Middleware for Resource-Aware Deployment and Configuration of Fault-Tolerant Real-time Systems

Jaiganesh Balasubramanian[†], Aniruddha Gokhale[†], Abhishek Dubey[†], Friedhelm Wolf[†],
Chenyang Lu[‡], Christopher Gill[‡], Douglas C. Schmidt[†]
[†]Department of EECS, Vanderbilt University, Nashville, TN, USA
[‡]Department of CSE, Washington University, St. Louis, MO, USA

***Abstract*—Developing large-scale distributed real-time and embedded (DRE) systems is hard in part due to complex deployment and configuration issues involved in satisfying multiple quality for service (QoS) properties, such as real-timeliness and fault tolerance. Effective deployment requires developing and evaluating a range of task allocation algorithms that satisfy DRE QoS properties while reducing resources usage. Effective configuration requires automated tuning of middleware QoS mechanisms to avoid tedious and error-prone manual configuration.**

This paper makes three contributions to the study of deployment and configuration middleware for DRE systems that satisfy multiple QoS properties. First, it describes a novel task allocation algorithm for passively replicated DRE systems to meet their real-time and fault-tolerance QoS properties while consuming significantly less resources. Second, it presents the design of a strategizable allocation engine that enables application developers to evaluate different allocation algorithms. Third, it presents the design of a middleware-agnostic configuration framework that uses allocation decisions to deploy application components/replicas and configure the underlying middleware automatically on the chosen nodes. These contributions are realized in the DeCoRAM (Deployment and Configuration Reasoning and Analysis via Modeling) middleware. Empirical results on a distributed testbed demonstrate DeCoRAM's ability to handle multiple failures and provide efficient and predictable real-time performance.

I. Introduction

Distributed real-time and embedded (DRE) systems operate in resource-constrained environments and are composed of tasks that must process events and provide soft real-time performance. Examples include shipboard computing environments; intelligence, surveillance and reconnaissance systems; and smart buildings. A second key quality of service (QoS) attribute of these DRE systems is *fault-tolerance* since system unavailability can degrade real-time performance and usability.

Fault-tolerant DRE systems are often built using *active* or *passive* replication [1], [2]. Due to its low resource consumption, passive replication is appealing for soft real-time applications that cannot afford the cost of maintaining active replicas and do not require hard real-time performance [3]. Despite improving availability, however, server replication invariably *increases* resource consumption, which is problematic for DRE systems that place a premium on minimizing the resources used [4].

To address these concerns, DRE systems require solutions that can exploit the benefits of replication, but share the available resources amongst the applications efficiently (*i.e.*, to minimize the number and capacities of utilized resources). These solutions must also provide both timeliness and high availability assurances for applications. For a class of DRE systems that are *closed* (*i.e.*, the number of tasks, their execution patterns, and their resource requirements are known ahead-of-time and are invariant), such solutions may be determined at design-time, which in turn can assure QoS properties at runtime.

The advent of middleware that supports application-transparent passive replication [5], [6], [2], [7] appears promising to provide such design-time QoS solutions for fault-tolerant DRE systems. Unfortunately, conventional passive replication schemes incur two challenges for resource-constrained DRE systems: (1) the middleware must generate the right replica-to-node mappings that meet both fault-tolerance and real-time requirements with a minimum number of nodes, and (2) the replica-to-node mapping decisions and QoS needs must be configured within the middleware. Developers must otherwise manually configure the middleware to host applications, which requires source code changes to applications whenever new allocation decisions are made or existing decisions change to handle new requirements. Due to differences in middleware architectures, these *ad hoc* and manual approaches are neither reusable nor reproducible, so this tedious and error-prone effort must be repeated.

To address the challenges associated with passive replication for DRE systems, this paper presents a resource-aware deployment and configuration middleware for DRE systems called DeCoRAM (*Deployment and Configuration Reasoning and Analysis via Modeling*). DeCoRAM automatically deploys and configures DRE systems to meet the real-time and fault-tolerance requirements via the following novel capabilities:

- **A resource-aware task allocation algorithm** that improves the current state-of-the-art in integrated passive replication and real-time task allocation algorithms [8], [9], [10], [11] by providing a novel replica-node mapping algorithm called FERRARI (*FailurE, Real-time, and Resource Awareness Reconciliation Intelligence*). The novelty of this algorithm are its simultaneous (1) *real-time awareness*, which honors application timing deadlines, (2) *failure awareness*, which handles a user-specified number of multiple processor failures by deploying multiple passive replicas such that each of those replicas can continue to meet client timing needs when processors fail while also addressing state consistency requirements,

This work is supported in part by NSF award 0915976

and (3) *resource awareness*, which reduces the number of processors used for replication.

- **A strategizable allocation engine** that decouples the deployment of a DRE system from a specific task allocation algorithm by providing a general framework that can be strategized by a variety of task allocation algorithms tailored to support different QoS properties of the DRE system. The novelty of DeCoRAM's allocation engine stems from its ability to vary the task allocation algorithm used from the feasibility test criteria.
- **A deployment and configuration (D&C) engine** that takes the decisions computed by the allocation algorithm and automatically deploys the tasks and their replicas in their appropriate nodes and configures the underlying middleware appropriately. The novelty of DeCoRAM's D&C engine stems from the design of the automated configuration capability, which is decoupled from the underlying middleware architecture.

DeCoRAM's allocation engine, and the deployment and configuration engine are implemented in $\sim$10,000 lines of C++. This paper empirically evaluates the capabilities of DeCoRAM in a real-time Linux cluster to show how its real-time fault-tolerance middleware incurs low (1) *resource consumption overhead*, where application replicas are deployed across processors in a resource-aware manner using the FERRARI algorithm, (2) *runtime processing overhead*, where failure recovery decisions are made at deployment-time, and (3) *development overhead*, where application developers need not write application-specific code to obtain a real-time fault-tolerance solution.

II. Related Work

Fault-tolerant middleware has emerged as a core platform for developing closed DRE systems. For example, MEAD [5], AQUA [7], TMO [12], Delta-4/XPA [2], ARMADA [13], and MARS [14], are fault-tolerant middleware frameworks that provide runtime replication management capabilities in a DRE system. The DeCoRAM middleware presented in this paper reduces runtime decision making overhead incurred in such middleware with a replica allocation and failover decision algorithm. Moreover, DeCoRAM reduces the manual efforts spent by developers to deploy application replicas and configure such fault-tolerant middleware so that application developers can just focus on the business logic. This section compares DeCoRAM with related work along the dimensions described below.

Replica-node mapping algorithms. Prior research on real-time fault-tolerant task allocation algorithms [15], [16], [17], [18] have focused on active replication, whose resource consumption overhead is not suitable for closed DRE systems. Research has also focused on transient failures (failures that appear and disappear quickly) in uniprocessor [19], [20] as well as multiprocessor [21], [22] environments. However, such approaches cannot provide high availability for applications in the presence of fail-stop processor failures, which is the focus of our work.

