and the standard deviation of the results remains almost constant, all the curves are similar and

overlapping. Also due to line clock interrupts, there is a small secondary peak between 470μS to 550μS.

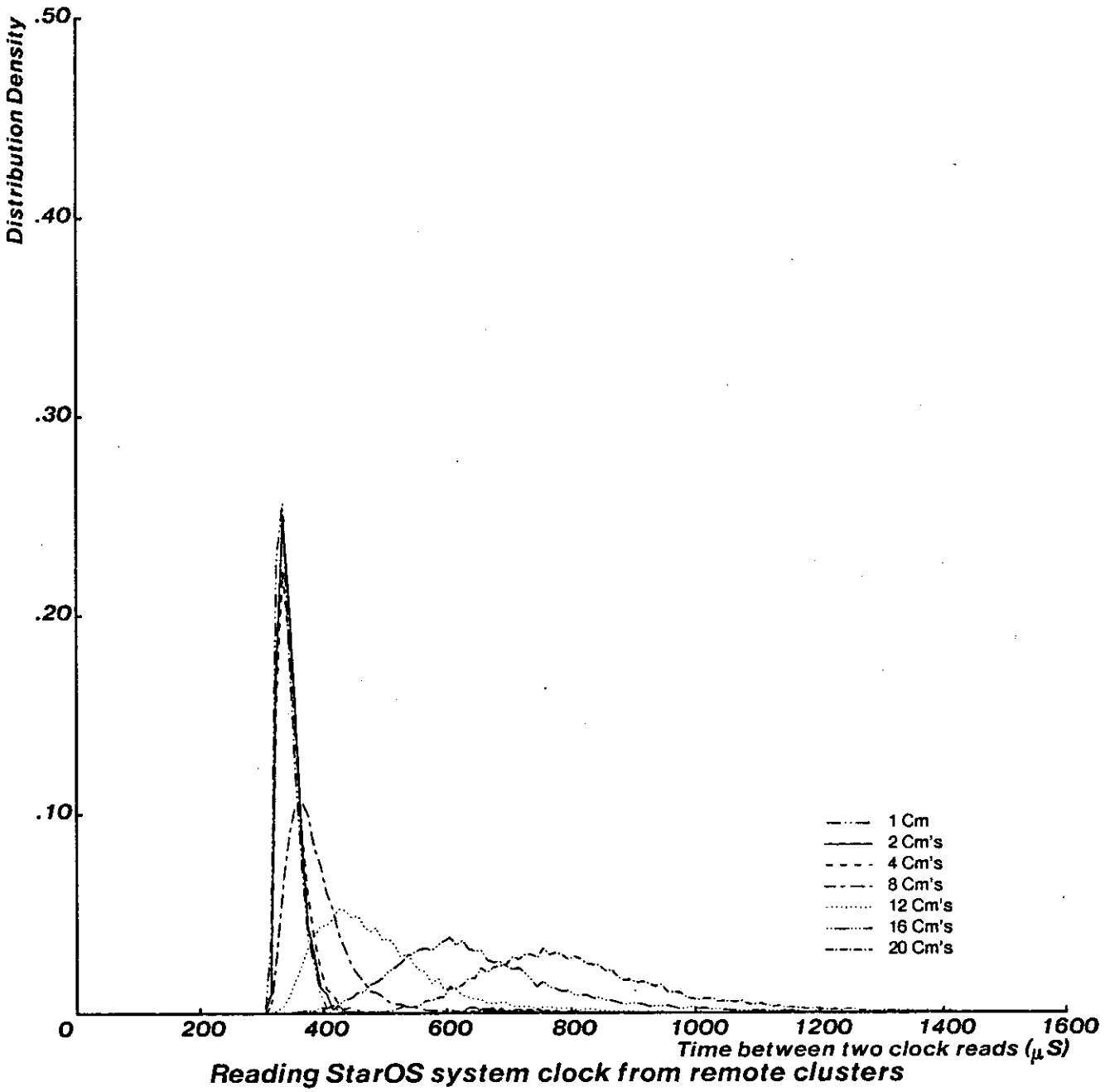

Figure 4: Performance of StarOS 4-Read clock routine

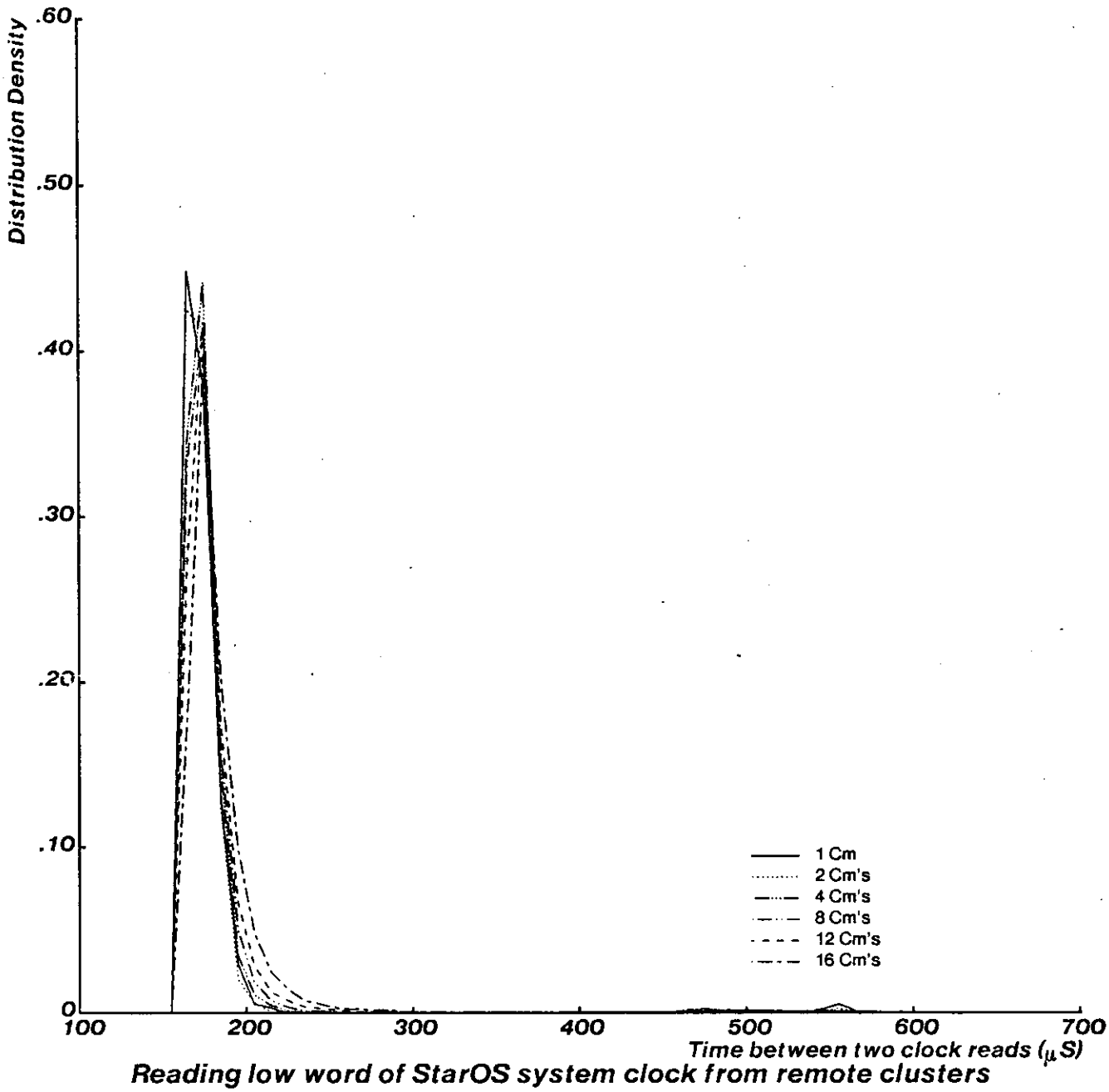

Figure 5: Performance of StarOS 1-Read clock routine

3.2.2 Medusa results

The minimum time required to execute the 4-Read Medusa clock routine is approximately 320μS. The average time increases as the number of Cm's reading the clock increases. Figure 6 is the normalized plot of all the histograms. An interesting observation is that while the standard

deviation of the result increases as the number of Cm's increases for small numbers of Cm's, it starts to fall at some point between 16 Cm's and 20 Cm's. This phenomenon is believed to be due to a complicated queuing mechanism at the Kmap.

Figure 7 is the normalized distribution that summarizes the results of the Medusa 1-Read clock routine. The standard deviation of the result is extremely small, and the mean value does not change significantly with increasing number of Cm's reading the clock. Apparently, the load presented by this routine is so small that all clock read requests to the Kmap are processed immediately without having to wait in the Kmap queue.

