An Efficient Real Time Fault Detection and Tolerance Framework Validated on the Intel SCC Processor

Devendra Rai, Pengcheng Huang, Nikolay Stoimenov and Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

ABSTRACT

We present a new framework that efficiently detects and tolerates *timing faults* in real time systems. Timing faults are observed when the inputs and/or outputs of a given system fail to meet their desired timing properties, such as I/O rates. Most current approaches either rely on heartbeat monitoring which is too restrictive; or on statistical or inexact methods which are not suitable for embedded real time systems. Current approaches based on the abstract real time model of the given application are resource intensive, and may not be suitable for embedded systems. Our framework utilizes active replication, and is based on already existing timing models for real time applications to develop fault detection and tolerance strategies. The approach does not require any timekeeping at runtime, and is efficient in terms of computational resources used. Experiments using three realistic applications on the Intel Baremetal SCC demonstrate the efficiency of our framework, both in memory and computational resources used.

1. INTRODUCTION AND RELATED WORK

Modern safety critical systems are often designed to be *fail-silent*, i.e., a non-faulty application provides the correct output, both in the value and time domain, or in case of a fault, stops providing any output altogether. Thus, such systems are designed to exhibit any fault *only* as a *timing fault.* Specifically, a system (or a part of it) exhibits a *timing fault* when one or more of its inputs or outputs fail to meet the desired timing properties, such as rates, or deadlines. For safety critical embedded systems, it is important that timing faults can be detected and tolerated, as efficiently as possible, in terms of memory and computational resources used. Various techniques already exist, both at the application level and at the hardware level, which ensure that all faults are exhibited solely as timing faults. Brasileiro *et. al.* describe the construction of a fail-silent system at the application level, whereas a patent provides an example of how processors are now designed to enforce fail-silent behavior, see [6, 10] for references.

Efficient detection of timing faults remains a challenge: In cases where the application exhibits simple timing behavior, (e.g., strictly periodic), timeout (e.g., watchdog) based solutions may be used. Such simple approaches are not effective for use in applications based on general dataflow process networks, which are in general asynchronous and can have bursty timing characteristics. Such process networks are generally used to design and implement streaming applications e.g., radar processing. Timing fault detection is particularly difficult in such process networks, since the fault detection logic has to contend with possible anomalies in the value of the output, as well as its timing.

Such challenges have resulted in the development of *inexact arbitration* approaches, which are statistical or probabilistic in nature, see [4, 5]. Genetic algorithms and neural networks have also been proposed for use in the arbitration logic, which may not be suitable for use in real time systems, see [12]. Though neural networks are capable of learning new and possibly complex fault detection rules, the major problem with this approach remains the design of appropriate training data which can cover all possible corner cases. Furthermore, a complex neural network may have considerable memory and runtime overhead, and is therefore not suitable to embedded real time systems.

In another approach, a set of processes in the network share their internal states with each other, and fault detection strategy is based on the difference in their internal states, see [9]. Therefore, the application to be monitored for faults must be specifically designed in a way that all processes make their internal states observable to the detector. Furthermore, it is not clear how the approach scales with the number of processes in the application.

A fault-detection approach based on *distance functions* works by monitoring stream properties (e.g., token arrival times), see [11]. The memory and runtime efficiency of this technique relies on an approximation of general distance functions with *l-repetitive* distance functions. Thus, the technique gains runtime resource efficiency at the cost of over approximating the real time properties of the given application leading to false positives and/or false negatives.

This paper focuses on real time process networks, and takes a new approach to detecting timing faults by utilizing analytic real time models for buffer sizing. Such models are usually co-developed at the design stage of the real time and safety critical applications. Faults are tolerated using the active replication technique, which is common in real time safety-critical systems. The application is treated as a black-box, whose interface-level timing models are either available, or can be generated quickly from calibrations, making our approach applicable to large and complex applications.

Specifically, we solve the following problem in this paper:

Provide a provably correct and efficient mechanism for detecting and tolerating single timing faults in real time process networks.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593085.

Without loss of generality, we focus on tolerating at most one permanent timing fault, using two replicas of a given real time data flow process network. This restriction can be easily relaxed by adding more replicas to the system, and a more general setup for tolerating upto n timing faults can be easily constructed using the principles outlined in this paper. The main contributions of this paper are summarized as follows:

1. Design of provably correct arbitration mechanisms for a duplicated real-time process network such that single timing faults can be efficiently tolerated.
2. Memory and time efficient fault detection algorithms which do not require any runtime time-keeping.
3. Validation on the Intel SCC processor using three representative streaming applications.

1.1 Motivational Example

It has already been discussed that detecting timing faults by present methods is hard. We now show that tolerating timing faults is not trivial. Consider a simple process network shown on the top side of Figure 1 that contains processes and communication channels with FIFO semantics. Part of the process network, containing all processes and channels that implement the main functionality, called the *critical subnetwork*, is duplicated for fault-tolerance, i.e., we have two replicas. A set of producer processes provides data tokens to the critical subnetwork, and a set of consumer processes consumes tokens from this subnetwork. For the simplicity of presentation, we assume a single producer and a single consumer processes, denoted as P and C, respectively, however, the critical subnetwork can be arbitrarily complex.

