Alternative Majority-Voting Methods for Real-Time Computing Systems

Kang G. Shin Senior Member IEEE
The University of Michigan, Ann Arbor
James W. Dolter
The University of Michigan, Ann Arbor

Key Words **— Error masking, Synchronous voting, Asynchronous voting, Quorum majority voting, Compare majority voting, Real-time architecture**

Reader Aids —
Purpose: Present an alternative to existing techniques.
Special math needed for explanations: None.
Special math needed to use results: None.
Results useful to: Computer architects and real-time system designers.

Summary & Conclusions **— Despite the increasing need for ultra-reliable computers for use in distributed environments, there has been no significant advances to reduce the time overhead of combining replicated data in conjunction with providing instantaneous error masking.**

We propose two techniques that provide a compromise between the high time overhead in maintaining synchronous voting and the difficulty of combining results in asynchronous voting. These techniques are specifically suited for real-time applications with a single-source/single-sink structure that need instantaneous error masking. A description of the techniques, their underlying system requirements, and possible time bounds are presented.

Two techniques provide a compromise between a tightly synchronized system in which the synchronization overhead can be quite high, and an asynchronous system which lacks suitable algorithms for combining the output data. Both QMV and CMV are most applicable to distributed real-time systems with single-source/single-sink tasks. All real-time systems eventually have to resolve their outputs into a single action at some stage. The development of the Advanced Information Processing System (AIPS) and other similar systems serve to emphasize the importance of this class of real-time systems and the possible applications of these techniques. Time bounds suggest that it is possible to reduce the overhead for quorum-majority voting to below that for synchronous voting. All the bounds assume that the computation phase is non-preemptive and that there is no multi-tasking.

1. INTRODUCTION

The need for ultra-reliable computers in support of real-time control applications has greatly increased in the past few years and is anticipated to continue growing at an even faster pace. Traditionally, errors in computing systems have been masked by replicating the application and system tasks, distributing these tasks on independent hardware, and combining the replicated results. Two notable methods used to combine the replicated results are:

- A *synchronous vote* applied to the replicated output
- Systems which combine the output in an *asynchronous* fashion.

Both of these methods have their advantages and disadvantages. The former lends itself to simpler implementation but requires synchronization overhead, whereas the latter takes full advantage of the truly asynchronous nature of distributed environments but lacks application independence.

The Fault Tolerant Multi-Processor (FTMP) [1] and Fault Tolerant Processor (FTP) [2] along with the Software Implemented Fault Tolerance (SIFT) computer [3] are examples of implemented architectures that combined replicated results using a synchronous vote. The major differences in these architectures were the choice of where to place and how often to perform the synchronous vote. The designers of FTMP and FTP choose to vote on information entering the processors from replicated buses, thus allowing the vote to be performed in hardware transparent to the applications software. This required the processors to be in tight synchronism and, thus, cannot be extended to a distributed environment. For SIFT a time-frame structure was used in which messages were exchanged and a vote performed. This time-frame placed restrictions on the structure of the application software and, thus, greatly hampers application independence. In all of these architectures there is a rendezvous point: the bus cycle for FTMP and FTP, and the task frame for SIFT, in which the replicated data are combined by performing a bit-for-bit majority vote on the output data stream.

Another approach [4, 5] is to have the replicated tasks run asynchronously and vote on the results. This approach, though conceptually similar to the earlier analog control circuits, suffers from the problem of specifying and implementing suitable algorithms to derive a single output request from the multiple output requests produced by the replicated tasks. These requests can differ since the processors run asynchronously with respect to one another and sample the input independently. Once an algorithm is developed, the resulting systems usually require great care for even minor modifications and, thus, are highly application dependent.

We propose two techniques, *quorum-majority* voting (QMV) and *compare-majority* voting (CMV) that incorporate some of the advantages of both asynchronous and synchronous voting. These techniques are intended for the class of problems whose error containment and timing requirements dictate distributed system solutions with replicated tasks.

Section 2 presents the problem along with our initial assumptions. Section 3 presents the required architecture and

0018-9529/89/0400-0058$01.00©1989 IEEE

an assumed software structure present in the real-time applications. Section 4 presents QMV and CMV. Section 5 focuses on the derivation of the time bounds that are possible using these methods.