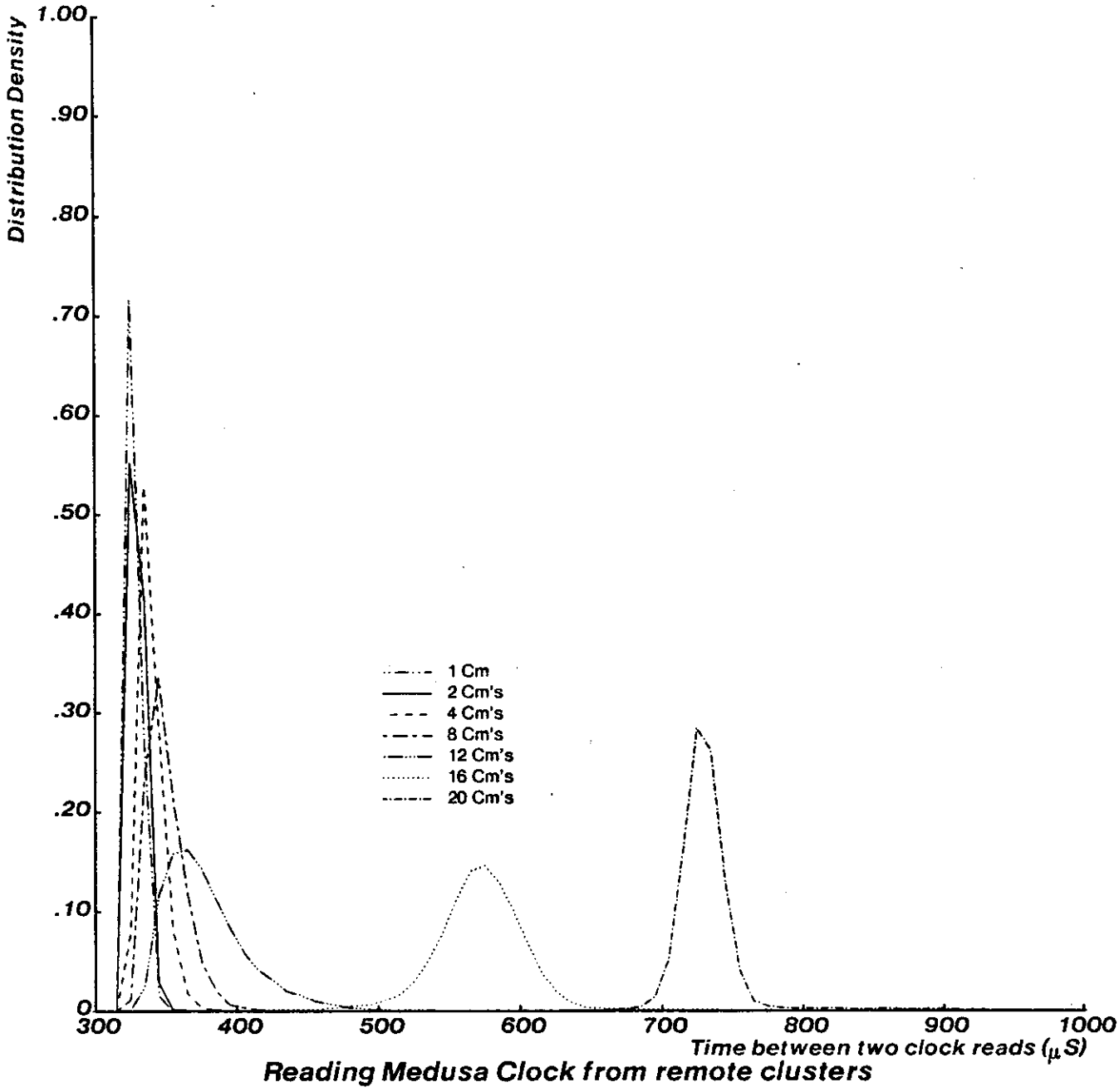

Figure 6: Performance of Medusa 4-Read clock routine

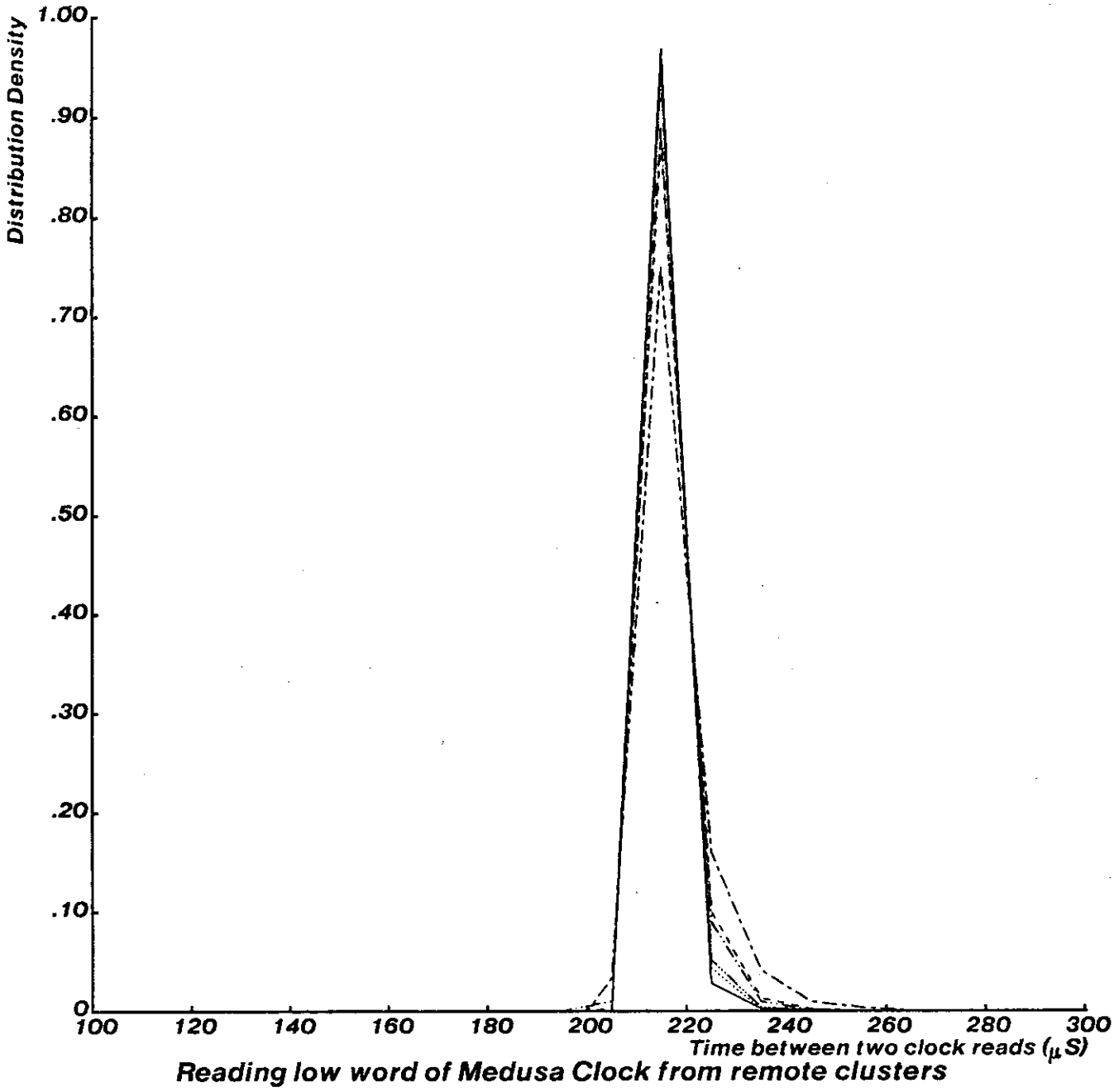

Figure 7: Performance of Medusa 1-Read clock routine

3.3 Conclusion

Comparing Figure 4 with Figure 6, one sees that even though the average execution times are roughly the same at light load, the execution time increases as a function of load faster under StarOS than it does under Medusa. The difference in the shape of the curves in Figure 4 and Figure 6 shows that the two operating systems have very different strategies for handling memory contention. Even when the effects of interrupts are discounted, the StarOS results show a larger standard deviation. The result is that it is more difficult to get accurate time measurements under StarOS. It is also likely that any StarOS experiments using Kmap operations probably have higher variability in execution times.

Although the Cm* global clock is capable of 0.5μS resolution, such resolution is not usable for accurate measurement of time intervals because of the uncertainty in delay involved in an intercluster reference in the presence of load. The results of the clock experiments show that short time intervals (500μS or less) cannot be accurately measured using any of the clock reading routines described.

One way to alleviate the problem is to read only the low order word of the clock. This way, only one LSI-11 instruction is needed to access the clock and the results should be much improved. The problem associated with just reading the low order word is the loss in clock range. With the clock tick set at 2μS, the range provided by the low order word is only about 0.131 second. Larger clock range can be obtained by increasing the clock tick value without sacrificing clock resolution because the usable clock resolution is limited by the uncertainty in communication delay. Therefore, a reasonable value for the clock tick should be commensurate with the uncertainty in communication delay. For example, under very light system load, reading a Medusa clock word has a standard deviation of 2.3μS, and reading a StarOS clock word has a standard deviation of 7.13μS.[2] Therefore, the 2μS resolution of the present clock is useful. However, under heavy loads, the standard deviation of reading a clock word can be very high. Under such loads, the clock tick can be lengthened substantially to increase the range of the clock without losing accuracy.