A *replicator channel* duplicates an output stream from a producer to each replica, whereas a *selector channel* combines the streams from the replicas into a single input stream for a consumer. In the paper, we refer to the process network with un-replicated critical subnetwork as *reference*, and to the process network with two replicas of its critical subnetwork as *duplicated*.

It is required that at their respective input/output processes, both in the reference and duplicated process networks, behave equivalently, even when one of the replicas suffers a single permanent timing fault. We assume that the sequence of data tokens produced by a process and the process network is independent of the timing of the network (e.g., Kahn Process Network). All FIFO queues have bounded capacities. Processes have blocking semantics. Therefore, a process attempting to write tokens to a full output FIFO queue, or attempting to read tokens from an empty input FIFO queue will block, until the said operation can be successfully completed. For simplicity, assume that only the critical subnetwork may suffer from a permanent timing fault.

Merging Streams at a Selector Channel. With only two replicas, arbitration by majority voting is ruled out. Simple techniques (e.g., timer, heartbeat) are not applicable here since the given process network may have asynchronous with bursty timing characteristics. An option is to have a dedicated fault detection mechanism which can detect faults by observing internal states of the real time application, but it requires that the application be redesigned to make its internal states observable. Additionally, the resulting detector may be too complex to be used in an embedded real time system. It is therefore desirable that at the selector we have a fault-tolerance mechanism that can take into account the possible complex and bursty behaviour of output streams but it is yet simple and efficient.

Deadlocked Non-Faulty Replicas. Assume that the selector is able to detect a timing fault in the top replica, and as a result, the selector stops destructively reading tokens from this subnetwork. Eventually, the top sub-network stops consuming tokens from its input, causing the FIFO queue at the replicator to fill up and the respective producer to block. This in turn starves the lower (correctly working) subnetwork from processing further tokens, causing the selector to erroneously flag it as faulty, compromising the reliability of the entire system. A dedicated fault monitor may be too complex to be used in an embedded real time system. In another approach, a fault tolerant network can be built to allow replicators and selectors to reliably exchange messages, which may require significant resources. Alternatively, the replicator may allow non-blocking writes by the producer process P, requiring that the replicator channel be able to store an unbounded number of tokens. It is therefore desirable to have an efficient fault-tolerance mechanism at the replicator without the need for a reliable communication with the selector.

2. NOTATIONS AND MODEL

For simplicity, we consider a simple dataflow process network with one critical subnetwork connected to a single producer and a single consumer via a FIFO queue on either side, see Figure 1. All presented results are equally applicable to a general model with the critical subnetwork having multiple input and output channels. The input and output ports of the critical subnetwork are denoted by $\mathcal{I}$ and $\mathcal{O}$, respectively. Communication between processes is done via read and write operations on FIFO channels with finite capacities, and the processes have blocking semantics. The capacity of a FIFO queue F_i is denoted by $|F_i|$. We require that a timing fault does not lead to wrong data (value of a token) in the application, and hence we assume that the process network is *determinate*, i.e., the sequence of tokens and their values produced by a process network is dependent only upon the sequence of input tokens, and not upon the timing of token availability. The producer, the consumer, and the reference and duplicated process networks have real time characteristics. Their timing properties can be, for example, specified in terms of *arrival curves* or any other real time model. Details on arrival curves are presented in Section 3, and can also be found in [1].

The fault tolerant system is constructed by duplicating the critical subnetwork, into replicas R_1 and R_2, along with necessary FIFOs queues and channels. The replicas have sufficient design diversity in order to prevent common-mode faults. A special *replicator* channel duplicates the stream from the producer process P to the corresponding input ports of the replicas, $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Similarly, a *selector* channel arbitrates (merges) the data streams from the output ports of the replicas, $\mathcal{O}_1$ and $\mathcal{O}_2$ and provides the resulting stream to the consumer process C. A token produced by a replica R_k on its output channel is denoted as $T_k[j]$, where $j \in \mathbb{N}^+$ is the monotonically increasing *sequence number* of the said token. A function $t : \mathbb{N}^+ \times \mathbb{N}^+ \rightarrow \mathbb{R}_{\geq 0}$ provides the timestamp of a token $T_k[j]$, given as $t(k, j)$, indicating the time instant when the token was produced. We assume that owing to hardware costs, only a part of the system can be made reliable, see [8]. Thus, only the processes and channels within the replicas R_1 and R_2 are unreliable, whereas the rest of the system is executing on reliable hardware. Of course, the replicas may be arbitrarily large and complex applications. We assume that the system can experience at most a single timing fault, which is eventually observed when the faulty replica either stops producing (or consuming) tokens, or does so at a rate lower than expected.