Future work should concentrate on relaxing the assumptions concerning: the computation phase being non-preemptive, no multi-tasking, and the single-source/single-sink I/O structure.

2. PROBLEM STATEMENT AND APPROACH

Our primary goal is to provide an environment in which real-time applications interfaced to life critical functions can be executed correctly in the presence of malfunctioning hardware. Correct operation not only involves masking up to a given number of faults but satisfying the real-time constraints (deadlines) of the applications. Thus, the worst case time bounds of any technique must be known *a priori* and be less than the application's deadlines. A secondary goal is to provide *application independence*--allowing a broader class of problems to be handled.

We provide the necessary error masking using an exact bit-for-bit majority vote applied to the output stream. The choice of a majority vote not only provides a highly application-independent means of combining the output data but can be further justified by the difficulty in specifying the "acceptance" tests for the recovery block approach and the voting filters used in *N*-version programming [6-8].

The choice of a majority vote on the output data stream does have implications on the input structure. For deterministic tasks the input to each of the replicated tasks must be identical. Deterministic tasks always produce the same output sequence for a given input sequence.

In a tightly synchronous system it is known when the tasks will need and produce data. If the tight synchronization requirement is relaxed, which is desirable because of the high overhead for software synchronization [10], one is faced with the following questions:

- When and how does the system sample the input sensors to provide a single input datum to each of the replicated tasks?
- What action should be taken if a task is not ready when the input is sampled?
- When and how does the system resolve the replicated output data and issue a single output action honoring the individual output requests?

We answer these questions by proposing QMV and CMV.

3. ARCHITECTURE AND SOFTWARE STRUCTURE

Figure 1 presents a block diagram of the distributed real-time architecture assumed in the development of

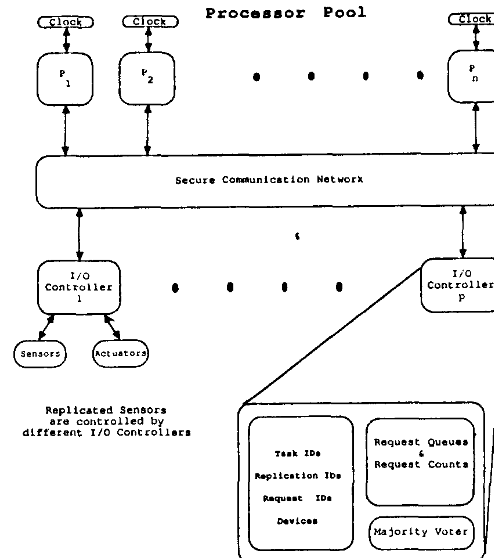

Fig. 1. Assumed Target Architecture

QMV and CMV. Each processor in the system has its own clock but the system is neither completely asynchronous nor tightly synchronized. This can be achieved by infrequent clock synchronization in software [3], which is relatively inexpensive. The clocks in any two processors may differ by at most some maximum skew. Although this assumption is not necessary for the operation of our techniques, it allows their timing behavior to be characterized as a function of the maximum skew. There must be a fault-tolerant software clock synchronization algorithm (Interactive Convergence Algorithm [3]; this algorithm entails a large time overhead [10]) allowing the entire system or subsets of the system to be resynchronized as necessary. All timing behavior of a processor depends only on its internal clock and the availability of data as provided by QMV.

The most important addition is the presence of semi-intelligent I/O controllers. These controllers must know which tasks will request service for devices under their control. This poses no major burden since the allocation of I/O devices is usually done either off-line or at the time of task creation to support deadlock prevention. This approach of preallocation of the devices is often necessary to meet the application deadlines and avoid the overhead of other deadlock resolution methods commonly found in operating system environments. The controllers must also maintain lists of requests and resolve the requests into a single action. The addition of these I/O controllers serves to insulate the I/O devices from the actions of a single processor. The ability of a single processor controlling the I/O bus and, thus, the I/O devices connected to the bus is a weakness found in the SIFT architecture. Though these I/O controllers proposed for QMV and CMV do have

complete control of the I/O devices connected to them, their limited functionality allows their complexity to be less than the requesting processors, thus achieving more reliable control of the I/O buses.