For any clock that cannot be completely read in one memory cycle, a clock read operation should be provided to latch the clock value and to allow all the clock words to be read indivisibly

[2] Assuming no processor interrupts.

before another clock read operation is accepted. For Cm*, such a feature could be provided by using a hardware latch and some Kmap microcode.

A lesson learned from this study of the clock measurements is that performance measurements must be done very carefully since even the most obvious items such as the clocks can fail to perform as expected.

4 Methodologies for measuring time

Since the inaccuracy of the clock routines for Cm* is mainly due to Kmap load and clock contention, corrective measures can compensate for the incorrect clock readings by correcting for the Kmap load and clock contention during a measurement. Based on this premise, this section discusses methods for obtaining more accurate time measurements for performance evaluation. More specifically, this section proposes a way to generate a repeatable workload for the system on which performance evaluation is done, develops methods for organizing performance evaluation experiments, presents algorithms to compute the net elapsed time given inaccurate clock readings, and tests these methodologies for validity.

4.1 Methodology of performance evaluation

In this project, workloads are synthesized by replicating the measured experiment. For example, to measure the performance of the message facility, the synthetic load will be the number of pairs of processes communicating with each other through messages.

The generation of a synthetic workload can be best illustrated by an example. Assume the execution time of a software routine, R, is to be measured under different system loads. The experiment then consists of a Cm executing R, while a number of other Cm's executing R constitute the synthetic load. The result is the execution time of R as a function of the number of Cm's executing R simultaneously.

A basic approach for measuring performance is to have N identical experiments running in the system. The system workload is parameterized by the value of N and by how the experiments are distributed within the system. A simple way to measure performance is to have only one experiment per cluster reading the clock. This reduces the number of clock reads generated, produces less system load due to fewer clock reads, and results in the improved performance of

both the clock reading software and the measured experiments. The decision to measure only one experiment per cluster is based on the assumption that all the Cm's have identical execution speed, and the symmetry of the Cm* architecture makes Cm's from the same cluster virtually indistinguishable from each other[3]. The validity of the assumption that all Cm's have the same execution speed will be shown in a later subsection.

Also, the timed experiment does not execute continuously. Rather, it is "injected" into the system at fixed intervals. This further reduces the amount of data generated. By injecting the timed experiment after the start up transients have decayed and the system workload has stabilized, more accurate results can be obtained. In the real situation, the user is often interested in finding out the execution time of a piece of software if he were to insert it in a system of a given workload. This situation is quite accurately modelled by the injection approach. Note that the effects of the transients caused by injecting an experiment have not been studied and may be a worthwhile subject for future research.

4.2 Methodologies for measuring elapsed time (Clock compensation)

Given the previously discussed methods for organizing performance evaluation experiments, the next step is to develop algorithms for accurately measuring elapsed time. Early in this section, it was postulated that one can monitor the Kmap workload to improve the accuracy of elapsed time measurements. Below are two such algorithms.

Since reading the clock twice successively yields a result with a mean and variance that are both functions of the system load, the net elapsed time of any experiment can be computed by subtracting the average value of the elapsed time between two clock reads from the measured result. Using this algorithm, the expected value of the computed result equals the true elapsed time, while the distribution of the computed result is identical to the distribution of the measured result. We shall refer to this algorithm as the *long term averaging technique.*

Since the load on the system is a time varying function, and since tasks performed by the system take time to complete, it is reasonable to assume that the system load at times separated by small

---

[3]Actually, some Cm's are connected to I/O devices which may affect their performance. In all experiments, Cm's with I/O devices such as serial lines or Ethernet interfaces must not be used

intervals should be highly correlated. Because the time elapsed between two clock reads is a function of load, the autocorrelation of this elapsed time for short time intervals should also be high. Based on this assumption, the *short term averaging technique* approximates the time required to read the clock during an experiment by using the elapsed time between two successive clock reads that occur closely in time. Below is a mathematical analysis of the short term averaging technique.

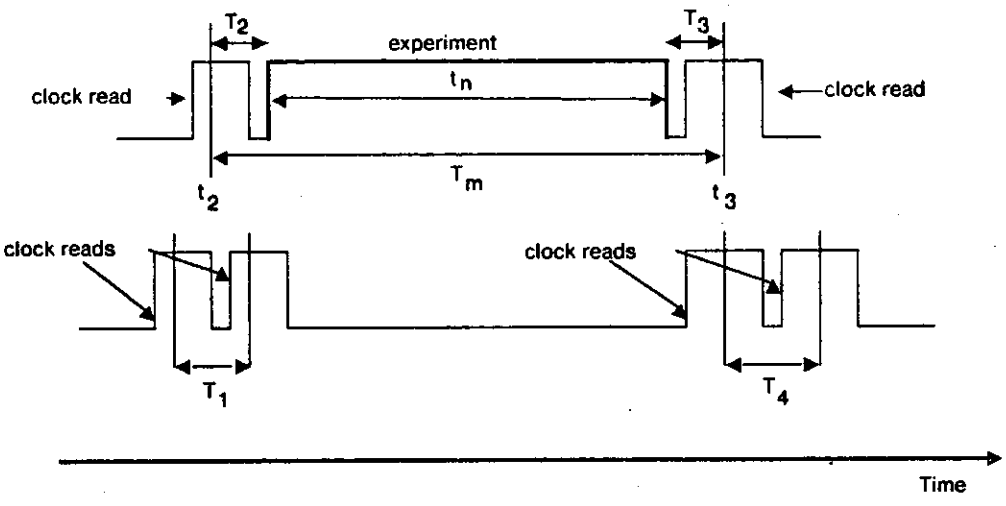

Figure 8: Short term averaging algorithm

Assume we are measuring the execution time of an experiment as illustrated in Figure 8. Then the variables are defined as follows[4]:

- t_n is the true elapsed time of the experiment.
- T_A is the computed elapsed time. (T_A approximates t_n.)
- T_m is the measured elapsed time.
- T_2 is the time interval between the moment the clock is read and the moment the experiment begins.
- T_3 is the time interval between the moment the experiment ends and the moment the clock is read again.
- T_1 and T_4 are the elapsed times between two pairs of clock reads.

Then $t_n = T_m - (T_2 + T_3)$ and

[4]Capital letters denote random variables

$$T_A = T_m - (T_1 + T_4)/2. \quad (1)$$

therefore

$$T_A = t_n + T_2 + T_3 - (T_1 + T_4)/2.$$

If the expected values

$$E(T_1) = E(T_4) = E(T_2 + T_3) = \delta,$$

then

$$E(T_A) = t_n + \delta - (\delta + \delta)/2 = t_n.$$

Therefore, the expected value of the computed result equals the true result.

The variance of the computed result is:

$$V(T_A) = V(t_n + T_2 + T_3 - (T_1 + T_4)/2).$$

Let

$$V(T_1) = \sigma_1^2,\ V(T_4) = \sigma_4^2,$$
$$V(T_2) = \sigma_2^2, \text{ and } V(T_3) = \sigma_3^2,$$

then[5]

$$\begin{aligned} V(T_A) = {} & \sigma_2^2 + \sigma_3^2 + \sigma_1^2/4 + \sigma_4^2/4 \\ & + 2\rho_{T_2,T_3}\sigma_2\sigma_3 - \rho_{T_1,T_2}\sigma_1\sigma_2 - \rho_{T_1,T_3}\sigma_1\sigma_3 \\ & - \rho_{T_2,T_4}\sigma_4\sigma_2 - \rho_{T_3,T_4}\sigma_4\sigma_3 + \rho_{T_1,T_4}\sigma_1\sigma_4/2, \end{aligned} \quad (2)$$

where ρ_{T_i,T_j} is the correlation coefficient of the random variables T_i and T_j.

To simplify Equation (2), the following assumptions are made:

$$E(T_2) \simeq E(T_1)/2,\ E(T_3) \simeq E(T_4)/2,\ V(T_1) = V(T_4) = \sigma^2. \quad (3)$$

Now,

$$\begin{aligned} V(T_A) = {} & \sigma^2[1 + 0.5\rho_{T_2,T_3} - 0.5\rho_{T_1,T_2} - 0.5\rho_{T_3,T_1} - 0.5\rho_{T_2,T_4} \\ & - 0.5\rho_{T_3,T_4} + 0.5\rho_{T_1,T_4}]. \end{aligned} \quad (4)$$

If all correlation coefficients are unity, then

$$V(T_A) = 0.$$

In the worst case when $\rho_{T_1,T_2} = \rho_{T_3,T_1} = \rho_{T_2,T_4} = \rho_{T_3,T_4} = 0$, and $\rho_{T_2,T_3} = \rho_{T_1,T_4} = 1$,

$$V(T_A) = 2\sigma^2$$

for all non-negative correlation coefficients.

---

[5] If $X = \sum_{i=1}^{n} a_i Y_i$, then $V(X) = \sum_{i=1}^{n} a_i^2 V(Y_i) + 2\sum\sum_{i<j} a_i a_j \rho_{i,j}\sigma_i\sigma_j$.