Prior work that focuses on passively replicated real-time fault-tolerant task allocation algorithms [23], [8], [9] deal with online algorithms, which incur extra overhead for closed DRE systems. In contrast, we focus on offline algorithms based on static scheduling that leverage the invariant properties of closed DRE systems, which enables us to seamlessly leverage existing operating systems schedulers. Prior research on static scheduling-based passive replication approaches [24], [11], [10] consider only one processor failure at a time.

DeCoRAM's replica allocation algorithm differs from those approaches as follows: (1) it handles multiple processor failures using passive replication while considering primary replicas, backup replicas, and state synchronization costs in the replica allocation problem, (2) it opportunistically overbooks processors with multiple backup replicas by analyzing feasible failover patterns caused due to multiple processor failures, and (3) it extends time-demand analysis [25] to meet real-time requirements both in normal conditions and after multiple processor failures.

Tools for task allocation, deployment, and configuration of DRE systems. Prior work on deployment and configuration tools for real-time systems includes VEST [26] and AIRES [27], which analyze domain-specific models of embedded real-time systems to perform schedulability analysis and provide automated allocation of components to processors. SysWeaver [28] supports design-time timing behavior verification of real-time systems and automatically generates OS interfacing code with predictable timing information for multiple target platforms.

SYNDEX [18] provides a graphical environment for automatically exploring various design space alternatives using timing analysis, active replication scheme, and simulations, and also generates a code as a real-time executive conforming to the generated schedule.

DeCoRAM differs from these approaches as follows: (1) it considers task allocation using minimal resources along with real-time (*i.e.*, response times) and fault-tolerance in a passive replication scheme (*i.e.*, replication and state synchronization) and (2) it automatically deploys and configures applications and replicas on top of fault-tolerant middleware on nodes chosen by the replica allocation algorithm.

Relation to our prior work. Our prior work on real-time fault-tolerant middleware also contains significant gaps. For example, while *Fault-tolerant, Load-aware and Adaptive middlewaRe* (FLARe) [29] maintains service availability and soft real-time performance in dynamic environments, it requires an initial deployment of replicas that are assumed to be optimally placed. Moreover, FLARe does not attempt to minimize the number of resources used; its goal is to maintain service availability and desired response times for the given number of resources. FLARe thus provides online failover in dynamic environments by changing the failover order of replicas according to the monitored utilization of resources.

The *COmponent Replication based on Failover Units* (CORFU) [30] middleware enhances FLARe to provide fault-tolerance for component-based DRE systems, specifically to support atomic failover for groups of components. Automated configuration is supported in CORFU, which is made feasible due to the declarative mechanisms supported by component middleware. CORFU's automation is limited to maintaining replica group semantics, however, and the solution is coupled to the CORBA Component Model middleware. Since CORFU is based on FLARe, it incurs the same limitations as FLARe.

DeCoRAM's FERRARI algorithm statically decides the failover order for replicas since it can leverage the invariant properties of closed DRE systems, which reduces the need for sophisticated runtime capabilities provided by FLARe. FERRARI focuses primarily on replica allocation while attempting to significantly reduce the number of resources needed. The allocation engine in DeCoRAM provides an opportunity to evaluate multiple different algorithms beyond FERRARI. Moreover, unlike CORFU, the deployment and configuration engine in DeCoRAM can work with a variety of fault-tolerant middleware and is not limited to FLARe and CORFU.

III. Problem Definition and System Model

This section defines the problem definition for our work on DeCoRAM in the context of the task and fault system models used.

A. DRE System Model

Our research focuses on a class of DRE systems where the system workloads and the number of tasks are known *a priori*. Examples include system health monitoring applications found in the automotive domain (*e.g.*, periodic transmission of aggregated vehicle health to a garage) or in industrial automation (*e.g.*, periodic monitoring and relaying of health of physical devices to operator consoles), or resource management in the software infrastructure for shipboard computing. These systems also demonstrate stringent constraints on the resources that are available to support the expected workloads and tasks.

Task model. We consider a set of N long running soft real-time tasks (denoted as $S = \{T_1, T_2, ..., T_N\}$) deployed on a cluster of hardware nodes. Clients access these tasks periodically via remote operation requests: each application T_i is associated with its worst-case execution time (denoted as E_i), its period (denoted as P_i), and its relative deadline (which is equal to its period). On each processor, the rate monotonic scheduling algorithm (RMS) [25] is used to schedule each task and individual task priorities are determined based on their periods. We assume that the networks within this class of DRE systems provide bounded communication latencies for application communication and do not fail or partition.

Fault model. We focus on fail-stop processor failures within DRE systems that prevent clients from accessing the services provided by hosted applications. We use *passive replication* [1] to recover from fail-stop processor failures. In passive replication, only one replica—called the primary—handles all client requests when the application state maintained at the primary replica could change. Since backup replicas are not involved in processing client's requests, their application state must be synchronized with the state of the primary replica. We assume that the primary replica (which executes for worst-case execution time E_i) uses non-blocking remote operation invocation mechanisms, such as asynchronous messaging, to send state update propagations to the backup replica, while immediately returning the response to the client.

Each backup replica of a task T_i is associated with its worst-case execution time for synchronizing state S_i, which significantly reduces the response times for clients, but supports only "best effort" guarantees for state synchronization. Replica consistency may be lost if the primary replica crashes after it responds to the client, but before it propagates its state update to the backup replicas. This design tradeoff is desirable in DRE systems where state can be reconstructed using subsequent (*e.g.*, sensor) data updates at the cost of transient degradation of services.

B. Problem Motivation and Research Challenges

The goal of DeCoRAM is to deploy and configure a passively replicated DRE system of N tasks that is tolerant to at most K fail-stop processor failures, while also ensuring that soft real-time requirements are met. To satisfy fault tolerance needs, no two replicas of the same task can be collocated. To satisfy real-time requirements, the system also must remain schedulable. These goals must be achieved while reducing resource utilization. To realize such a real-time fault-tolerant DRE system, a number of research questions arise, which we examine below via an example used throughout the paper.

Consider a sample task set with their individual periods, as shown in Table I. Assuming that the system being deployed

Task	E_i	S_i	P_i	Util
<A1,A2,A3>	20	0.2	50	40
<B1,B2,B3>	40	0.4	100	40
<C1,C2,C3>	50	0.5	200	25
<D1,D2,D3>	200	2	500	40
<E1,E2,E3>	250	2.5	1000	25

TABLE I: **Sample Ordered Task Set with Replicas**

must tolerate a maximum of two processor failures, two backup replicas of each task are needed as shown. The table also shows the execution times taken by the primary replica, the state synchronization times taken by the backup replicas, and the utilization of a primary replica.