Secure communication primitives must be available for use in processor—I/O controller messages and must have the capability of detecting the origin and any modifications of the messages. Such a capability can be provided using a suitable encryption scheme. Secure communication allows the I/O controllers to detect multiple messages from faulty processors. This feature is necessary for the reliable operation of both QMV and CMV.

The last architectural requirement is that sensors obtaining replicated data be placed in different I/O controllers. This is needed to ensure that the failure of an I/O controller does not affect the ability of an application task in obtaining data from sensors. Data from replicated sensors is not resolved by the I/O controllers but is handled in the applications. This preserves application independence and allows the applications flexibility in managing redundant sensors.

We are primarily interested in real-time applications and thus will restrict the scope of the applications by assuming a structure present in real-time tasks. A block diagram of a representative real-time task is presented in figure 2. This structure hinges on the assumptions that real-time applications—

- can be naturally decomposed into tasks with three phases [9]
- have a single-source/single-sink I/O requirement.

During the first phase, input operations collect the data from a single (but replicated) input source needed for the following computation phase. This operation is referenced to the processor's clock in which the task is resident and, thus, periodic task invocations will drift as a function of their host processor's clock drift. Once the data become available, a computation phase begins with the further assumption that this phase is non-preemptive. The task is concluded with an output phase to a single device. Due to the time constraints associated with real-time tasks, this assumption is reasonable and widely used. This 3-phase structure in which the computation phase is non-preemptive and the processors operate in a non-multitasking mode is not a requirement for QMV and CMV but provides a framework in which to discuss the time bounds.

4. QUORUM-MAJORITY VOTING AND COMPARE-MAJORITY VOTING

4.1 *Naming Requests*

The operating system and underlying architecture must support several naming conventions that are used in the implementation of QMV and CMV. Tasks have to be named uniquely throughout the entire system. This poses no problems in implementation since most systems use the task names for resource allocation. We refer to these names as *task id*'s. For tasks requiring replication, each replicated copy has to be distinguishable by assigning new names called *replication id*'s. These names are easily constructed at the time the task is replicated and distributed. The last name required is used to identify the actual section (line of code) of the application task requesting I/O service. We refer to this as the *request origin* and to the name used to distinguish different request origins as the *request id*.

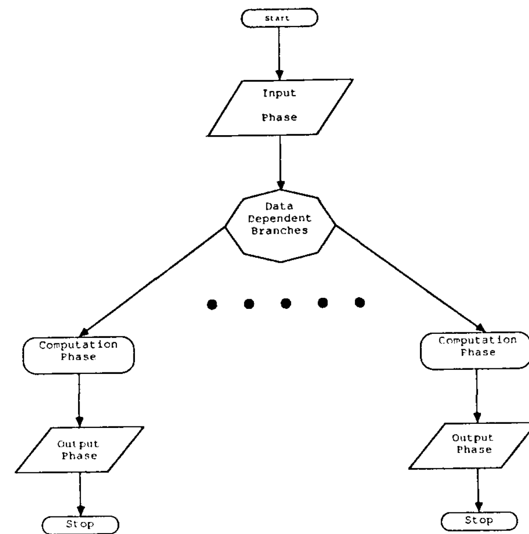

Fig. 2. Task Structure

To construct the request id's we consider a model slightly more general than the real-time task shown in figure 2. In general, requests for I/O can be functions of the input data and can be embodied within iterative loops. This implies that not only the location of the request but the progress through the loop has to be determined. This can be done by using the offset into the virtual address space of the request concatenated with the number of I/O requests received from the particular image of the task. Each properly functioning image of a task will follow the same path of execution, assuming each image receives the same input data; thus, the offset and I/O counts are identical. Tasks at different iterations through a loop have identical offsets but can be distinguished by the I/O request count. If a faulty image has its request count or offset corrupted such that it matches a non-faulty image's request, the error is masked in the vote as discussed later.

All requests in the system are then identified by the concatenation of the above three names. We refer to this as a *request tag*. The structure of the request tag is shown in figure 3.