For very short interval measurements, the time-stamps t_2 and t_3 are very close together and the random variables T_2 and T_3 can be replaced by a new T_3 equals to the old T_2 + T_3. Then $\sigma_2^2 = 0$ and $\sigma_3^2 = \sigma^2$. We now have

$$V(T_A) = 1.5\sigma^2 - \rho_{T_1,T_3}\sigma^2 - \rho_{T_3,T_4}\sigma^2 + 0.5\rho_{T_1,T_4}\sigma^2. \quad (5)$$

If all correlation coefficients are unity, then

$$V(T_A) = 0.$$

If all correlation coefficients are zero, then

$$V(T_A) = 1.5\sigma^2.$$

In the worst case when $\rho_{T_1,T_3} = \rho_{T_3,T_4} = 0$, and $\rho_{T_1,T_4} = 1$,

$$V(T_A) = 2\sigma^2$$

for all non-negative correlation coefficients.

Even though Equation (2) expresses the value of the variance of the result, it cannot be solved unless the variances of T_2 and T_3 are known. In our case, this information is not available from the experiment. To simplify the problem, the time elapsed between the issuing of a clock read to the actual reading of the clock and the time elapsed between the reading of the clock to the returning of the result to the reader are assumed to have the same mean and variance. Equation (4) then expresses the variance of the result. Equation (5) applies when the duration of the experiment to be measured is extremely short.

This algorithm shows that for any method used to select the two pairs of clock reads, the worst case will yield a result with a variance twice the variance of the elapsed time between two clock reads. In the best case, the variance of the result is zero. In cases where assumptions of Equation (3) apply, zero variance in the result is obtained when the correlation coefficients between T_1 and T_2 (ρ_{T_1,T_2}) and between T_3 and T_4 (ρ_{T_3,T_4}) are both unity.

A way to evaluate the methods used to select the clock read pairs is to compute the improvement factor k. In any experiment that measures a fixed time interval, let $V(T_A)$ be the variance of the corrected result and let σ^2 be the variance of the uncorrected result. Then k is defined such that

$$k = \sigma^2 / V(T_A).$$

The larger the value of k is, the better the improvement. The range of k is between 0.5 and infinity. When k is unity, the variance of the corrected result is unchanged. Note that the Long Term Averaging algorithm always yields a k of unity.

The objective for selecting the two pairs of clock reads for compensation is to maximize the correlation coefficients ρ_{T_1,T_2} and ρ_{T_3,T_4}. Though there are many ways this can be done, only two methods will be presented. Readers are encouraged to design their own implementations, bearing in mind that the objective is to maximize the correlation coefficients stated above.

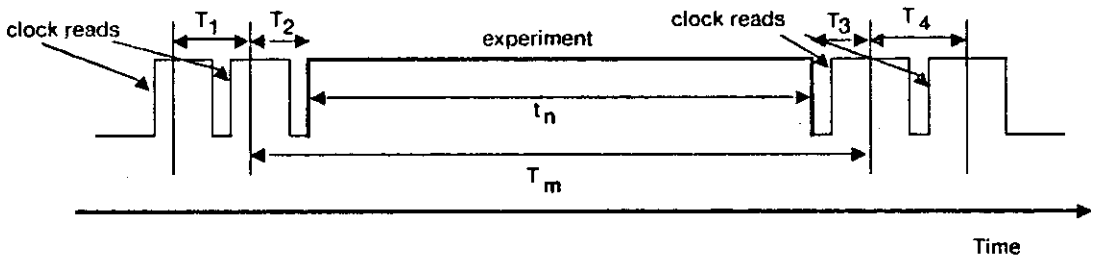

Figure 9: Short term averaging, Method I

The first method presented, hereafter referred to as Method I, is illustrated in Figure 9. If the processor that starts the measurement reads the clock twice at the beginning of the experiment and the processor that terminates the measurement reads the clock twice after the experiment, then T_1 should highly correlate with T_2 while T_3 should highly correlate with T_4. This is because these clock reads occur very closely in time.

This method has the advantage that no clock read occurs during the experiment and therefore the performance of the experiment under measurement is not affected.

The second method presented is referred to as Method II, as illustrated in Figure 10. If a clock process runs concurrently with the experiment and periodically samples the load by reading the clock twice in succession, the elapsed time can be used for compensation. One approach is to select the clock read pair of the clock process that is closest in time to the clock read that starts the measurement to give T_1, and to select the clock read pair of the clock process that is closest in time to the clock read that stops the measurement to give T_4.

This method is less desirable than the previous method because of its added complexity and because of its interference with the performance of the experiment by the presence of a clock reading process. However, subsequent sections will show that this method yields quite accurate results.

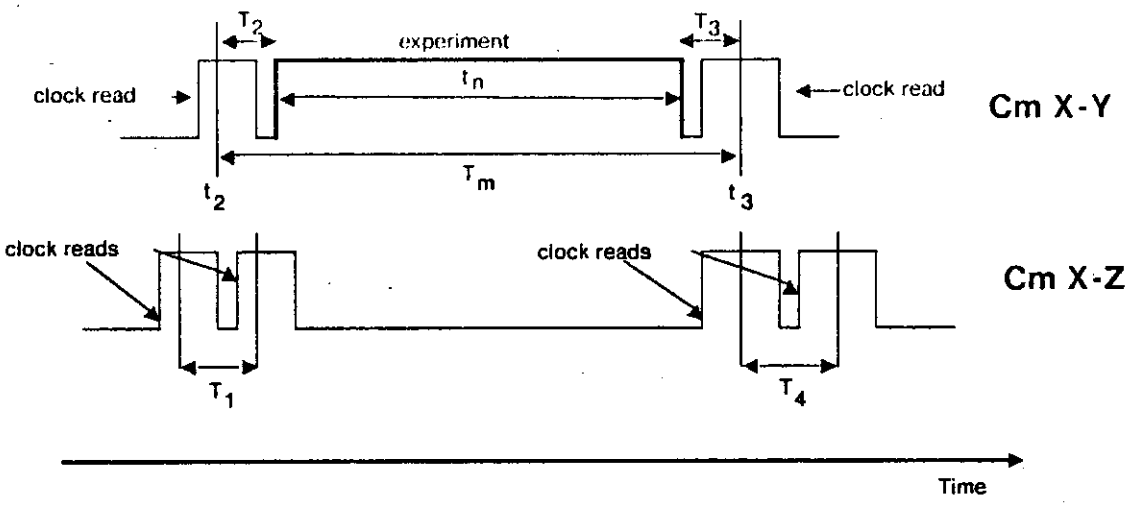

Figure 10: Short term algorithm, Method II

4.3 Execution speed of computer modules

The method for generating workload assumes that all the computer modules in Cm* execute at the same speed. To verify this assumption, an experiment was set up to measure the execution speed of the computer modules in Cm*. This experiment involved timing the execution of a piece of code stored in the local memory of each Cm. Once the program execution begins, the Kmaps are not involved.

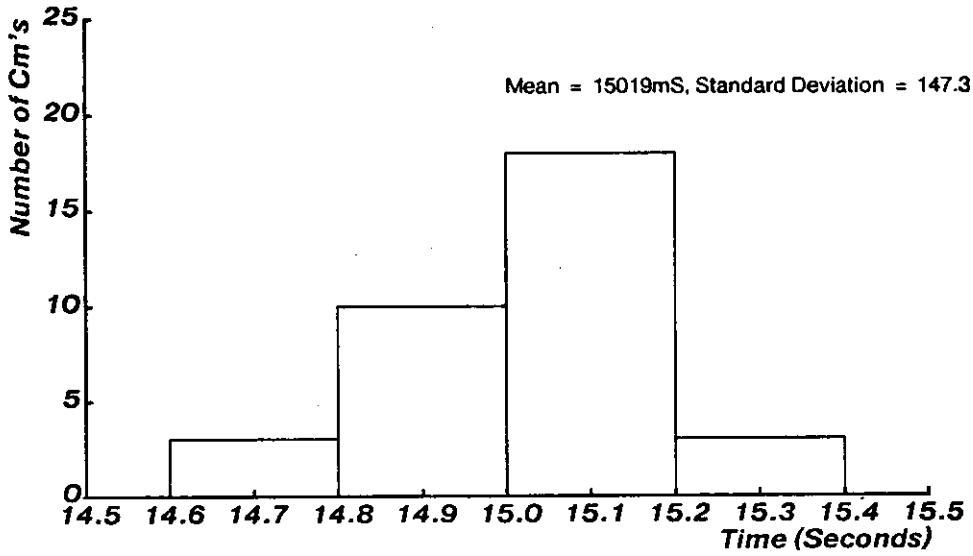

Figure 11: Histogram of execution time of 34 Cm's

The results show that all of the thirty-four computer modules tested had execution speeds within 4.6% of each other. A histogram of the execution speed of the computer modules is shown in Figure 11. The conclusion for this experiment is that every computer module can be considered to have essentially identical execution speed, therefore experiments performed on any computer module should be equally valid.