Using bin packing algorithms [31], [32] (*e.g.*, based on first-fit allocation) and ensuring that no two replicas of the same task are collocated, we can identify the lower and upper bounds on the number of processors required to host the system. For example, Figure 1 shows the placement of the tasks, indicating a lower bound on processors that is determined using a bin packing algorithm when no faults are considered. Figure 2 shows the upper bound on processors

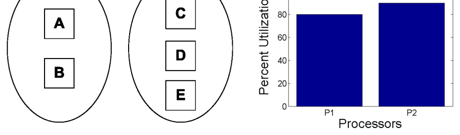


Fig. 1: **Lower Bound on Processors (No FT Case)**

needed when the system uses active replication. This case represents an upper bound because in active replication, all replicas contribute WCET. Passive replication can reduce the

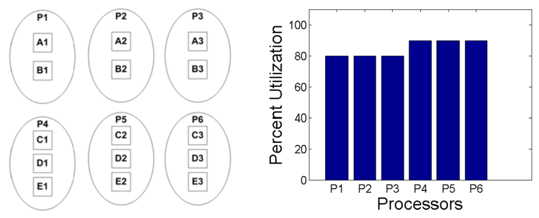


Fig. 2: **Upper Bound on Processors (Active FT Case)**

number of resources used because the backup replica in a passively replicated system only contributes to the state synchronization overhead. Naturally, the number of processors required for passive replication should be within the range identified above.

Researchers and developers must address the following questions when deploying and configuring DRE systems that must assure key QoS properties:

• How can developers accurately pinpoint the number of resources required?

• Does this number depend on the task allocation algorithm used?

• How can application developers experiment with different allocation algorithms and evaluate their pros and cons?

• How can the results of the allocations be integrated with the runtime infrastructures and how much effort is expended on the part of an application developer?

The three key challenges described below arise when addressing these questions.

Challenge 1: Reduction in resource needs. Since backups contribute to state synchronization overhead, a bin-packing algorithm can pack more replicas, thereby reducing the number of resources used. The resulting packing of replicas, however, is a valid deployment only in no-failure scenarios, which is unrealistic for DRE systems. On failures, some backups will be promoted to primaries (thereby contributing to WCET). Bin packing algorithm cannot identify which backups will get promoted, however, since failures are unpredictable and these decisions are made entirely at runtime. What is needed, therefore, is the ability to identify *a priori* the potential failures in the system and determine which backups will be promoted to primaries so as to determine the number of resources needed. Section IV-A describes an algorithm that uses the bounded and invariant properties of closed DRE systems to address this challenge in a design-time algorithm.

Challenge 2: Ability to evaluate different deployment algorithms. An algorithm for task allocation has limited benefit if there is no capability to integrate it with production systems where the algorithm can be executed for different DRE system requirements. Moreover, since different DRE systems may impose different QoS requirements, any one allocation algorithm is often limited in its applicability for a broader class of systems. What is needed, therefore, is a framework that can evaluate different task allocation algorithms for a range of DRE systems. Section IV-B discusses how the DeCoRAM framework evaluates different task allocation algorithms.

Challenge 3: Automated configuration of applications on real-time fault-tolerant middleware. Even after the replica-to-node mappings are determined via task allocation algorithms, these decisions must be enforced within the runtime middleware infrastructure for DRE systems. Although developers often manually configure the middleware, differences in middleware architectures (*e.g.*, object-based vs. component-based vs. service-based) and mechanisms (*e.g.*, declarative vs. imperative) make manual configuration tedious and error-prone. What is needed, therefore, is a capability that can (1) decouple the configuration process from the middleware infrastructure and (2) seamlessly automate the configuration process. Section IV-C describes how the DeCoRAM configuration engine automates the configuration process.

IV. The Structure and Functionality of DeCoRAM

This section presents the structure and functionality of DeCoRAM and shows how it resolves the three challenges described in Section III-B.

A. *DeCoRAM's Resource-aware Task Allocation Algorithm*

Challenge 1 described in Section III-B is a well-known NP-hard problem [16], [18], [31]. Although this problem is similar to bin-packing problems [31], it is significantly harder due to the added burden of satisfying both fault-tolerance and real-time system constraints. We developed an algorithm called *FailurE, Real-time, and Resource Awareness Reconciliation Intelligence* (FERRARI) presented below to satisfy the real-time and fault-tolerance properties of DRE systems while reducing resource utilization. FERRARI is explained using the sample task set shown in Table I.

*1) **Allocation Heuristic**:* Algorithm 1 describes the design of DeCoRAM's replica allocation algorithm called FERRARI. Line 3 replicates the original task set corresponding to the *K* fault tolerance requirements, and orders these tuples according to the task ordering strategy (Line 4). For example, to tolerate two processor failures, tasks could be ordered by RMS priorities and the resulting set could contain tasks arranged with tuples from highest priority to lowest as shown in a sample task set of Table I.

Lines 5 and 6 show how FERRARI allocates a task and all of its *K* replicas before allocating the next task. For example, for the set of tasks in Table I, first all replicas belonging to task A will be allocated followed by B and so on. To allocate each replica, FERRARI selects a candidate processor based on the configured bin-packing heuristic (Line 8). To satisfy fault-tolerance requirements, FERRARI ensures that the processor does not host another replica of the same application being allocated when selecting a candidate processor.

For the candidate processor, FERRARI runs a feasibility test using novel enhancements to the well-known time-demand analysis [25], which is used to test feasibility (see Section IV-A2). We chose the time-demand analysis for its accuracy in scheduling multiple tasks in a processor. Although the time-demand analysis method is computationally expensive, it is acceptable since DeCoRAM is a deployment-time solution.

Algorithm 1: **Replica Allocation Algorithm**

```
    Input:
      T    set of N tasks to be deployed (not including replicas),
      K    number of processor failures to tolerate,
    Output:
      Deployment plan DP    set of two tuples mapping a replica to a processor,
      P_F: resulting set of processors used
 1  begin
 2    Intially, $DP = \{\}, P_F =$ default set of one processor
 3    Let $T'$    $\{< t_{ik} >\}, 1 \leq i \leq N, 1 \leq k \leq K$ // Replicate each tasks in $T$, $K$
        times so that $T'$ contains set of $N$ $K$-tuples
 4    $Task\_Ordering(T')$ // Order the tasks and replicas
 5    foreach $tuple_i \in T', 1 \leq i \leq N$ do
 6      for $k = 1$ to $K$ do
 7        // Allocate a task and all its $K$ replicas before moving to the next
 8        Proc_Select: Pick a candidate processor $p_c$ from the set $P_F$ not yet
          being evaluated for allocation
 9        /* Check if allocation is feasible on this processor */
10        bool $result = Test\_Alloc\_for\_Feasibility(t_{ik}, k, p_c, K)$
11        if $result$==$false$ then // Infeasible allocation
12          GoTo Proc_Select for selecting the next candidate processor for
            this replica
13        else // Update the deployment plan
14          $DP$    $DP \bigcup \{< t_{ik}, p_c >\}$ // add this allocation
15        if no $p_c$ from set $P_F$ is a feasible allocation then
16          Add a new processor to $P_F$
17          GoTo Proc_Select// Attempt allocation again with the new set of
              candidate processors
18  end
```

The feasibility criteria evaluates if the replica could be allocated to the processor subject to the specified real-time and fault-tolerance constraints (Line 10). If the test fails for the current processor under consideration, a new candidate processor is chosen. For our sample task set, after deploying task sets A and B along with their replicas (as shown in Figure 3), the next step is to decide a processor for the primary replica of task C. Processor P1 is determined an infeasible

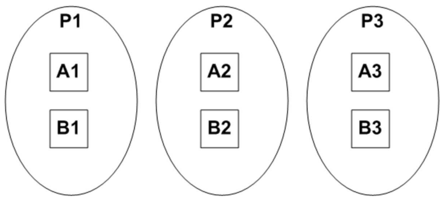


Fig. 3: **Allocation of Primary and Backup Replicas for Tasks A and B**

solution since the combined utilization on the processor would exceed 100% if C1 were allocated on P1 already hosting A1 and B1 (40+40+25=105).