<table>
<tr><td rowspan="2">Task_ID</td><td rowspan="2">Replication_ID</td><td>Address Offset</td><td>I/O Count</td></tr>
<tr><td colspan="2">Request_ID</td></tr>
<tr><td colspan="4">Request_Tag</td></tr>
</table>

Fig. 3. Request Tag Structure

4.2 *Description of Quorum-Majority Voting*

The basis for QMV, as the name suggests, is to ensure that I/O operations are triggered only on the behavior of a quorum of proper participants. For each request for service, as identified by the request tags, a quorum of the replicated images must issue the request before an action takes place. Once a quorum is established, the nature of the operation is decided: for input the sample is taken and sent, for output the received data values are voted upon and the resulting action taken.

Prior to the execution of a task, the required I/O controllers must be informed of the task id's and their associated replication id's. Given this information the controllers can decide if a quorum of the requests has been established for a particular I/O request. The threshold defining a quorum is set such that the non-faulty tasks will always control the majority vote.

I/O controllers must maintain a list of requests for each active request id. Upon receiving a request, the request must first be checked for valid task and replication id's. Once these are established, the controller checks the list corresponding to the request id for a duplicate request. Duplicate requests can be detected since the origin of the message is guaranteed by the assumption of secure communications. If the request was not a duplicate and a quorum of requests has been received, the controller performs a majority vote on the requests received thus far. For input, the value is sampled and sent to all participants, even those who have not yet issued that particular request. For output, the data values received are used as input to a bit-for-bit majority vote to derive a single output action. If a quorum has not been established, the request is saved and the requesting task must wait.

In order to tolerate k faulty task-images, the non-faulty images must first be able to control the establishment of a quorum. Once a quorum is formed, the non-faulty images must also be able to dominate the majority vote for the selection of the operation and in the case of an output operation the output data as well. Input and output operations can be treated symmetrically by voting on both the data and the operation selection as a single entity. To obtain the number of images required to tolerate k faulty images, start working backwards from the final majority vote. Thus, we need $k + 1$ non-faulty images against the k faulty images in the final majority vote giving rise to having the quorum established on $2k + 1$ requests. Since the quorum is established on $2k + 1$ images and the non-faulty images must be able to establish a quorum independently of the faulty images, we need $2k + 1$ non-faulty images. All $2k + 1$ non-faulty images are needed in the case that all k faulty images chose to abstain from making a request. This requires that there be a total of $3k + 1$ replicated images. In summary, $3k + 1$ replicated images are required in order to tolerate the behavior of k faulty images when QMV is used. Despite their fundamental differences, the minimum number of participants required for QMV is the same as that for the Byzantine Generals agreement [3].

If k is reasonably bounded, the I/O controllers can efficiently support the majority vote in hardware by bit serially voting on the data of the quorum.

4.3 *Description of Compare-Majority Voting*

In contrast to QMV, CMV requires only $2k + 1$ replicated images but handles the requests for I/O operations in a different manner. In performing CMV the I/O controllers wait for $k + 1$ requests for I/O from different processors with the *same* data. If the operation is an input request, the data is sampled and sent to all $2k + 1$ images. For an output operation the data is sent to the specified device. In either case, since $k + 1$ requests with the same data have to be received before any action can take place, I/O operations cannot occur solely as the result of faulty task images.

The disadvantages of CMV as compared to QMV is that the I/O controllers have to partition the incoming requests into equivalence classes as defined by the data of the requests. These operations are not trivial and could appreciably increase the hardware complexity as compared to the requirements for QMV.

5. TIMING BOUNDS

5.1 *Notation*

We introduce notation that can formally describe QMV and CMV. The notation presented below describes one particular request as identified by the request id.

Upon the the arrival of a request at the I/O controller, the request must first be validated. This involves ensuring that the task and replication id match known identities and are coded in such a manner that only the replicated process with that replication id could have issued the request. Once these are established, a check for duplicate requests concludes the validation phase.

Notation

Req$_i$ valid request i received at the controller for a given request id; $1 \leqslant i \leqslant 3k + 1$. These requests can be either an input or output operation on a particular device under the control of the I/O controller.

$\mathbf{Q}_i$ set of i requests for a given request id

$\mathbf{Q}_1 \equiv \{\mathbf{Req}_1\}$
$\mathbf{Q}_i \equiv \{\mathbf{Req}_i\} \cup \mathbf{Q}_{i-1},\ 2 \leqslant i \leqslant 3k + 1.$

These sets represent the lists used to collect the requests.