3. PROPOSED SOLUTION

First we discuss the design of the replicator and the selector channels. It yields a duplicated process network equivalent to the reference process network, both in functionality and

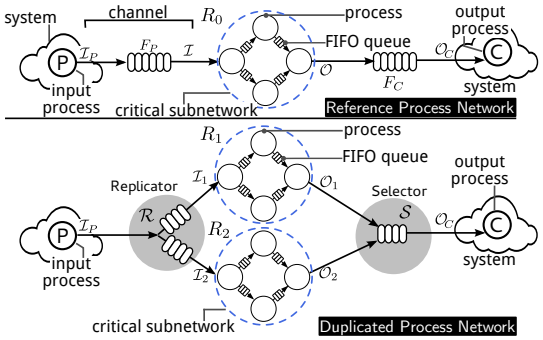

Figure 1: The reference and duplicated process networks. For simplicity, the critical subnetwork has only one input and output channel(s).

timing, see Section 3.2. Section 3.3 discusses fault-detection mechanisms in the replicator and the selector while considering the bounded system memory. Finally, Section 3.4 presents the necessary mathematical details for queue sizing.

3.1 Replicator and Selector

We assume that all read and write operations to the replicator and the selector channels to be atomic. A replicator channel $\mathcal{R}$ has two reading interfaces and a single writing interface, and is described by the following rules:

1. It contains two FIFO queues of sizes $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ respectively, one for each reading interface. Each queue has a space and fill variables which are initially set to: $fill_1 = fill_2 = 0$ and $space_1 = |\mathcal{R}_1|, space_2 = |\mathcal{R}_2|$.
2. Each reading interface of the replicator has a destructive and blocking read access to the corresponding queue. A *read* event increments the corresponding space variable and decrements the corresponding fill variable.
3. If $\min\{space_1, space_2\} > 0$, then a *write* event to the write interface queues a token to both FIFO queues, decrements $space_1$ and $space_2$, and increments $fill_1$ and $fill_2$, else the write to the replicator is blocked.

In other words, the replicator channel duplicates every input token to both FIFO queues, each one linked to a read interface. More efficient implementations utilizing circular FIFO buffers with two readers are possible, but we retain the simple design for the present discussion.

A selector channel $\mathcal{S}$ has two writing interfaces and a single reading interface, and is described by the following rules:

1. There are two space variables $space_1, space_2$ and a single variable *fill* associated with the queue. Initially, we have $fill = 0$ and $space_1 = |\mathcal{S}_1|, space_2 = |\mathcal{S}_2|$. The selector maintains only a single FIFO queue for the channel, with size given by $|\mathcal{S}| = \max\{|\mathcal{S}_1|, |\mathcal{S}_2|\}$.
2. The reading interface of the selector has a destructive and blocking read access to the queue. A *read* event increments all space variables and decrements the fill variable.
3. A *write* event to a write interface such as interface 1 blocks if $space_1 = 0$. Suppose now that $space_1 > 0$. If $space_1 \leq space_2$, then the token to be written by interface 1 is enqueued in the FIFO, *fill* is incremented and $space_1$ is decremented. Otherwise, just $space_1$ is decremented and the corresponding token is dropped.

In other words, the selector contains two virtual queues, one for each writing interface. Under fault-free conditions, both replicas provide the same sequence of tokens to the selector, and selector must queue the token from a write interface which provides the first token of each duplicate pair. Therefore, the selector queues a token from interface 1 if $space_1 \leq space_2$, else it queues from interface 2. A process can successfully read from the selector FIFO if $fill > 0$.

3.2 Equivalence

We show that if the FIFO queues in the replicator are unbounded, the duplicated process network is equivalent, to the reference process network, both in functionality and timing, even if one replica suffers a single timing fault. First, we present necessary definitions and a lemma.

A sequence of tokens produced by replica R_k is denoted as $\mathcal{Q}_k = \langle T_k[1], T_k[2], T_k[3], \cdots \rangle$. When a sequence $\mathcal{Q}'_k$ is a *prefix* of $\mathcal{Q}_k$, it is represented as $\mathcal{Q}'_k \sqsubseteq \mathcal{Q}_k$. For example $\langle T_k[1], T_k[2] \rangle \sqsubseteq \langle T_k[1], T_k[2], T_k[3] \rangle$. The sequence of timestamps associated with $\mathcal{Q}_k$ is given by $t(\mathcal{Q}_k) = \langle t(k,1), t(k,2), t(k,3), \cdots \rangle$.