If a feasible allocation is found, the output deployment plan set *DP* is updated (Line 14). If no candidate processor results in a feasible allocation, however, the set of candidate processors P_F is updated (Line 16) and the replica allocation is attempted again. As shown in Section IV-A2, C1 cannot be allocated to any of P1, P2 or P3, thereby requiring an additional processor (as shown in Figure 4). FERRARI completes after allocating all the tasks and its replicas.

*2) **Failure-Aware Look-Ahead Feasibility Algorithm**:* Challenge 1 implied exploring the state space for all possible failures in determining the feasible allocations. The time-demand analysis on its own cannot determine this state space. We therefore modify the well-known time-demand function $r_i(t)$ for task T_i in time-demand analysis [25] as follows:

$$r_i(t) = E_i + \left\{ \begin{array}{ll} \sum_{k=1}^{i-1} \lceil \frac{t}{P_k} \rceil E_k & \text{if k is primary} \\ \sum_{k=1}^{i-1} \lceil \frac{t}{P_k} \rceil S_k & \text{if k is backup} \end{array} \right\} for \quad 0 < t < P_i$$

where the tasks are sorted in non-increasing order of RMS priorities. This condition is checked for each task T_i at an instant called the *critical instant phasing* [25], which corresponds to the instant when the task is activated along with all the tasks that have a higher priority than T_i. The task set is feasible if all tasks can be scheduled under the critical instant phasing criteria.

Using this modified definition, we now enhance the feasibility test criteria using the following novel features:

(1) Necessary criteria: "lookahead" for failures. Section III-A explained how a task being allocated can play the role of a primary (which consumes worst case execution time *E*) or a backup replica (which consumes worst case state synchronization time *S*). Due to failures, some backups on a processor will get promoted to primaries and because *E* $>>$ *S*, the time-demand analysis method must consider failure scenarios so that the task allocation is determined feasible in both a non-failure and failure case. For our sample task set, this criteria implies that all possible failure scenarios must be explored for the snapshot shown in Figure 3 when allocating the primary replica for task C (*i.e.*, C1).

For any two processor failure combinations (*e.g.*, the failure of P1 and P2 or P1 and P3), the backups of tasks A and B will be promoted to being primaries. It is therefore no longer feasible to allocate C1 on either P2 or P3 (using the same reasoning that eliminated P1 as a choice). An enhancement

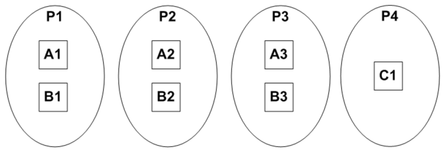


Fig. 4: **Feasible Allocation for Task C1**

to perform such a check must be made available in the time-demand analysis, which then results in an extra processor to host C1, as shown in Figure 4.

(2) Relaxation criteria: assign "failover ordering" to minimize processors utilized. Clause 1 above helps determine the placement of newly considered primaries (*e.g.*, C1). We next address the allocation of backups. One approach is to allocate C2 and C3 on processors P5 and P6 (see Figure 2). This straightforward approach, however, requires the same number of resources used in active replication, which is contrary to the intuition that passive replication utilizes fewer resources.

Using Clause 1, P1 can be eliminated as a choice to host backup C2 since a failure of P4 will make C2 a primary on P1, which is an infeasible allocation. Clause 1 provides only limited information, however, on whether P2 and P3 are acceptable choices to host backups of C (and also those of D and E since they form a group according to the first-fit criteria). We show this case via our sample task set.

Consider a potential feasible allocation in a non-failure case that minimizes resources, as shown in Figure 5. Using Clause 1, we lookahead for any 2-processor failure combinations. If

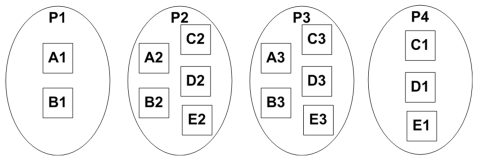


Fig. 5: **Determining Allocation of Backups of C, D and E**

P1 and P2 fail, the allocation is still valid since only A3 and B3 on P3 will be promoted to primaries, whereas C1, D1 and E1 continue as primaries on P4. If P2 and P3 were to fail, the allocation will still be feasible since the existing primaries on P1 and P4 are not affected.

An interesting scenario occurs when P1 and P4 fail. There are two possibilities for how backups are promoted. If the fault management system promotes A2 and B2 on processor P2, and C3, D3 and E3 on processor P3 to primaries the allocation will still be feasible and there will be no correlation between the failures of individual tasks and/or processors. If the fault management system promotes all of A2, B2, C2, D2 and E2 to primaries on processor P2, however, an infeasible allocation will result. The unpredictable nature of failures and decisions made at runtime is the key limitation of Clause 1.

A potential solution is to have the runtime fault management system identify situations that lead to infeasible allocations and not enforce them. The drawback with this approach, however, is that the number of failure combinations increases exponentially, thereby making the runtime extremely complex and degrading performance as the system scale increases. A complex runtime scheme is unaffordable for closed DRE systems that place a premium on resources. Moreover, despite many properties of closed DRE systems being invariant, the runtime cannot leverage these properties to optimize the performance.

It is possible to overcome the limitation of Clause 1 if the runtime fault management system follows a specific order for failovers. Our algorithm therefore orders the failover of the replicas according to their suffixes, which eliminates the possibility of infeasible allocations at design-time. Naturally, the replica-to-node mapping and hence the time-demand analysis must be enhanced to follow this ordering.

Based on this intuition, even with K processor failures it is unlikely that backups on a live processor will be promoted all at once. In other words, only a subset of backups on a given processor will be promoted in the worst case, without causing an infeasible allocation. The rest of the backups will continue to contribute only S load, which enables the overbooking of more backup replicas on a processor [23], thereby reducing the number of processors utilized.

Algorithm 2 describes the failure-aware look-ahead feasibility algorithm. The algorithm first gathers all the tasks allocated on this processor (Line 5), which may comprise a mix of primaries and backups. Next, the algorithm determines how much CPU time is consumed by the task being allocated (Line 7). If its rank is 1 (implying it is a primary), then it contributes E else it contributes S.