Majority(•) function that maps a set $\mathbf{Q}_i$ to a single operation according to the standard majority function.

$\mathbf{T(Req}_i\mathbf{)}$ time that the request $\mathbf{Req}_i$ was received by the I/O controller, for QMV and CMV. This function maps requests to the time of their creation—to analyze the timing behavior.

$\mathbf{I}_i, \mathbf{O}_i, \mathbf{R}_i$ either the request i for input, output, or a renegade request, respectively. This notation is similar to $\mathbf{Req}_i$ but allows differentiating faulty from non-faulty. We *notationally* discriminate faulty and non-faulty requests. Since the behavior of the faulty processors is unpredictable, it is useful to obtain the timing bounds in terms of the non-faulty processors behavior. This information is not used in the implementation of QMV.

m number of faulty tasks for this request

k maximum number of faulty tasks that can be tolerated.

5.2 *Timing Bounds for QMV*

Consider the timing behavior of QMV and use figure 2 as an example real-time task. The task is replicated $3k + 1$ times and distributed on processors $\{p_1, \ldots, p_{3k+1}\}$ in order to tolerate up to a total of k failures. QMV is implemented as follows:

1. Wait for the set $\mathbf{Q}_{2k+1}$ to become available.
2. Perform operation dictated by $\mathbf{Majority(Q}_{2k+1}\mathbf{)}$.

These operations take place at $\mathbf{T(Req}_{2k+1}\mathbf{)}$ and the variation in this quantity is a function of non-faulty processors clock drift and faulty task's behavior. The variation (jitter) in $\mathbf{T(Req}_{2k+1}\mathbf{)}$ forms an *operation window*. If the application deadlines are to be met, the growth of this operation window must be limited. If the growth of the window were not limited, the faulty tasks could either prematurely trigger an operation or indefinitely delay an operation, resulting in a failure of the system. We now present the bounds on the operation window.

Theorem 1. The operation window formed by $\mathbf{T(Req}_{2k+1}\mathbf{)}$ is bounded as follows:

$$\left.\begin{array}{ll} \mathbf{T(I}_{2k+1-m}\mathbf{)} \leqslant \mathbf{T(Req}_{2k+1}\mathbf{)} \leqslant \mathbf{T(I}_{2k+1}\mathbf{)} & \text{(Input Operation)}, \\ \mathbf{T(O}_{2k+1-m}\mathbf{)} \leqslant \mathbf{T(Req}_{2k+1}\mathbf{)} \leqslant \mathbf{T(O}_{2k+1} & \text{(Output Operation)}, \end{array}\right. : 0 \leqslant m \leqslant k$$

Proof. The $3k + 1 - m$ non-faulty requests for service arrive in an ordered sequence at the I/O controller for the device in question. These are shown in figure 4 for the case of an input operation. In the presence of the m faulty images, their requests (images not making requests are considered to make their requests at $+\infty$) are dispersed among the non-faulty requests. This allows us to divide the time that the operation will be triggered into three cases.

Case 1 $[\mathbf{T(R}_i\mathbf{)} \leqslant \mathbf{T(I}_{2k+1-m}\mathbf{)},\ 1 \leqslant i \leqslant m,\ 0 \leqslant m \leqslant k]$: The m faulty tasks all make their requests before $\mathbf{T(I}_{2k+1-m}\mathbf{)}$. The $2k + 1 - m$ non-faulty tasks dominate the m faulty tasks in the decision for an input operation with this occuring at $\mathbf{T(I}_{2k+1-m}\mathbf{)}$.

Case 2 $[\mathbf{T(R}_i\mathbf{)} \geqslant \mathbf{T(I}_{2k+1}\mathbf{)},\ 1 \leqslant i \leqslant m,\ 0 \leqslant m \leqslant k]$: All of the faulty tasks chose to either delay their requests until after $\mathbf{T(I}_{2k+1}\mathbf{)}$ or abstained from making requests. The quorum is then established at $\mathbf{T(I}_{2k+1}\mathbf{)}$ and have only non-faulty requests present. Therefore, the operation is an input and occurs at $\mathbf{T(I}_{2k+1}\mathbf{)}$.