4.4 Evaluation of clock reading compensation techniques (Method I)

The methods to compensate the clock readings cannot be rigorously proved to produce correct results because they employ only heuristic approaches. Therefore, to validate our methods, an attempt is made only to show that an accurate result for a fixed interval measurement (e.g., 0 seconds) is obtained under some reasonable system load.

The experiment to validate Method I consists of a process reading the clock four times in succession. The first two clock reads are used to compute T_1, the second and third clock reads measure a null experiment which has zero execution time. The third and fourth clock reads are used to compute T_4. The synthetic workload is generated by replicating a large number of processes distributed evenly among the clusters reading the system clock. The experiment was performed for both StarOS and Medusa.

Figure 12 illustrates the result of the experiment using the StarOS 4-Read clock routine to measure time. The solid curve is the distribution density of the compensated result, while the dashed curve is the distribution density of the result before correction is applied. The ideal result is an impulse of unit magnitude at 0μS. The mean compensated result was -1.90μS, and the improvement factor, k, was 0.8. Recall that for $k < 1$, the variance of the result is increased. The same experiment using the 1-Read clock routine gave an improvement factor of 0.81.

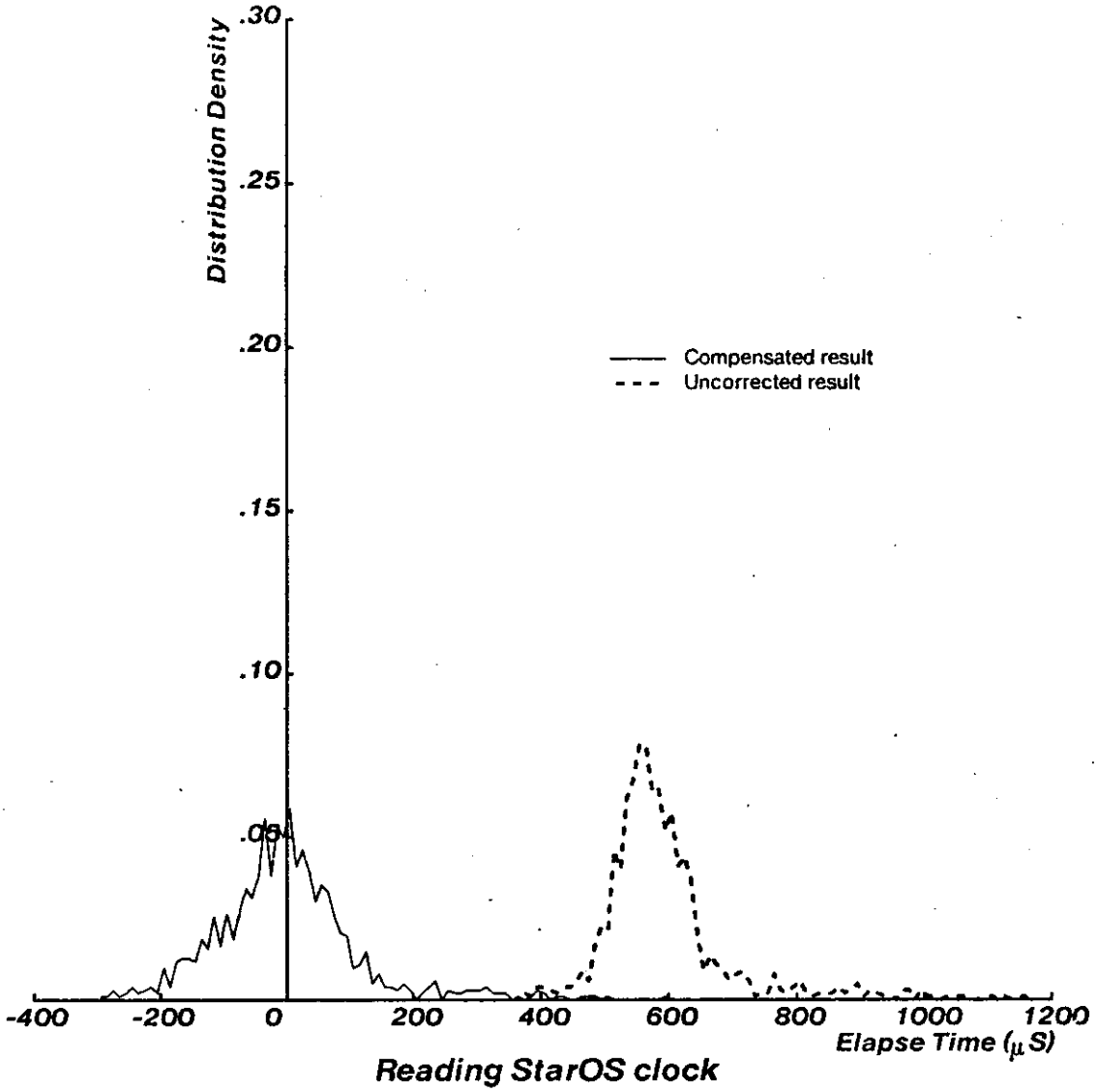

Figure 12: Measuring zero elapsed time using Method I with 4-Read routine

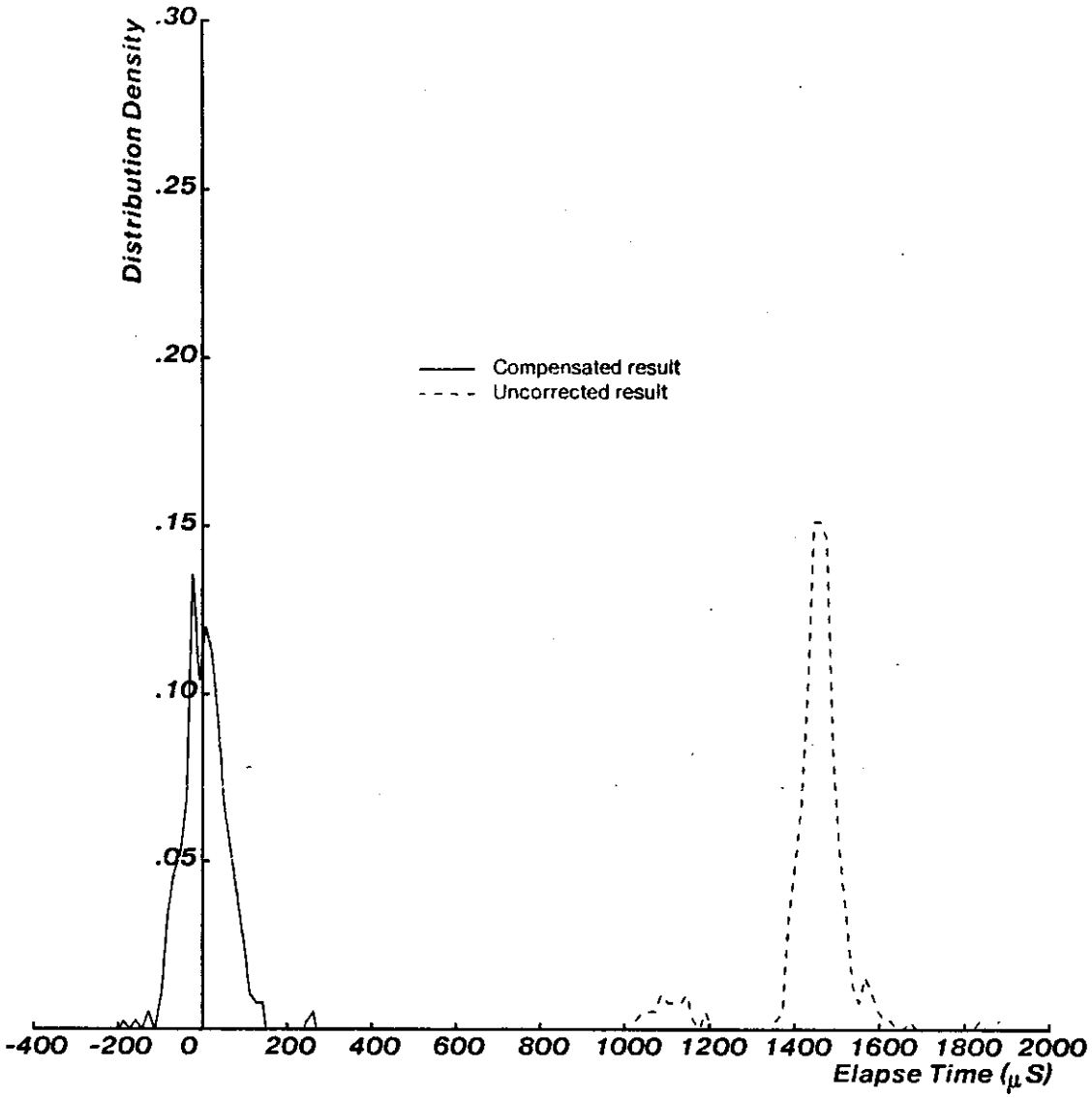

Figure 13: Measuring zero elapsed time using Method I with Medusa 4-Read routine

Figure 13 illustrates the result under Medusa using the 4-Read clock routine to measure time. The mean compensated result was 6.69μS, and the improvement factor k was 3.57. This represents a great improvement in the variance of the results. The 1-Read clock routine gave an improvement factor of 0.68.