The algorithm executes time demand analysis method to determine the feasibility of allocating the new task in the

Algorithm 2: **Test_Alloc_for_Feasibility**

```
   Input:
      t: task $T_i$ to be allocated (including its properties, such as worst-case
      execution time $E_i$ and worst-case state synchronization time $S_i$)
      k: rank of the task being allocated ($1 \Rightarrow$ if the task will play the role of a
      primary, backup otherwise)
      p: candidate processor on which $T_i$ is allocated
      K: total processor failures to be tolerated
   Output:
      result: boolean
1  begin
2     /* Step I: Check the feasiblity of allocating $T_i$ to processor $p$ when there are no
         failures */
3     if No Failure Scenario then
4        /* Determine the primary and backup replicas that are already allocated in
            processor $p$ */
5        Let $T_p$     $\{t_a\}$ be a set of tasks that are already allocated on processor $p$
6        /* Determine the execution time of $T_i$ that needs to be used in the time
            demand analysis method */
7        if $k$ == 1 then // task is a primary
8           time = $E_i$ // task $T_i$ consumes worst-case execution time
9        else // backup case
10          time = $S_i$ // task $T_i$ consumes only state synchronization time
11       /* Execute time demand analysis method on this processor $p$ for task $T_i$ with
            the above determined time */
12       result = Time_Demand_Analysis_Method ($T_i$, time, $T_p$)
13       if result==false then
14          return false; // infeasible allocation
15    /* Step 2: Check the feasiblity of allocating $T_i$ to processor $p$ when there are $K$
         failures */
16    Let $P_t = \{\}$ // Initialize the processor set to empty
17    /* Step 3: Determine all those processors whose failure can make backups on
         this processor to become a primary */
18    foreach backup, $b \in$ processor, $p$ do // recall that the rank of backup will be 2
      or more
19       $P_t$     $P_t \bigcup \{p_{b_i}\}$ where $1 \leq i \leq rank(b) - 1$// $p_{b_i}$ is the processor hosting
         backup $b_i$
20    /* Step 4: Determine all processors hosting higher ranked backups of this task
         whose failure may trigger this task to become a primary */
21    if $k > 1$ then // check if this task is a backup
22       $P_t$     $P_t \bigcup \{p_{t_i}\}$ where $1 \leq i \leq k - 1$// $p_{t_i}$ is the processor hosting higher
         ranked task $t_i$ of task t
23    /* Step 5: (Lookahead step) Apply time demand analysis to every $K$ failure
         combination from the processor set $P_t$. Only if all succeed, the allocation is
         feasible */
24    foreach $P_{sub} \subseteq \binom{P_t}{K}$ do
25       /* For the current subset of processor failures, change roles of those
            backups in $p$ getting promoted only for this iteration. */
26       Update $T_p$
27       /* Execute time demand analysis method on this processor $p$ for task $T_i$ with
            the above determined time and $T_p$ */
28       result = Time_Demand_Analysis_Method ($T_i$, time, $T_p$)
29       if result==false then
30          return false; // infeasible allocation
31    return true // Success. Feasible allocation of $T_i$ on $p$
32 end
```

presence of all other tasks in the same processor (because tasks are allocated in non-increasing RMS priority order, applying time demand analysis is fairly simple). If this test itself fails then there is no reason to check for subsequent failure scenarios in which some of the backup replicas could become primary replicas.

If the feasibility test succeeds for the non-failure scenario, the algorithm proceeds with testing feasibility when there are at most K processor failures. This part of the algorithm includes both the necessary and relaxation criteria outlined above. There are two cases to be considered. In both these cases our goal is to identify the scenario when a backup can become a primary thereby contributing E instead of S to the execution time (recall $E >> S$).

First, the failure of K processors may trigger some of the backups on the current processor under consideration to be promoted to primary (the necessary criteria). All such processors whose failure may trigger backups on this processor to be promoted is determined (Line 19). Second, if the task being allocated is a backup, then it can become a primary if and only if all the processors hosting its higher ranked backups were to fail. The algorithm next collects all such processors (Line 22) and updates the list collected in Line 19.

So the next step is to "look ahead" for all K failure combinations from these set of processors to make sure the tasks are feasible even in the presence of K processor failures (Line 24). For every such combination (or subset) of processor failures, the algorithm determines who all in the current processor p will get promoted to a primary and updates the task set properties (specifically, the CPU time that will be used in the time demand analysis) accordingly (Line 26).

Recall that not every backup replica will get promoted to a primary replica for every processor failure, giving us a chance to overbook more backup replicas since they only consume S as opposed to E. For each such combination of processor failure, the algorithm proceeds with the time demand analysis method to check for feasibility of the allocation (Line 28). If the feasibility test succeeds for all combinations, the algorithm returns a true to the top-level algorithm described in Algorithm 1. The top level algorithm then updates the deployment plan with the replica-node mapping. As described above, for each failure combination, time-demand analysis is run; hence the algorithm first checks for feasibility in a non-failure scenario (Line 3) to save on such runs.

Figure 6 shows a feasible allocation determined by FERRARI for the sample set of tasks and their replicas, which reduces the number of resources used and supports real-time performance even in the presence of up to two processor failures.

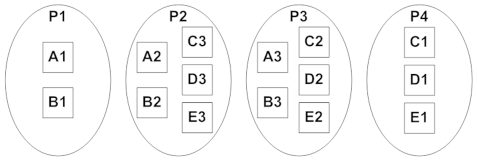


Fig. 6: **Allocation of Sample Task Set**

*3) **DeCoRAM Algorithm Complexity**:* We now briefly discuss the complexity of FERRARI. The top-level algorithm (Algorithm 1) comprises an ordering step on Line 4, which results in $O(Nlog(N)$ for N tasks. Allocation decision must then be made for each of the N tasks, their K replicas, and upto M processors if the feasibility test fails for $M-1$ processors.

The overall complexity is thus $O(N * K * M * feasibility\_test)$, where feasibility_test is the failure-aware look-ahead feasibility algorithm described in Section IV-A2. Each execution of the feasibility test requires $(1 + \binom{P_t}{K})$ executions (see Line 12 and 28 of Algorithm 2) of the enhanced time-demand analysis [25]. Since the replica allocation algorithm allocates tasks according to non-increasing RMS priority order, however, the time-demand analysis is not overly costly and can be performed incrementally.

B. DeCoRAM Allocation Engine

The FERRARI algorithm presented in Section IV-A is one of many possible task allocation algorithms that target different QoS requirements of DRE systems. Moreover, it may be necessary to decouple an allocation algorithm from the feasibility test criteria. For example, FERRARI can leverage other schedulability testing mechanisms beyond time-demand analysis. To address these variabilities, Challenge 2 in Section III-B highlighted the need for a framework to evaluate multiple different algorithms that can work with different feasibility criteria.

The DeCoRAM Allocation Engine shown in Figure 7 provides such a framework comprising multiple components, each designed for a specific purpose. DeCoRAM's Allocation

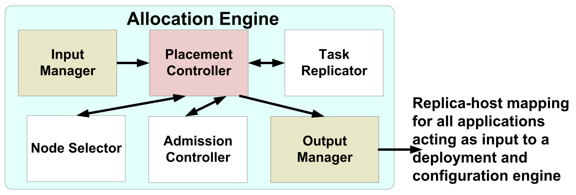


Fig. 7: **Architecture of the DeCoRAM Allocation Engine**

Engine is implemented in ∼6,500 lines of C++ and provides a *placement controller component* that can be strategized with different allocation algorithms, including FERRARI (see Section IV-A). This component coordinates its activities with the following other DeCoRAM components:

1. Input manager. DRE system developers who need to deploy a system with a set of real-time and fault-tolerance constraints express these requirements via QoS specifications that include: (1) the name of each task in the DRE system, (2) the period, worst-case execution time, and worst-case state synchronization time of each task, and (3) the number of processor failures to tolerate. Any technique for gathering these QoS requirements can be used as long as DeCoRAM can understand the information format. For the examples in this paper, we used our CoSMIC modeling tool (www.dre.vanderbilt.edu/cosmic), which supplies information to DeCoRAM as XML metadata. The input manager component parses this XML metadata into an in-memory data structure to start the replica allocation process.