Case 3 [Otherwise]: The quorum is established some time between $\mathbf{T(I}_{2k+1-m}\mathbf{)}$ and $\mathbf{T(I}_{2k+1}\mathbf{)}$. Thus, the time that the quorum is established is bounded between two non-faulty requests. Since there are at least $k + 1$ non-faulty tasks in the quorum, the non-faulty tasks can again dominate the majority function.

The above argument gives the time bound for an input operation and a symmetric argument is applicable to output operations. □

Since the operation takes place between non-faulty requests, the behavior of faulty task images cannot force a premature triggering or indefinite delay of the operation.

Processors that have requested the input data prior to $\mathbf{T(Req}_{2k+1}\mathbf{)}$ can start their computation phase upon receipt of the data but the other lagging processors cannot start until the attempt is made to request the data. This leads to the output operation window floating with respect to the input window and, thus, we need to bound on the growth of the output window as a function of the input window.

Before we can obtain a bound on the output window in terms of the input window, we have to define *computation time* and account for drift in the processor clocks.

More Notation

W amount of computational time between the input phase and the following output phase for an "ideal" processor without any drift in its clock.

δ_{max} maximum clock drift rate for all the non-faulty processors. $\delta_{max} > 0$. It is dimensionless and is used as a proportionality constant giving either the maximum time gained or lost when multiplied by a time interval.

A bound on the output window growth is presented in the following theorem.

Theorem 2. The maximum growth of an output operation window with respect to the preceding input window is $2\delta_{max}W$.

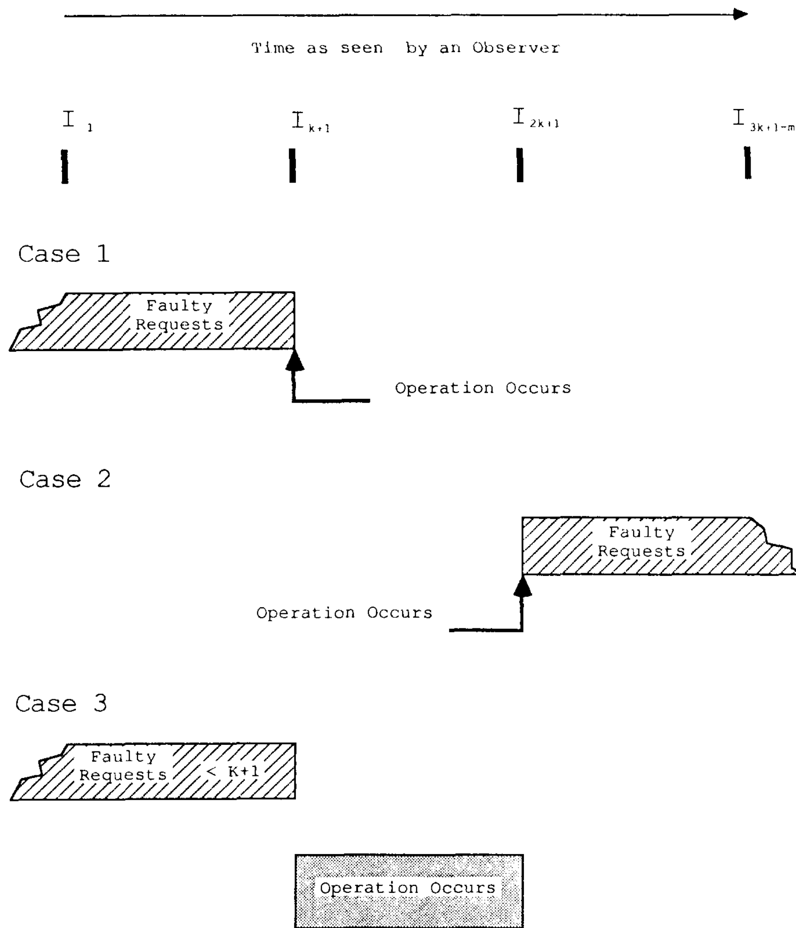

Fig. 4. Input Operation Case

Proof. To arrive at the bound, we first obtain ${}^{min}\mathbf{T}(\mathbf{O}_{2k+1-m})$ and ${}^{max}\mathbf{T}(\mathbf{O}_{2k+1})$ which represent the earliest and latest possible times that the output operation can occur.