4.5 Evaluation of clock reading compensation techniques (Method II)

The experiment that validates Method II consists of a clock process executing in a computer module from the cluster where the experiment is performed, a process that does two successive clock reads to measure the elapsed time (which should ideally be zero if reading the clock does not take any time), and a number of pairs of communicating processes that send each other messages to create a synthetic system workload. Each pair of these communicating processes is independent of the other processes in the system, and their sole purpose is to generate load to the Kmaps through which clock read requests are routed. The experiment process measuring zero elapsed time is synchronized with the clock process. It signals the clock process to start reading the clock, reads the clock twice successively, and then sends the results of the two clock reads to the clock process which computes the net elapsed time according to Equation (1).

Figure 14 shows the distribution density of the results of the StarOS experiment. The dashed curve is the result of the measured reading (T_m in Equation (1) and Figure 10). The solid curve is the result after Method II has been applied (T_A in Equation (1)). The results were taken from 1000 repetitions of the experiment. The mean value was -5.24μS, while the improvement factor k was 1.14. The improvement factor for the 1-Read clock routine was 1.98.

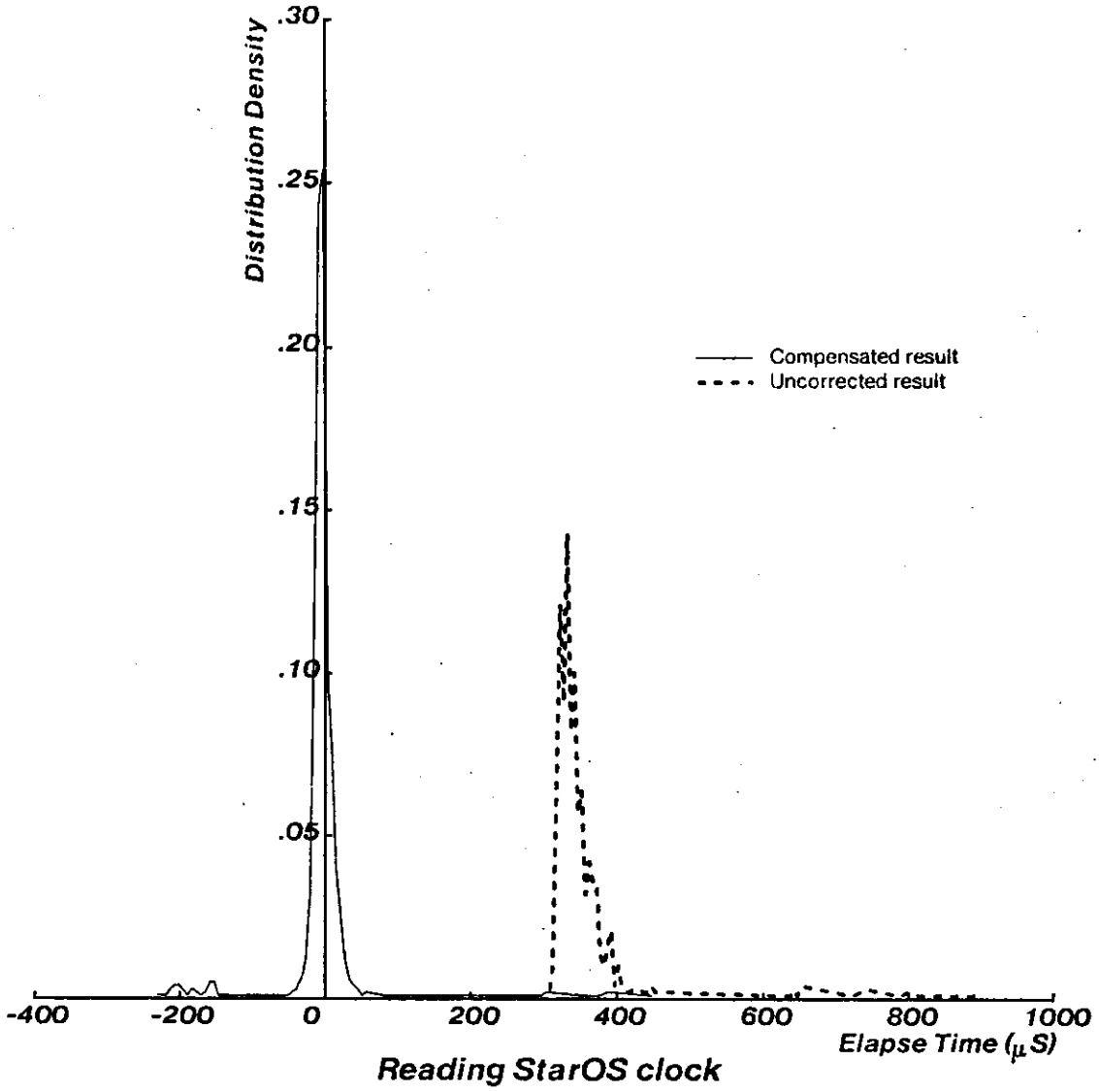

Figure 14: Measuring zero elapsed time using Method II with StarOS 4-Read routine

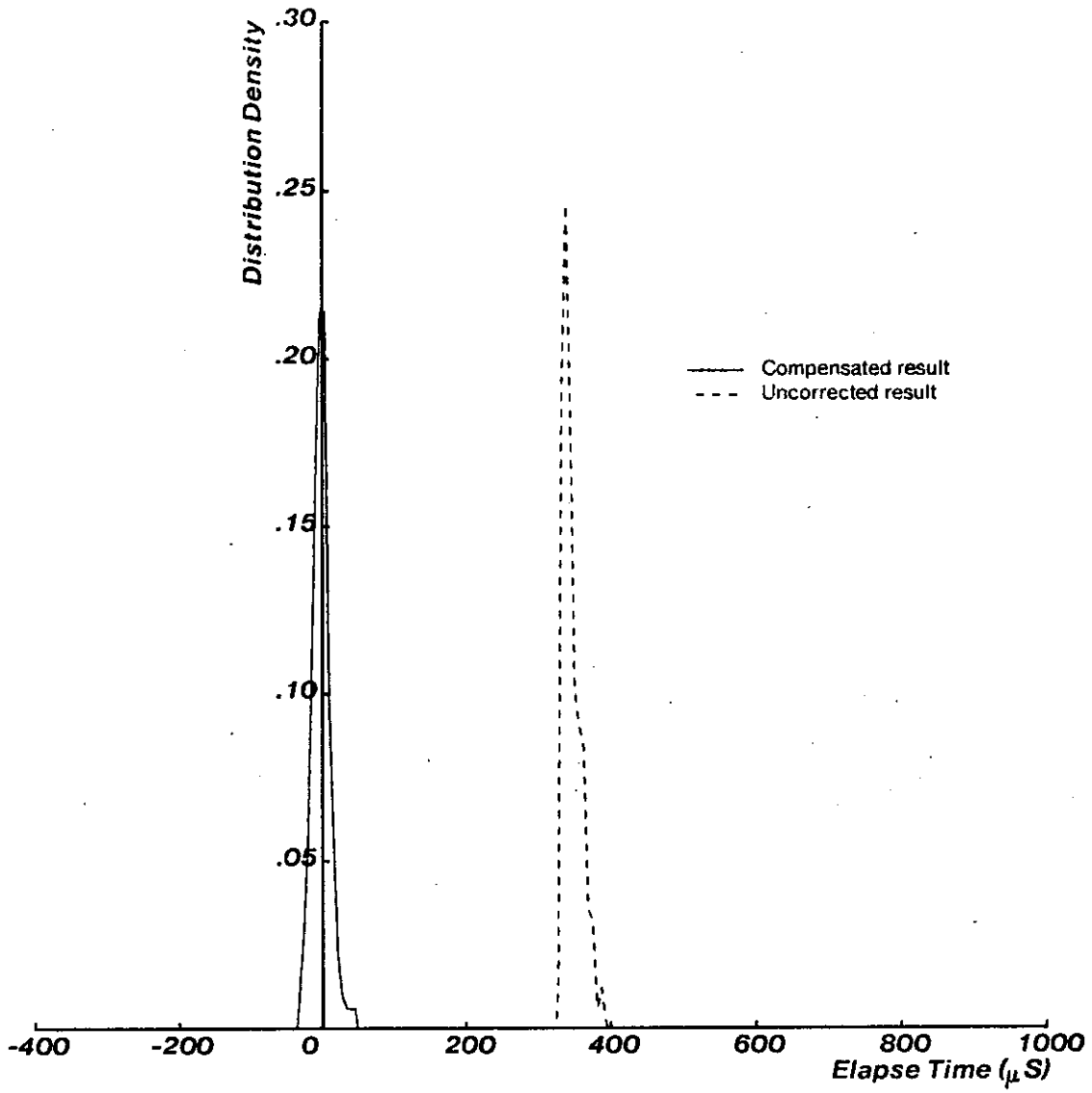

Figure 15: Measuring zero elapsed time using Method II with Medusa 4-Read routine

When executing under Medusa, the experiment yields different results. Figure 15 shows the distribution density of the Medusa experiment. The mean value of the compensated result was 6.69μS and the improvement factor was 1.11. The improvement factor was 0.81 for the 1-Read clock routine.