2. Node selector. To attempt a replica allocation, the allocation algorithm must select a candidate node, *e.g.*, using efficient processor selection heuristics based on bin-packing [31]. The *node selector component* can be configured to select suitable processors based on first-fit and best-fit bin packing heuristics [33] that reduce the total number of processors used, though other strategies can also be configured.

3. Admission controller. Feasibility checks are required to allocate a replica to a processor. As described above, the goal of DeCoRAM's allocation algorithm is to ensure both real-time and fault-tolerance requirements are satisfied when allocating a replica to a processor. The *admission controller component* can be strategized by a feasibility testing strategy, such as our enhanced time-demand analysis algorithm (see Section IV-A2).

4. Task replicator. The *task replicator component* adds a set of K replicas for each task in the input task set and sorts the resultant task set according to a task ordering strategy to facilitate applying the feasibility algorithm by the admission controller component. Since FERRARI uses time-demand analysis [25] for its feasibility criteria, the chosen task ordering strategy is RMS prioritization, with the tasks sorted from highest to lowest rate to facilitate easy application of the feasibility algorithm. Other task ordering criteria also can be used by the task replicator component.

For the closed DRE systems that we focus on in this paper, the output from the DeCoRAM Allocation Engine framework is (1) the replica-to-node mapping decisions for all the tasks and their replicas in the system, and (2) the RMS priorities in which the primary and backup replicas need to operate in each processor. This output format may change depending on the type of algorithm and feasibility criteria used. The output serves as input to the deployment and configuration (D&C) engine (described in Section IV-C). This staged approach helps automate the entire D&C process for closed DRE systems.

C. DeCoRAM Deployment and Configuration (D&C) Engine

The replica-to-node mapping decisions must be configured within the middleware, which provides the runtime infrastructure for fault management in DRE systems. Challenge 3 in Section III-B highlighted the need for a deployment and configuration capability that is decoupled from the underlying middleware. This capability improves reuse and decouples the task allocation algorithms from the middleware infrastructure.

The DeCoRAM D&C Engine automatically deploys tasks and replicas in their appropriate nodes and configures the underlying middleware using ∼3,500 lines of C++. Figure 8 shows how this D&C engine is designed using the Bridge pattern [34], which decouples the interface of the DeCoRAM D&C engine from the implementation so that the latter can vary. In our case, any real-time fault-tolerant component middleware can serve as the implementation. By using a common interface, DeCoRAM can operate using various component middleware, such as [5], [7].

The building blocks of DeCoRAM's D&C engine are described below:

- **XML parser.** The *XML parser component* converts the allocation decisions captured in the deployment plan (which is the output of the allocation engine) into in-memory data structures used by the underlying middleware.
- **Middleware deployer.** The *middleware deployer component* instantiates middleware-specific entities on behalf of application developers, including essential building blocks of any fault tolerance solution, such as the *replication manager*, which manages the replicas; a *per-process monitor*, which checks liveness of a host; and *state transfer agent*, which synchronizes state of primary with backups.
- **Middleware configurator.** The *middleware configurator component* configures the QoS policies of the real-time fault-tolerant middleware to prepare the required operating environment for the tasks that will be deployed. Examples of these QoS policies include thread pools that are configured with appropriate threads and priorities, *e.g.*, RMS priorities for periodic tasks.
- **Application installer.** The *application installer component* installs and registers tasks with the real-time fault-tolerant middleware, *e.g.*, it registers the created object references for the tasks with the real-time fault-tolerant middleware. Often these references are maintained by middleware entities, such as the replication manager and fault detectors. Client applications also may be transparently notified of these object references.

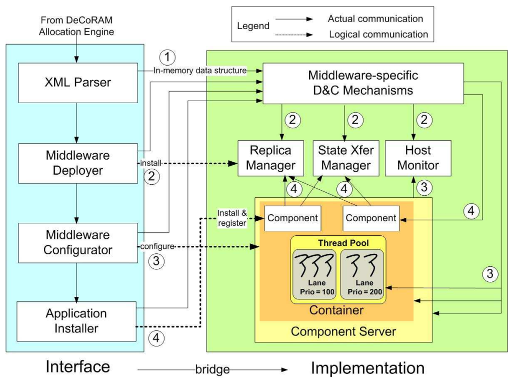


Fig. 8: **Architecture of the DeCoRAM D&C Engine**

DeCoRAM's D&C engine provides two key capabilities: (1) application developers need not write code to achieve fault-tolerance, as DeCoRAM automates this task for the application developer, and (2) applications need not be restricted to any particular fault-tolerant middleware; for every different backend, DeCoRAM is required to support the implementation of the bridge. This cost is acceptable since the benefits can be amortized over the number of DRE systems that can benefit from the automation.

V. Evaluation of DeCoRAM

This section empirically evaluates DeCoRAM along several dimensions by varying the synthetic workloads and the number of tasks/replicas.

A. Effectiveness of the DeCoRAM Allocation Heuristic

By executing FERRARI on a range of DRE system tasks and QoS requirements, we demonstrate the effectiveness of DeCoRAM's allocation heuristic in terms of reducing the number of processors utilized.

Variation in input parameters. We randomly generated task sets of different sizes N, where $N = \{10, 20, 40, 80, 160\}$. We also varied the number of failures we tolerated, K, where $K = \{1, 2, 3, 4\}$. DRE systems often consist of hundreds of applications, while passively replicated systems often use 3 replicas, which make these input parameters reflect real-world systems. For each run of the allocation engine, we varied a parameter called *max load*, which is the maximum utilization load of any task in the experiment. Our experiments varied *max load* between 10%, 15%, 20%, and 25%.

For each task in our experiments, we chose task periods that were uniformly distributed with a minimum period of 1 msec and a maximum period of 1,000 msec. After the task period was obtained, each task load was picked at random from a uniformly distributed collection with a minimum task load of 0% up to the specified maximum task load, which determines the worst-case execution times of each task.

We applied a similar methodology to pick the worst-case state synchronization times for all tasks between 1% and 2% of the worst-case execution times of each task. The deadline of each task was set to be equal to its period. Our objective in varying these parameters as outlined above was to understand how effectively DeCoRAM reduces resources and how each input parameter impacts the result.

Evaluation criteria. To determine how many resources FERRARI was able to save, we defined two baseline bounds: • *Lower bound*, where FERRARI determined the lower bound on processors needed by implementing the allocation heuristic [35] that is known to allocate tasks without fault tolerance (No-FT) in the minimal number of processors.