To obtain ${}^{min}\mathbf{T}(\mathbf{O}_{2k+1-m})$, we present the following argument. Given that the input occurs at time $\mathbf{T}(\mathbf{I}_{2k+1-m})$, there are $2k + 1 - m$ non-faulty processors starting their computation phase. Thus, the earliest that those non-faulty processors can finish the computation phase is in $W - \delta_{max}W$ time units later. For this to occur, the m faulty processors that requested early for the input phase have to do so again along with all $2k + 1 - m$ non-faulty processor experiencing the maximum speedup drift, a highly unlikely event.

$$^{min}\mathbf{T}(\mathbf{O}_{2k+1-m}) = \mathbf{T}(\mathbf{I}_{2k+1-m}) + W - W \cdot \delta_{max}, 0 \leqslant m \leqslant k.$$

For ${}^{max}\mathbf{T}(\mathbf{O}_{2k+1})$ the argument is symmetric to ${}^{min}\mathbf{T}(\mathbf{O}_{2k+1-m})$. The input has to occur at $\mathbf{T}(\mathbf{I}_{2k+1})$ requiring all faulty processors to have requested their output after $\mathbf{T}(\mathbf{I}_{2k+1})$. At this time there will be $2k + 1$ non-faulty processors in their computation phase. The latest these $2k + 1$ processors can finish their computation is in $W + \delta_{max}W$ time units later. Again, requiring the faulty processors to delay their requests after the first $2k + 1$ non-faulty requests, we get:

$$^{max}\mathbf{T}(\mathbf{O}_{2k+1}) = \mathbf{T}(\mathbf{I}_{2k+1}) + W + W \cdot \delta_{max}, 0 \leqslant m \leqslant k.$$

Then it is easy to get:

$$\textbf{Maximum Output Window} \leqslant {}^{max}\mathbf{T}(\mathbf{O}_{2k+1}) - {}^{min}\mathbf{T}(\mathbf{O}_{2k+1-m}) \leqslant \mathbf{T}(\mathbf{I}_{2k+1}) - \mathbf{T}(\mathbf{I}_{2k+1-m}) + 2W \cdot \delta_{max}: 0 \leqslant m \leqslant k$$

The window growth is then the difference between the maximum output window and the input window giving rise to a growth of $2W\delta_{max}$. □

5.3 *Timing Bounds for CMV*

More Notation

P_i partition of $\mathbf{Q}_i$ such that $\mathbf{Req}_j \equiv \mathbf{Req}_k$ iff value $(\mathbf{Req}_j)$ = value(Req_k), where value (•) extracts the data content of a request, and the symbol ≡ represents an equivalence relation.

$F(\bullet)$ $F(\mathbf{Q}_i) \equiv \max_{S \in P_i}\{|S|\}$.

The operation of CMV then occurs at $\mathbf{T}(\mathbf{Q}_x)$, where $x \equiv \min\{j : F(\mathbf{Q}_j) \geqslant k + 1\}$.

The operation window defined when using CMV can then be derived.

Theorem 3. The operation window formed by $\mathbf{T}(\mathbf{Q}_x)$, $x \equiv \min\{j : F(\mathbf{Q}_j) \geqslant k + 1\}$, is bounded as follows:

$$\mathbf{T}(\mathbf{I}_1) \leqslant \mathbf{T}(\mathbf{Q}_x) \leqslant \mathbf{T}(\mathbf{I}_{k+1}) \quad \text{(Input Operation)}$$
$$\mathbf{T}(\mathbf{O}_1) \leqslant \mathbf{T}(\mathbf{Q}_x) \leqslant \mathbf{T}(\mathbf{O}_{k+1}) \quad \text{(Output Operation)},$$
$$0 \leqslant m \leqslant k$$

Proof. Consider the case of an input operation. Assume there exists an equivalent class with cardinality $k + 1$ or greater prior to receiving the first non-faulty input request $(\mathbf{I}_1)$. This cannot be so since there can be at most k faulty requests prior to the first non-faulty input request. This shows that $\mathbf{T}(\mathbf{Q}_x) \geqslant \mathbf{T}(\mathbf{I}_1)$. Since there will be at least $k + 1$ identical input requests by the $k + 1^{th}$ non-faulty input request, the inequality $\mathbf{T}(\mathbf{Q}_x) \leqslant \mathbf{T}(\mathbf{I}_{k+1})$ trivially holds.