4.6 Discussion of results

As shown by the result of the above eight experiments, the mean corrected value was less than 6.7μS. This provides an upper bound to the accuracy of measurement obtainable. It is concluded that these measurement methods are not suitable for measuring elapsed times that are less than fifty microseconds because the relative error for small interval measurements is high.

Of the eight experiments performed, four showed improvement in the variance of the corrected result (with the improvement factor k ranging from 1.11 to 3.57). The other four cases showed a k less than 1 but greater than 0.67. Recall in Equation (4), it was shown that the worst case k would be 0.5, while if the clock reads used for compensation were totally uncorrelated to the clock reads that they were supposed to compensate, the value of k would be 0.67. In the Medusa experiment using Method I with the 1-Read clock routine, the value of k was 0.68. This shows that the system load was changing so rapidly during the experiment that the execution time of any clock read was essentially uncorrelated to the execution time of any previous or subsequent clock reads.

It is interesting to note that three out of the four experiments using Method II resulted in improved variance, while only one out of the four experiments using Method I resulted in the improved variance. This phenomenon is mostly due to the difference in the type of system load. In all the experiments using Method II, the system workload was the load created by a large number of processes sending and receiving messages. Since sending and receiving of messages are lengthy processes (on the order of a millisecond), the load of the system is trackable by the clock reads. When the granularity of the system load decreases to a duration comparable or shorter than the time required to execute a clock read, the tracking of the system load using clock reads fails. This was the case for the experiments to validate Method I. The synthetic workload was a large number of processes reading the clock. Because the load on the system had the same duration as the clock reads used to sample the load on the system, the tracking of the system load failed.

A problem with Method I is that it does not track system load correctly when an interrupt occurs. This is because an interrupt during an experiment will either affect the clock read used to obtain the measurement or the clock read used to obtain the compensator, but not both. This explains why Method I did not work very well under StarOS since the StarOS processes were interrupted sixty times per second. Method II tracks well even with interrupts. This is because interrupts by the line time clocks are system-wide, therefore an interrupt affecting the clock read

used to obtain the measurement is likely to occur in the clock process as well. This means that the extra time required to handle an interrupt during a clock read is likely to be compensated for.

4.7 Conclusion

In this section, methods have been developed to measure the performance of Cm* software under system load. Two algorithms have been developed to yield more accurate elapsed time measurements than the clock routines can provide.

Experiments were performed to validate the measurement methodologies. The variation of execution speed among different Cm's was found to be around 4.6%. The long term algorithm developed to compensate for clock readings has a very predictable behavior and no experiment was performed to test its validity. The short term algorithm was implemented with two variations - Method I and Method II. Experiments were set up to evaluate both methods. The base line accuracy of these methods was around 6.7μS. Therefore these methods are not suitable for measuring short duration events (50μS or less), but are perfectly suitable for measuring longer duration events such as operating system calls.

Because Method I was theoretically superior to Method II, it was given the tough task of executing under system load of very small granularity. Results showed that Method I was unable to perform properly in small granular system loads. Method II was tested with a more reasonable load and was found to perform quite well. The short term algorithm using Method II performed better than the long term algorithm would have performed in three of the four experiments tested. Because Method I is more desirable than Method II in that it does not affect the experiment under measurement, it is believed that Method I should perform at least as well as Method II under a reasonable system load provided that there are no interrupts. When there are interrupts, Method II is the preferred method.

An important conclusion is that it is not possible to present a clock compensation scheme that works under arbitrary system load because the clock readings can only sample the system load at a finite rate. Readers are encouraged to develop their own clock compensation technique. However, they must test their scheme to ensure that it tracks the system load reasonably well. The zero elapsed time measuring experiment is recommended for such testing. Below is the procedure for testing a clock compensation method:

```
DO
      Pick a clock compensation method;
      DO
          Experimentally, measure zero elapsed time;
          IF result not satisfactory THEN
              Fine tune the method;
      UNTIL method is optimal or results satisfactory;
UNTIL exhausted all methods or results satisfactory;
```

5 An example experiment

This experiment evaluates two performance measures of a message-based operating system. The first measure is the latency of the message mechanism. Latency is defined as the time elapsed from the moment a sender begins to send a message to the moment the receiver receives the message. The second measure is the execution time of a message-based remote procedure call (RPC).

While a message mechanism is often provided by an operating system as a primitive for interprocess communication (IPC), remote procedure calls using request/response protocols are often layered on the basic mechanism [Nelson 81] [Spector 82] [Birrell and Nelson 84]. The remote procedure call of this experiment consists of a *client*, which sends a message containing the arguments for the call, and a *server*, which receives the message, performs the specified function, and returns a message containing the result. The time elapsed from the moment the client begins to send a message to the moment it finishes accessing the returned result constitutes the execution time of the RPC. While this RPC implementation does not do type checking or error recovery, it is a simplified model of RPC that can indicate the RPC performance of a system.

5.1 Organization of experiment

The experiment consists of one or more client/server pairs. Below is the pseudo-code for such a client/server pair:

```
client:                                  server:
       prepare argument
       T1
       send the arguments ------------------> wait for a message
                                              T2
                                              access parameters
                                              perform computation
       wait for the results <---------------- send results
       access results
       T3
```

The latency of a message is T2 - T1, while the execution time of the RPC is T3 - T1. For the latency measurement to be meaningful, the server must be blocked before the client sends a message.

The experiment is implemented under StarOS and consists of a master process which spawns client/server pairs in locations specified by the user. Since both processes of a client/server pair reside in the same cluster, all communications within a client/server pair are intracluster. Of all the client/server pairs spawned, only one pair reads the clock to measure performance. This pair is responsible for sending all its results to the master process. The master process ships the results to a VAX/UNIX system via the Ethernet for storage and analysis. The clock compensation technique used is Method II of the short term averaging algorithm, as presented in Section 4.

5.2 Experiments

The experiment was performed with different levels of load ranging from one client/server pair per cluster to three client/server pairs per cluster. The total number of words accessed by the client and server varied from 0 to 200 in increments of 50. All measurements were repeated 512 times.

The measurement of zero elapsed time was performed during each repetition of the experiment as a run-time check to see how well the clock compensation scheme was tracking the system load. It was found that the mean error was no worse than 4.9μS, while the improvement factor was between 0.74 and 0.78. This low improvement factor was due to interrupts that were not trackable by Method I. By simulating the situation that there are no interrupts, the improvement factor was between 1.01 and 1.22.

5.3 Results

5.3.1 Latency measurements

Figure 16 illustrates the latency of the StarOS messages in our experiment. The solid curve shows that latency is constant at approximately 7050μS and is independent of the total number of message words accessed. The dashed curve shows that when two client/server pairs are executing in the same cluster, the latency rises to 7960μS because of increased Kmap load. As the total number of message words accessed increases, more time is spent accessing remote memory, resulting in fewer RPC executions per unit time. This causes a decrease in the rate of message

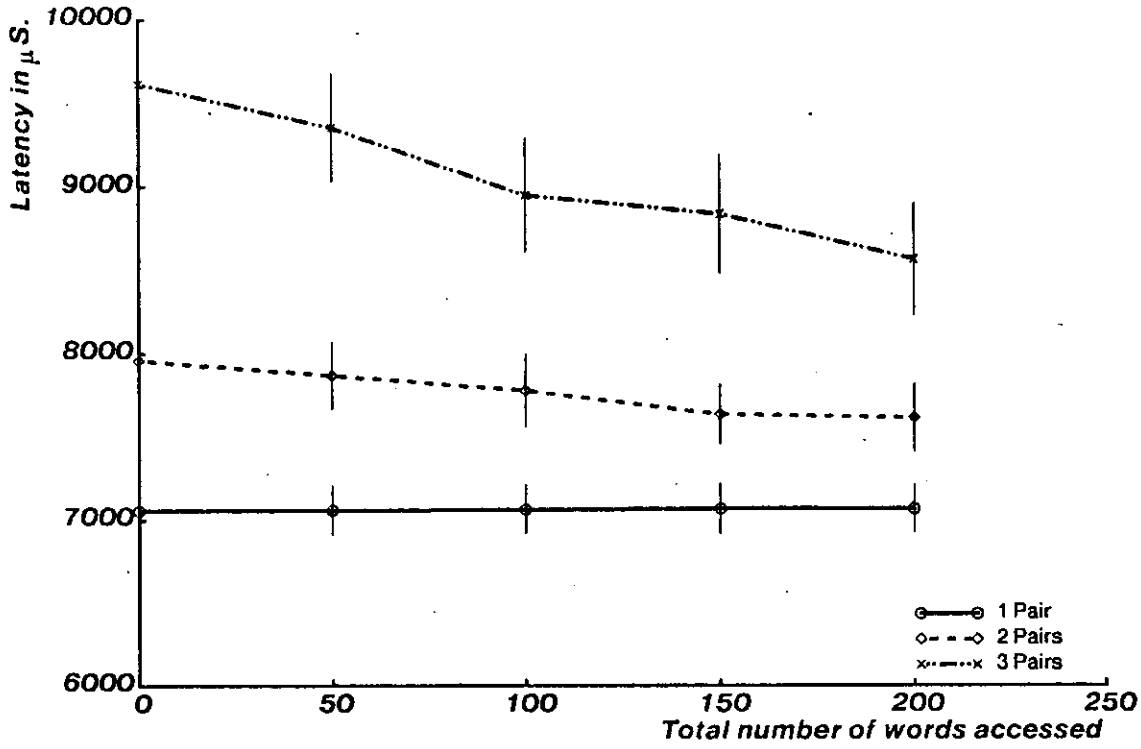

Figure 16: Latency of StarOS messages in the experiment

operations, resulting in a decreased latency. The broken curve shows that latency is 9615μS when three client/server pairs are executing in the same cluster. The vertical bars at each point of the curves show the magnitude of the standard deviation at that point.