• *Upper bound*, where FERRARI determined the upper bound on number of processors needed by allocating *K* replicas for each task using the same heuristic [35] (we make sure that no two replicas of a task are in the same processor). This configuration represents active replication fault-tolerance (AFT) with the minimal number of processors used.

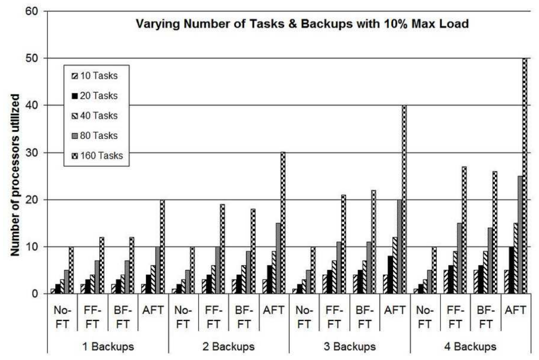
We then strategized FERRARI to use the first-fit (FF-FT) and best-fit (BF-FT) (max utilization) allocation techniques, and computed the number of processors needed. Section IV-B showed how the node selector component in the DeCoRAM Allocation Engine can be strategized with these techniques.

Analysis of results. Figures 9a, 9b, 9c, and 9d show the number of processors used when each allocation heuristic attempts to allocate varying number of tasks with varying *max load* for a task set. As *N* and *K* increase, the number of processors used also increased exponentially for *AFT*. This exponential increase in processors is due to the behavior of the active replication scheme, which executes all the replicas to provide fast failure recovery on a processor failure.

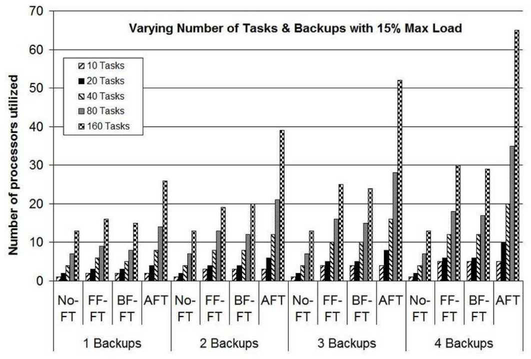
In contrast, when DeCoRAM uses the *FF-FT* or the *BF-FT* allocation heuristics, the rate of increase in number of processors used in comparison with the *No-FT* allocation heuristic is slower compared to *AFT*. For example, when *K* is equal to 1, the number of processors used by both the *FF-FT* and *BF-FT* allocation heuristics is only slightly larger than those used by the *No-FT* allocation heuristics.

As the number of tasks and processor failures to tolerate increases, the ratio of the number of processors used by the *FF-FT* and the *BF-FT* allocation heuristics to those used by the *No-FT* allocation heuristic increases, but at a much slower rate than the increase in the case of *AFT*. For large *N* and *K* (*e.g.*, see Figure 9d, 160 tasks and 4 backups for each task), the number of processors used by the *FF-FT* and the *BF-FT* allocation heuristics is only half the number of processors used by *AFT*.

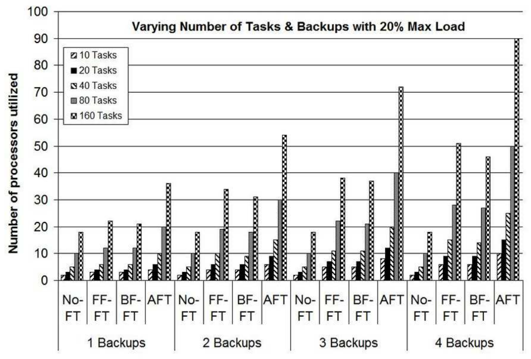
This result is a direct consequence of the relaxation criteria described in Section IV-A2. As the number of tasks to allocate and number of backup replicas increases, the look ahead step finds more opportunities for passive overbooking of backups on a processor for FF-FT and BF-FT allocation heuristics.



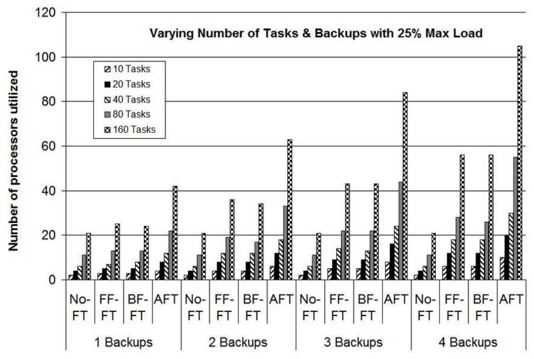
(a) Varying number of tasks and backups with 10% max load



(b) Varying number of tasks and backups with 15% max load



(c) Varying number of tasks and backups with 20% max load



(d) Varying number of tasks and backups with 25% max load

Fig. 9: **Performance of FERRARI with Varying Tasks, Backups, and Loads**

B. Validation of Real-time Performance

We now empirically validate the real-time and fault-tolerance properties of an experimental DRE system task set deployed and configured using DeCoRAM. The experiment was conducted in the ISISlab testbed (www.dre.vanderbilt.edu/ISISlab) using 10 blades (each with two 2.8 GHz CPUs, 1GB memory, and a 40 GB disk) and running the Fedora Core 6 Linux distribution with real-time preemption patches (www.kernel.org/pub/linux/kernel/projects/rt) for the kernel. Our experiments used one CPU per blade and the blades were connected via a CISCO 3750G switch to a 1 Gbps LAN.

The experimental setup and task allocation follows the model presented in Figure 6 and Table I. For our experiment we implemented the Bridge pattern [34] in the DeCoRAM D&C engine for our FLARe middleware [29]. Clients of each of the 5 tasks are hosted in 5 separate blades. FLARe's middleware replication manager ran in the remaining blade.

The experiment ran for 300 seconds. We introduced 2 processor failures (processors P1 and P2 in Figure 6) 100 and 200 seconds, respectively, after the experiment was started. We used a fault injection mechanism where server tasks call the *exit()* system call (crashing the process hosting the server tasks) while the clients CLIENT-A or CLIENT-B make invocations on server tasks. The clients receive COMM_FAILURE exceptions and then failover to replicas according to the order chosen by DeCoRAM.

Figure 10 shows the response times observed by the clients despite the failures of 2 processors. As shown by the label A in

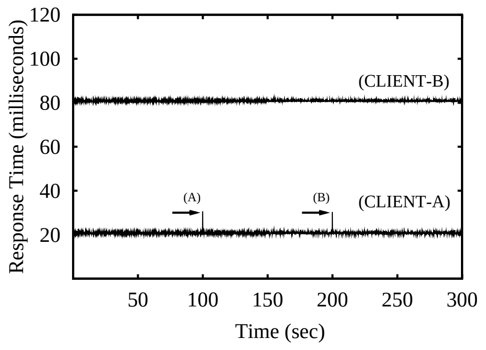


Fig. 10: **DeCoRAM Empirical Validation**

Figure 10, at 100 seconds when replica A1 fails (processor P1 fails, thereby failing B1 as well), client CLIENT-A experiences a momentary increase of 10.6 milliseconds in its end-to-end response time, which is the combined time for failure detection and subsequent failover but stabilizes immediately, thereby ensuring soft real-time requirements. The same behavior is also observed at 200 seconds (see label B) when P2 fails.