The output case follows from a symmetric argument. □

5.4 *Bounds Interpretation*

Two interesting properties of the operational windows have been presented.

1. For QMV(CMV), the largest that these operational windows become is the inner ⅓ (first ½) of the non-faulty processors with this condition occurring only when there is a maximum number of faulty tasks in the system. The maximum size of the windows for QMV improves (shrinks) as the number of faulty tasks decreases with no window occurring in a fault-free system.
2. These windows grow as a function of the maximum drift rate of the processors and the time that has elapsed since the processors were last resynchronized.

The exploitation of these properties provides an alternative to current real-time system implementations. The parameter that system designers may vary is the resynchronization interval. This allows the reduction of the resynchronization overhead as a function of the applications tolerance to the maximum jitter of the operational windows provided by QMV and CMV.

ACKNOWLEDGMENT

This work has been supported in part by the US Office of Naval Research under contracts N00014-85-K-0531 and N00014-85-K-0122, and NASA under grant NAG-1-296. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not reflect the views of funding agencies.

REFERENCES

[1] T. B. Smith, J. H. Lala, *Development and Evaluation of a Fault-Tolerant Multi-Processor (FTMP) Computer Volume I: FTMP Principles of Operation*, NASA Contractor Report 166071, 1985 May.

[2] T. B. Smith, *Fault Tolerant Processor Concepts and Operation*, Contractor Report CSPL-P-1727, Charles Stark Draper Laboratory, 1983 May.

[3] Goldberg, Green, Kautz, Levitt, Melliar-Smith, Schwartz, Weinstock, *Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer*, NASA Contractor Report 172146, 1984 February.

[4] W. R. Dunn, "Distributed asynchronous microprocessor architectures in fault tolerant integrated flight systems", *AIAA Computers in Aerospace IV Conf.*, 1983 October, pp 115-123.

[5] K. N. Levitt, P. M. Melliar-Smith, R. L. Schwartz, *Fault Tolerant Architectures for Integrated Aircraft Electronics Systems*, NASA Contractor Report 172226, 1983 August.

[6] T. Anderson, P. A. Lee, *Fault Tolerance: Principles and Practice*, Prentice-Hall, 1981.

[7] L. Chen, A. Avizienis, "N-version programming: a fault-tolerance approach to reliability of software operation", *Digest of Papers, 8th Ann. Intern. Symp. Fault-Tolerant Computing Systems*, 1978, pp 3-9.

[8] R. K. Scott, J. W. Gault, D. F. McAllister, "Fault-tolerant software reliability modeling", *IEEE Trans. Software Engineering*, vol SE-13, 1987 May, pp 582-592.

[9] K. G. Shin, C. M. Krishna, Y. H. Lee, "A unified method for evaluating real-time computer controllers and its application", *IEEE Trans. Automatic Control*, vol AC-30, 1985 Apr, pp 157-366.

[10] C. M. Krishna, K. G. Shin, R. W. Butler, "Synchronization and fault-masking in redundant real-time systems", *Digest of Papers, 14th* Ann. Intern. Symp. Fault-Tolerant Computing Systems, 1984, pp 152-157.

AUTHORS

Dr. Kang G. Shin, Professor; Real-Time Computing Laboratory; Department of Electrical Engineering and Computer Science; The University of Michigan; Ann Arbor, Michigan 48109-2122 USA.

Kang G. Shin [S'75, M'78, SM'83] is a Professor in the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, which he joined in 1982. He has been active and authored/coauthored over 120 technical papers in the areas of fault-tolerant real-time computing, computer architecture, and robotics and automation. In 1987, he received the Outstanding Paper Award from the *IEEE Transactions on Automatic Control* for a paper on robot trajectory planning. In 1985 he founded the Real-Time Computing Laboratory, where he and his students are building a 19-node hexagonal mesh multiprocessor, called **HARTS**, to validate various architectures and analytic results in the area of distributed real-time computing. He received the BS degree in Electronics Engineering from Seoul National University, Seoul, Korea in 1970, and both the MS and PhD degrees in Electrical Engineering from Cornell University, Ithaca in 1976 and 1978, respectively.

(continued to page 67)