5.3.2 Execution time of RPC

Figure 17 shows the execution time of the remote procedure call as a function of the total number of words (0 to 200) accessed by the client and the server. The execution time of the call is a linear function of the number of words accessed. In the solid curve, the slope is 49.9μS per word[6].

---

[6]This value must not be interpreted as the intracluster memory access time for StarOS. Rather, it is the time required for an iteration through the following Bliss-11 program loop:

```
INCR k FROM 0 TO .lreused - 1 DO
        temp = .ResultPage[.k];
```

This loop compiles into six LSI-11 instructions, with one of them performing a remote memory reference.

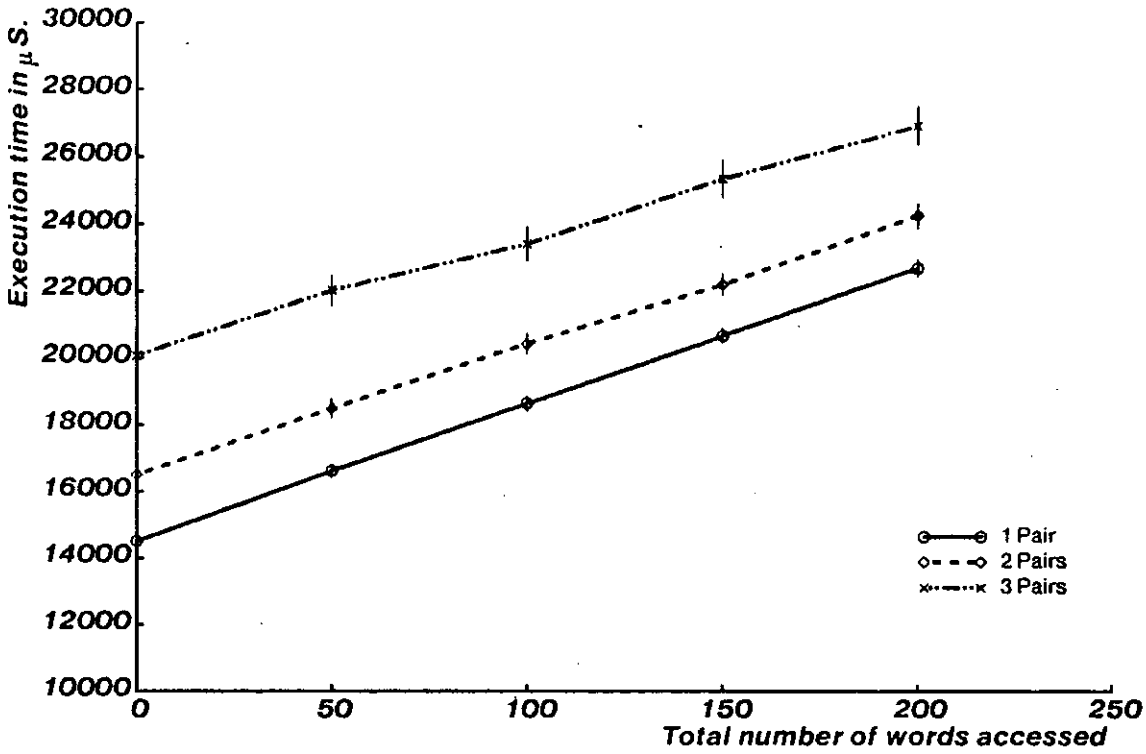

Figure 17: RPC execution time versus the total number of words accessed

5.4 Conclusion

This section presented an example of measuring StarOS message latency and message-based RPC execution time. The example experiment implemented two ideas developed in Section 4. First, it generated system workload by replicating the experiment in different parts of the system. The number of replicas and how they were distributed in the system were both controlled by the experimenter at program run time. Second, the experiment employed one of the clock compensation techniques developed in Section 4. The addition of the clock compensation technique into the basic experiment required only the addition of a subroutine which computed net elapsed times given four time-stamps.

6 Conclusion

This project discovered the erratic behavior of the Cm* clock reading software and presented an alternate set of clock reading software. Additionally, it recommended that the length of a clock tick should be set to be commensurate with the variation in communication delays. It proposed that a

Kmap operation be provided to latch the clock value and to read the clock register indivisibly. Most important of all, it provided a clock compensation scheme for measuring elapsed time with an accuracy much greater than that provided by the clock reading software.

However, in developing the mathematical model of the clock compensation techniques, it was assumed that the correlation between the time elapsed for two successive clock reads and the system load was non-negative, and that the autocorrelation of system loads separated by short time intervals was non-negative. The validity of these assumptions should be investigated. Also, the clock compensation technique fails when the granularity of system load is too small. At present, little is known about the granularity and the time profile of system load. Further study is required to gain this knowledge.

Of more general concern, this project stressed the importance of real-time clock designs in multiprocessors since they greatly affect the clock's usability. Future multiprocessor designs should include a globally readable clock that is accessed through a special bus unaffected by system load and contention. An example of such a clock is the system clock of C.mmp multiprocessor. When such a design is not feasible, the clock should be of the same width as the data bus so that the entire clock word can be read in one access. If this is not practicable, there should be an instruction that latches the clock value and reads it indivisibly. The implementation of such an instruction should be straightforward when the clock register size is compatible with one of the machine supported data types, since multiprocessor should allow indivisible read/write accesses to all machine supported data types to guarantee data consistency. Thus, the instruction in essence is simply a latch operation followed by a read operation.

For multiprocessor systems using a single global clock and experiencing the same problems experienced by Cm*, this project provided a general scheme for measuring elapsed time accurately. The study of Cm* clock performance shows that system load can be gauged by reading the system clock. This is because reading the clock exercises many of the system resources (Map-bus, intercluster bus, Kmaps, etc.). This idea of exercising system resources to measure their load is worth exploring.

References

[Birrell and Nelson 84]
Andrew D. Birrell, Bruce J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):38-59, 1984.

[Ellingson 73] C. Ellingson & R. J. Kulpinski.
Dissemination of System Time.
IEEE Trans. Comm. Com. 23(5):605-624, May, 1973.

[Ferrari 78] D. Ferrari.
Computer Systems Performance Evaluation.
Prentice Hall, 1978.

[Gehringer 81] E. F. Gehringer and R. J. Chansler, Jr.
StarOS User and System Structure Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1981.

[Jones 79] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans, S. R. Vegdahl.
StarOS: A Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings of the 7th Symposium on Operating Systems.* Asoilomar Grove, CA, December, 1979.

[Jones 80] A. K. Jones and E. F. Gehringer, editors.
The Cm Multiprocessor Project: A Research Review.*
Technical Report, Carnegie-Mellon University, Computer Science Department, July, 1980.

[Kong 82] Thomas H. Kong.
Measuring Time for Performance Evaluation of Multiprocessor Systems.
November, 1982.
Masters Project Carnegie-Mellon University Department of Electrical Engineering.

[Lamport 78] Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[Marathe 77] Madhav V. Marathe.
Performance Evaluation at the Hardware Architecture Level and the Operating System Kernel Design Level.
PhD thesis, Carnegie-Mellon University Computer Science Department, December, 1977.

[Marzullo 83] K. Marzullo, S. Owicki.
Maintaining the Time in a Distributed System.
In *Proceedings of the 1983 Principles of Distributed Computing Confererence.* Montreal, Canada, August, 1983.

[Nelson 81] Bruce Jay Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University Computer Science Department, May, 1981.

[Ousterhout 80a] J. K. Ousterhout, D. A. Scelza and P. S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2), February, 1980.

[Ousterhout 80b] John K. Ousterhout.
Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.
PhD thesis, Carnegie-Mellon University, Computer Science Department, April, 1980.

[Raskin 78] Levy Raskin.
Performance Evaluation of Multiple Processor Systems.
PhD thesis, Carnegie-Mellon University Department of Electrical Engineering, August, 1978.

[Ritchie 74] D. M. Ritchie and K. Thompson.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[Spector 82] Alfred Z. Spector.
Performing Remote Operations Efficiently on a Local Computer Network.
Communications of the ACM 25(4), April, 1982.

[Wulf 81] William A. Wulf, Roy Levin, and Samuel P. Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.