We assume that the consumer process has real time characteristics, and expects at its input only sequences of tokens that can meet these characteristics. All sequences of timestamps associated with $\mathcal{Q}_k$ satisfying the timing requirements of the consumer are represented by the set $\mathcal{T}_C$ of sequence of timestamps. If a sequence $\mathcal{Q}_k$ with timestamps $t_1(\mathcal{Q}_k)$ satisfies the requirements of the consumer, i.e., $t_1(\mathcal{Q}_k) \in \mathcal{T}_C$, then the same sequence with different timestamps $t_2(\mathcal{Q}_k)$ also satisfies the requirements of the consumer if some of the tokens arrive *earlier*, i.e., we have $t_2(\mathcal{Q}_k) \in \mathcal{T}_C$, if:

$$t_2(k,j) \leq t_1(k,j) \quad \forall j \tag{1}$$

We also assume that each replica individually must be able to satisfy the timing characteristics of the consumer. Therefore, in a duplicated process network, the timestamps $t(\mathcal{Q}_1)$ and $t(\mathcal{Q}_2)$ produced by replicas R_1 and R_2, respectively, must satisfy $t(\mathcal{Q}_1) \in \mathcal{T}_C$ and $t(\mathcal{Q}_2) \in \mathcal{T}_C$.

A pre-requisite to equivalence between the duplicated and the reference process networks is that the selector *isolates* the replicas from each other:

LEMMA 1. *The selector prevents the output of one replica from affecting the output of the other, both in value and time.*

Proof: From the properties of the process network, and those of the selector, a replica, say R_2 can *only delay* the tokens from the replica R_1. This delay would be due to any backpressure caused by R_2, which is experienced by R_1. However, from rule 3 of the selector, the only variable that governs the back-pressure felt by R_1 is $space_1$. From the construction of the selector, the $space_1$ variable is never modified by write interface 2 (and vice versa), and hence, the back-pressure felt by R_1 is never caused (or contributed to) by R_2. The lemma follows. ■

The functional and timing equivalence between the duplicated and reference process network is shown next:

THEOREM 2. *If the replicator has unbounded FIFO queues, then a sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$ provided to the reference and duplicated process networks results in the same output sequence $\mathcal{Q}_C$ from both the reference and duplicated networks, even if the duplicated process network suffers a single timing fault. Furthermore, if the timestamps of the sequence generated by the reference process network $t(\mathcal{Q}_C) \in \mathcal{T}_C$, then the sequence of timestamps $t'(\mathcal{Q}_C)$ generated by the duplicated process network is also in $\mathcal{T}_C$.*

Proof: Since the replicator FIFO queues are unbounded, $\min\{space_1, space_2\} > 0$ (rule 3 of the selector) is always true, and consequently, a replicator channel always duplicates each token to both input ports $\mathcal{I}_1$ and $\mathcal{I}_2$ of the replicas. Furthermore, the replicator does not change the timestamp of a token when it inserts it into both FIFO queues. Thus, a sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$ at the write interface

of the replica always results in the same sequence $\mathcal{Q}_P$, with the same timestamps, at $\mathcal{I}_1$ and $\mathcal{I}_2$.

Under no fault conditions, the replicas are determinate but non-deterministic in timing characteristics, therefore, given the same input sequence $\mathcal{Q}_P$ with timestamps $t(\mathcal{Q}_P)$, the replicas produce at their output ports output sequences $\mathcal{Q}_1 = \mathcal{Q}_2$, with non-equal sequences of timestamps $t(\mathcal{Q}_1) \neq t(\mathcal{Q}_2)$ respectively.

Next, the selector evaluates which replica has provided the most recent token of a duplicate pair, by evaluating $space_1 \leq space_2$. If $space_1 \leq space_2$, then the replica R_1 has provided the first token of the most recent duplicate pair, which is queued into the FIFO, and the selector simply discards the corresponding late arriving token from R_2. In other words, the selector queues the earlier arriving token from each duplicate pair into its FIFO, resulting in a sequence $\mathcal{Q}_C = \mathcal{Q}_1 = \mathcal{Q}_2$ with timestamps $t(\mathcal{Q}_C)$. Since $t(\mathcal{Q}_1) \in \mathcal{T}_C$ and $t(\mathcal{Q}_2) \in \mathcal{T}_C$, then as in (1), we have $t(\mathcal{Q}_C) \in \mathcal{T}_C$ (also see Lemma 1).

If replica R_1 experiences a timing fault at any instant t, then eventually, we have $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$ and $space_2 \leq space_1$, and the selector simply queues the tokens from replica R_2. The timestamp of a token missing in $\mathcal{Q}_1$ but with a corresponding token in $\mathcal{Q}_2$ is taken to be infinity, and therefore timestamps of the tokens produced by the selector subsequent to a fault correspond to those from R_2.

For comparison, given $\mathcal{Q}_P$, the reference process network produces a sequence $\mathcal{Q}_C = \mathcal{Q}_2$, which is the same output as the non-faulty replica R_2 produces (since the replicas are determinate and are derived from the reference process network). Furthermore, *if* the reference process network meets the timing requirements of the consumer, then $t(\mathcal{Q}_C) \in \mathcal{T}_C$. ■

3.3 Fault Tolerance with Bounded Memory

We assume that the reference process network has been designed correctly, i.e., all FIFO queues have been sized appropriately such that a producer never blocks on a full FIFO queue, and a consumer never stalls on an empty FIFO queue.

In order to ensure that the reference process network and the duplicated process network are equivalent even when the latter experiences a single timing fault, it is required that in the duplicated process network, the producer never blocks on a full replicator FIFO queue possibly associated with a faulty replica (the selector channel already has bounded memory). Therefore, functional and timing equivalence between duplicated and reference process network requires that the selector and the replicator channels be able to autonomously detect timing faults and prevent producer and consumer from blocking and stalling, respectively.

Fault Detection at the Replicator Channel. First note that the replicator FIFO queues with capacities $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ should never overflow under fault free conditions. Therefore, a replica, say R_1, is deemed faulty if the actual number of tokens in the associated FIFO attempts to exceed $|\mathcal{R}_1|$, causing the producer to block on the full FIFO. In other words, if $space_1 = 0$ when the producer attempts to write a token, then the replica R_1 is faulty. We introduce variables $fault_1$ and $fault_2$ for the replicator channels, each initialized to `FALSE`. If $space_1 = 0$ when the producer writes a new token to the replicator, then $fault_1$ = `TRUE`, and the replicator does not insert new tokens into this FIFO. Similar arguments also apply to the case with a fault in replica R_2.

This also makes it possible to detect a timing fault wherein the rate at which a replica consumes tokens from the producer is lower than predicted at design time.

Fault Detection at the Selector Channel. There are two methods for detecting a fault at the selector. The first method is simple: the replica R_1 may stall the consumer (and is hence faulty) if $space_1 > |\mathcal{S}_1|$, and similarly for R_2. The second approach is based on the intuition that if both replicas serve and satisfy timing bounds imposed by a common consumer, then the outputs from both replicas must not diverge too much from each other. The divergence is quantified by the difference in total number of tokens received by the selector over both input channels. Therefore, the selector monitors the difference $|space_1 - space_2|$ and if the difference exceeds a threshold D, then the replica R_1 is faulty if $space_1 > space_2$, else R_2 is faulty. The details will be elaborated in the next section. The rule 3 of the selector can be easily modified to include fault detection at the selector channel. Notice that this approach naturally also enables detection of timing faults wherein the actual rate at which a replica supplies tokens to a consumer falls below the one calculated at design time.

3.4 FIFO Conditions and Threshold Calculations

We present brief mathematical formulations for deriving FIFO capacities and thresholds in this section.

FIFO Capacities and Initial Fill Conditions. Let $\mathcal{G}_P[s,t)$ denote the total number of tokens generated by a producer in the interval $[s,t)$. Then, the upper and lower *arrival curves*, $[\alpha_P^u, \alpha_P^l]$ denote the maximum and minimum number of tokens generated by the producer in *any* time interval $\triangle$, see [1]:

$$\alpha_P^l(t-s) \leq \mathcal{G}_P[s,t) \leq \alpha_P^u(t-s) \;\; \forall s < t \tag{2}$$

Equation (2) is either provided as a part of the timing model, or is derived from calibration experiments. Let $[\alpha_{i,in}^u, \alpha_{i,in}^l]$ be the maximum and minimum number of tokens consumed by a replica $R_i \mid i \in \{1,2\}$ in any time interval $\triangle$. We require that the producer never blocks on its output FIFO, i.e., F_P in reference network, and equivalently, FIFO queues $\mathcal{R}_1$ and $\mathcal{R}_2$ in the replicator channel. The required capacity of the FIFO $|F_P|$ (equivalently, the capacities $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$) is given by the relation:

$$\alpha_P^u(\triangle) \quad \leq \quad \alpha_{i,in}^l(\triangle) + |F_P| \;\; \forall \triangle \geq 0 \tag{3}$$

Notice that it is *acceptable* that the replica(s) may stall on empty FIFO queues $\mathcal{R}_1$ and $\mathcal{R}_2$ as long as the consumer does not stall on *its* empty input FIFO queue. That the consumer does not stall on its empty FIFO queue, i.e., F_C in the reference network, and $\mathcal{S}_1$, $\mathcal{S}_2$ in the duplicated process network, requires an initial number of tokens, $F_{C,0}$:

$$\alpha_{i,out}^l(\triangle) \quad \geq \quad \alpha_C^u(\triangle) - F_{C,0} \;\; \forall \triangle \geq 0 \tag{4}$$

where $\alpha_{i,out}^l(\triangle)$ is the minimum number of tokens produced by the replica R_i, and $\alpha_C^u(\triangle)$ is the maximum number of tokens consumed by the consumer, in any time interval $\triangle$ respectively.

Threshold Calculations. We present the calculations only for the selector channel, and computations for the replicator channel are analogous. The difference in the total number of tokens received from both replicas, D over *any* time interval $\triangle$ is bounded by finding the smallest integer D satisfying the following inequality:

$$D \quad > \quad \sup_{\forall i,j,i\neq j,\lambda \geq 0} \{\alpha_{i,out}^u(\lambda) - \alpha_{j,out}^l(\lambda)\} \tag{5}$$

where `sup` is the supremum of a set. The equation can be easily verified by applying the definition of arrival curves. Notice that (5) guarantees that there are no false-positives.

			MJPEG$_1$		MJPEG$_2$		
			Token Consumption	Token Production	Token Consumption	Token Production	Consumer Token Consumption
MJPEG Decoder	Input Encoded Frame Rate *Frame interarrival timings* 30,2,30	Network Service (Bandwidth) 500-833(KB/s)	*Frame consumption timings* 30,5,30	*Frame production timings* 30,5,30	*Frame consumption timings* 30,30,30	*Frame production timings* 30,30,30	*Frame consumption timings* 30,2,30
			ADPCM$_1$		ADPCM$_2$		
			Token Consumption	Token Production	Token Consumption	Token Production	System Data Consumption
ADPCM Application	Input Data Sample Rate *Sample interarrival timings* 6.3, 2, 6.3	Network Service (Bandwidth) 500-833(KB/s)	*Sample consumption timings* 6.3, 3.1, 6.3	*Sample production timings* 6.3, 3.1, 6.3	*Sample consumption timings* 6.3,12.3, 6.3	*Sample production timings* 6.3,12.3, 6.3	*Sample consumption timings* 6.3, 2, 6.3

(All times in milliseconds. All timings reported as <Period, Jitter, Delay> tuple. Service curves e.g., $[\alpha^u_P, \alpha^l_P]$ are derived from <Period, Jitter, Delay> tuple information)

Table 1: Parameters for Fault Tolerance Experiments

Fault Detection Times. A replica is considered to have suffered a timing fault when it fails to meet the timing properties *at its interfaces*, and *not* when a particular node(s) inside the replica may have experienced a fault. Suppose that at time s, R_1 and R_2 have produced a total of T and $T-(D-1)$ tokens respectively, when R_1 suffers a timing fault. Subsequently, R_2 must produce a $(D-1)+D = 2D-1$ tokens more than R_1 before the selector can detect a fault. Let the fault be detected at time t. For maximum fault detection time, let the replica R_2 supply tokens at the lowest possible rate, i.e., its arrival curve *subsequent* to the fault is α^l_2. Let $\overline{\alpha}^u_1$ indicate the upper arrival curve of R_1 *subsequent* to the fault, which still fails to meet the required real time constraints. The maximum time to detect the fault, relative to s is given by $\triangle$ ($\triangle = t - s$) satisfies:

$$\inf\{\triangle \mid (\alpha^l_2 - \overline{\alpha}^u_1)(\triangle) \geq (2D-1)\} \tag{6}$$

where `inf` is the infimum of a set. Generalizing the streams, the maximum fault detection time is:

$$\max_{\forall i,j,i\neq j}\{\inf\{\triangle \mid (\alpha^l_i - \overline{\alpha}^u_j)(\triangle) \geq (2D-1)\}\} \tag{7}$$

For the case when the faulty replica stops producing any tokens altogether, (7) can be simplified to:

$$\max_{\forall i}\{\inf\{\triangle \mid (\alpha^l_i)(\triangle) \geq (2D-1)\}\} \tag{8}$$

4. EXPERIMENTS AND RESULTS

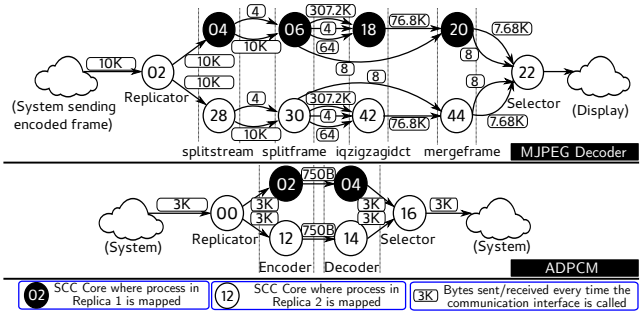

Figure 2: The MJPEG Decoder (top) and the ADPCM Application (bottom).

4.1 Hardware Platform

We used Intel's 48-core Single Chip Cloud Computer(SCC) for experiments, see [7]. Real time performance was achieved by using the SCC in the baremetal mode (i.e., without any operating system support), switching off all L2-caches and disabling all interrupts, see [14]. Furthermore, only one process was mapped *per tile* in a way which reduces cross traffic at the routers, see [13]. The SCC was booted with the following parameters: tile frequency: 533MHz, Router frequency: 800MHz, DDR3 Memory frequency: 800MHz. Furthermore, timing measurements of each core are derived from the local time stamp counter (TSC). All clocks are synchronized at application boot time in order to get valid timing results. The iRCCE non-blocking communication library was used, and

FIFO	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_1\|_0$	$\mathcal{S}_2\|_0$
Theoretical Capacity	*Capacities (tokens)* 2	3	4	6	*Initial tokens* 2	3
Max. Observed fill (No Faults, 20 runs)	1	3	1	1	–	–

Fault Detection Latency	*At Selector (ms)*		*At Replicator (ms)*	
	Min 99	*Upper Bound*	*Min* 99	*Upper Bound*
	Max 103	180	*Max* 102	180
	Mean 100		*Mean* 100	

	Selector	*Replicator*
Overhead		
Memory	2.1KB+10Tokens (0.7%)	1.5KB+5Tokens (0.5%)
Runtime	7µs (0.02%)	2.9µs (0.01%)

Decoded Inter-Frame Timings

Reference(ms)	
Min	29
Max	43
Mean	30
Duplicated(ms)	
Min	29
Max	43
Mean	30

MJPEG Decoder

FIFO	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_1\|_0$	$\mathcal{S}_2\|_0$
Theoretical Capacity	*Capacities (tokens)* 2	4	4	8	*Initial tokens* 2	4
Max. Observed fill (No Faults, 20 runs)	1	3	1	1	–	–

Fault Detection Latency	*At Selector (ms)*		*At Replicator (ms)*	
	Min 31	*Upper Bound*	*Min* 26	*Upper Bound*
	Max 38	69	*Max* 40	69
	Mean 33		*Mean* 34	

	Selector	*Replicator*
Overhead		
Memory	2.1KB+12Tokens (6%)	1.5KB+6Tokens (4.6%)
Runtime	7µs (0.1%)	2.9µs (0.05%)

Decoded Inter-Frame Timings

Reference(ms)	
Min	4.70
Max	8.25
Mean	6.18
Duplicated(ms)	
Min	4.71
Max	8.25
Mean	6.18

ADPCM

Table 2: Results for the MJPEG decoder and the ADPCM Application.

all data was sent/received in chunk sizes not exceeding 3KB, ensuring that all messages are routed exclusively via the message passing buffers, see [2, 3]. The fast on-chip communication does not significantly influence FIFO sizes or fault detection timings.

4.2 Applications

Three representative real time process network based applications were used for experiments: (a) a Motion JPEG (MJPEG) decoder (b) the Adaptive Differential Pulse Code Modulation (ADPCM) application (encoder+decoder) and (c) an H.264 encoder. The experiment was repeated with the H.264 encoder with similar results. Due to space constraints, we do not present the results in the paper. The design diversity between the replicas is captured by different jitter values, see Table 1. All timing parameters are reported as <period, jitter, delay> tuple, as is common in real time systems. In case of a fault, the faulty replica stops producing (or consuming) tokens altogether.

The MJPEG Decoder. For the fault tolerant MJPEG decoder, the input to the replicas is an encoded frame ($\sim$ 30 fps). The replicator channel duplicates each token and provides it to the splitstream process in each replica. The mergeframe process(s) provides decoded frames to the selector, 320x240 pixels each. A token at the replicator and the selector channel is one encoded and decoded frame of sizes 10KB and 76.8 KB respectively. Note that it is possible to reduce token sizes by restructuring the application: i.e., split input frames into parts, and split decoded frames into parts. However, such adjustments depend on the application and the fault-detection latency requirements and are independent of the fault tolerance framework itself. After 18,000 frames, timing faults were introduced into the duplicated network and fault detection times are reported over 20 such runs.

The ADPCM Application. The system provides one data sample to the replicator every of 3KB every $\sim$ 6.3ms. Note that the decoder rate is specifically tuned for the SCC. The

encoder performs a 4:1 compression, which is reverted by the decoder. A token at both the selector and the replicator is one data sample of size 3KB. After 20,000 samples, faults were introduced in the ADPCM network, and fault detection times for 20 such runs are summarized.

4.3 Evaluation of the Framework

The framework described in this paper is evaluated on the basis of (a) runtime overhead of the framework, (b) memory overhead of the framework (c) fault detection latencies and (d) comparison to distance function fault detection approach, see [11].

Results and Discussion. For all duplicated process networks, results in Table 2 show that under fault free conditions, the observed maximum number of *tokens* in various FIFO queues is below theoretically computed capacities (*Theoretical Capacity* vs. *Max. Observed Fill*) validating the calculations presented in Section 3.4. The framework is extremely light, in both runtime and memory overhead. For example, the memory overhead in the case of the duplicated MJPEG decoder is 0.7 % and 0.5 % of the application code at the replicator and the selector channel respectively (excluding token storage, which depends on the application). The corresponding time overhead is at most 0.02% of the decoder inter-frame period. The overhead is practically found to be small enough that the duplicated and reference process networks can provide similar runtime performance. For example, for the MJPEG decoder, the decoded frame rate is almost identical (differences due to runtime overhead are in the order of microseconds) for both the reference and the duplicated process networks. Similar results hold for other applications. The framework detects faults within the bounds computed in Section 3.4, as can be seen by comparing fault detection latency statistics for each application vs. the computed upper bound. For instance, for the MJPEG decoder, the maximum latency for detecting a fault was found to be 103ms at the replicator channel, well within the computed upper bound of 180ms. Similarly, the maximum fault detection latency at the selector channel was found to be 102ms against the expected upper bound of 180ms. Notice that the in practical situations (i.e., in the experiments), the actual faults are detected much faster than the computed worst case bounds, since worst cases are only rarely encountered. Notice that the upper bounds for fault detection latency are not always symmetrical (e.g., the H.264 application). Also note that the selector and the replicator can *independently* detect faulty replicas as proposed in the paper.

Brief Comparison to the State-of-the-Art. We present a brief comparison to distance function approach as it is superior to the simple watchdog method. For fault monitoring at the replicator, timing variations from the replicas were minimized, enabling the distance function to be implemented with $l = 1$. For monitoring at the selector side, the timing variations from the consumer was removed (replicas may have timing jitters). The fault-monitor itself was slightly modified to take into account the fail-silent fault model assumed in this paper. The results comparing fault detection latencies at the replicator are summarized in Table 3. Notice that the performane of our method is similar to distance function method *without requiring any runtime timer support.* The fault detection latencies at the selector are similar, and therefore, are not shown.

Brief Discussion. Note that the fault detection latencies using the detection approach is always greater than our method. This is solely due to the choice of having a 1ms polling interval and having non-integer application periods (e.g., 6.3ms for the ADPCM application). In principle, it is possible to set the polling interval at a finer granularity, but at the cost higher resource overhead. In summary, at the cost of four timers (two at the replicator and two at the selector) and some modifications to the distance function approach, both fault detection techniques are equivalent.

Application	Fault Detection Latency(ms)					
	Distance Function Approach			Our Approach		
	Max	*Min*	*Mean*	*Max*	*Min*	*Mean*
MJPEG Decodper	48.2	48.1	48.1	47.1	47.0	47.0
ADPCM Application	7.3	7.1	7.2	6.3	6.3	6.3
H.264 Encoder	31.4	31.2	31.3	30.4	30.1	30.3

Table 3: Comparison of our proposed approach with distance function approach.

5. CONCLUDING REMARKS

We presented efficient (i.e., memory and runtime overhead) arbitration logic (the selector and the replicator channels) together with simple timing fault-detection strategies to construct a fault-tolerant real time process network. We showed that the fault tolerant network is equivalent in functionality and timing to the original process network it was derived from. Our approach is scalable, since it is based on (already available) timing models of real time applications. The proposed fault detection framework was validated by extensive experiments on the state-of-the-art many core processor, the Intel SCC. We are also grateful to Mark Aughenbaugh from Intel IT client services for the support he has extended to us for the SCC processor.

6. REFERENCES

[1] Chakraborty, S. et.al. Interface-based rate analysis of embedded systems. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 25–34, 2006.

[2] Clauss, C. et. al. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011.*, pages 525–532, 2011.

[3] Devendra Rai et. al. Designing Applications with Predictable Runtime Characteristics for the Baremetal Intel SCC. *Runtime and Operating Systems for the Many-core Era (ROME)*, 2013.

[4] Goseva-Popstojanova, K et. al. Performability and reliability modeling of n version fault tolerant software in real time systems. In *Proc. 23rd EUROMICRO Conference*, pages 532–539, 1997.

[5] Hagbae Kim et. al. Evaluation of fault tolerance latency from real-time application's perspectives. *Computers, IEEE Transactions on*, pages 55–64, 2000.

[6] Hopkins, A.L., Jr. A highly reliable fault-tolerant multiprocess for aircraft. *Proc. IEEE*, pages 1221–1239, 1978.

[7] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proc. ISSCC*, pages 108–109, 2010.

[8] Ian A. Troxel et.al. Reliable management services for cots-based space systems and applications. In *Proc. International Conference on Embedded Systems &Applications*, pages 169–175, 2006.

[9] Meng Guo et. al. Distributed real-time fault detection and isolation for cooperative multi-agent systems. In *American Control Conference (ACC), 2012*, pages 5270–5275, 2012.

[10] B. D. Milburn. Apparatus and method for initializing a master/checker fault detecting microprocessor, 1998.

[11] Neukirchner, M. et. al. Monitoring arbitrary activation patterns in real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 293–302, 2012.

[12] P.R. Croll et. al. Dependable, intelligent voting for real-time control software. *Engineering Applications of Artificial Intelligence*, pages 615 – 623, 1995.

[13] Zimmer, C. et. al. Low contention mapping of real-time tasks onto tilepro 64 core processors. In *Proc. Real-Time and Embedded Technology and Applications Symposium*, pages 131–140, 2012.

[14] M. Ziwisky et al. BareMichael: A Minimalistic Bare-Metal Framework for the Intel SCC. In *Proc. MARC*, pages 66–71, 2012.