These results demonstrate that irrespective of the overbooking of the passive replicas, DeCoRAM can still assure real-time and fault-tolerance for applications.

C. Evaluating DeCoRAM's Automation Capabilities

We now define a metric that counts the number of steps per deployment and configuration activity to provide a qualitative evaluation of developer effort saved using DeCoRAM. Assuming N number of tasks, K number of failures to tolerate, and M processors needed to host the tasks, Table II shows the efforts expended by the developer in conventional approaches versus using DeCoRAM (we assume the use of our FLARe [29] real-time fault-tolerant middleware).

Activity	Effort (Steps Required)	
	Manual	DeCoRAM
Specification	N	N
Allocation	N*(K+1)	0
XML Parsing	1	0
Middleware Deployment	1 + N + 2*M	0
Middleware Configuration	M	0
Application Installation	2*N*(K+1)	0

TABLE II: **Effort Comparison**

The contents of the table are explained below. For N tasks, both the conventional and DeCoRAM approaches require developers to specify the QoS requirements. All steps in DeCoRAM are then automated and hence no effort is expended by developers. In contrast, in a manual approach, developers must determine the allocation for $K+1$ replicas (primary and K backups) of the N tasks followed by one step in parsing the XML output.

Middleware deployment requires one step in deploying the FLARe middleware replication manager, N steps to install the FLARe client request interceptors on the N clients of the servers, and 2 steps each to deploy the FLARe monitor and FLARe state transfer agent on each of the M processors. One step is then necessary to configure the underlying middleware (*e.g.*, setting up thread pools with priorities) on M processors for a total of M steps. Finally, installation of each task requires two steps to register a task with the FLARe middleware replication manager and FLARe state transfer agent for the N tasks with $K+1$ replicas each.

VI. CONCLUDING REMARKS

This paper describes the structure, functionality, and performance of the DeCoRAM deployment and configuration framework, which provides a novel replica allocation algorithm called FERRARI that provides real-time and fault-tolerance to closed DRE systems while significantly reducing resource utilization. DeCoRAM also provides a strategizable allocation engine that is used to evaluate FERRARI's ability to reduce the resources required in passively replicated closed DRE systems. Based on the decisions made by FERRARI, DeCoRAM's deployment and configuration engine automatically deploys application components/replicas and configures the middleware in the appropriate nodes, thereby eliminating manual tasks needed to implement replica allocation decisions. The results from our experiments demonstrate how DeCoRAM provides cost-effective replication solutions for resource-constrained, closed DRE systems.

Below is a summary of lessons learned from our work developing and empirically evaluating DeCoRAM:

- DeCoRAM requires a small number of additional processors to provide fault-tolerance, particularly for smaller number of processor failures to tolerate, *i.e.*, smaller values of K.
- As loads contributed by individual tasks increases, the gains in processor reduction increases when compared with active replication since DeCoRAM exploits the failover order of backup replicas to overbook multiple backup replicas whose ranks are high and whose lower

ranked replicas are deployed across different processors.

- The gains seen by FERRARI hold when the state synchronization overhead is a small fraction of the worst case execution time. As the state synchronization overhead approaches 50% or more of the WCET, the reduction seen in processors consumed is no longer attractive, which indicates that such DRE systems may benefit from using active replication.

DeCoRAM is available in open-source format at www.dre.vanderbilt.edu/~jai/DeCoRAM.

References

[1] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-backup Approach," in *Distributed systems (2nd Ed.)*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 199–216.

[2] D. Powell, "Distributed Fault Tolerance: Lessons from Delta-4," *IEEE Micro*, vol. 14, no. 1, pp. 36–47, 1994.

[3] P. Felber and P. Narasimhan, "Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems," *Computers, IEEE Transactions on*, vol. 54, no. 5, pp. 497–511, May 2004.

[4] M. Broy, "Challenges in Automotive Software Engineering," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. Shanghai, China: ACM, 2006, pp. 33–42.

[5] P. Narasimhan, T. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: support for Real-Time Fault-Tolerant CORBA." *Concurrency - Practice and Experience*, vol. 17, no. 12, pp. 1527–1545, 2005.

[6] H. Zou and F. Jahanian, "A Real-time Primary-backup Replication Service," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 6, pp. 533–548, 1999.

[7] Y. Ren, D. Bakken, T. Courtney, M. Cukier, D. Karr, P. Rubel, C. Sabnis, W. Sanders, R. Schantz, and M. Seri, "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Computers, IEEE Transactions on*, vol. 52, no. 1, pp. 31–50, 2003.

[8] Q. Zheng, B. Veeravalli, and C.-K. Tham, "On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational Grids with Low Replication Costs," *Computers, IEEE Transactions on*, vol. 58, no. 3, pp. 380–393, March 2009.

[9] W. Sun, Y. Zhang, C. Yu, X. Defago, and Y. Inoguchi, "Hybrid Overloading and Stochastic Analysis for Redundant Real-time Multiprocessor Systems," in *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, Oct. 2007, pp. 265–274.

[10] X. Qin, H. Jiang, and D. R. Swanson, "An Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks with Precedence Constraints in Heterogeneous Systems," in *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*. IEEE Computer Society, 2002, p. 360.

[11] A. A. Bertossi, L. V. Mancini, and F. Rossini, "Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 9, pp. 934–945, 1999.

[12] K. H. K. Kim and J. J. Liu, "Techniques for Implementing Support Middleware for the PSTR Scheme for Real-Time Object Replication," *ISORC*, vol. 00, pp. 163–172, 2004.

[13] T. F. Abdelzaher, S. Dawson, W. cheng Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. G. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron, "ARMADA Middleware and Communication Services," *Real-Time Systems*, vol. 16, no. 2-3, pp. 127–153, 1999.

[14] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, vol. 09, no. 1, pp. 25–40, 1989.

[15] S. Gopalakrishnan and M. Caccamo, "Task Partitioning with Replication upon Heterogeneous Multiprocessor Systems," in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–207.

[16] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and S.-Y. Tseng, "Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems," in *RTAS '07: Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–258.

[17] P. Emberson and I. Bate, "Extending a Task Allocation Algorithm for Graceful Degradation of Real-Time Distributed Embedded Systems," in *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 270–279.

[18] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel, "An Algorithm for Automatically Obtaining Distributed and Fault-tolerant Static Schedules," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, June 2003, pp. 159–168.

[19] H. Aydin, "Exact Fault-Sensitive Feasibility Analysis of Real-Time Tasks," *IEEE Trans. Comput.*, vol. 56, no. 10, pp. 1372–1386, 2007.

[20] G. de A Lima and A. Burns, "An Optimal Fixed-priority Assignment Algorithm for Supporting Fault-tolerant Hard Real-time Systems," *Computers, IEEE Transactions on*, vol. 52, no. 10, pp. 1332–1346, Oct. 2003.

[21] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Trans. on Comp.*, vol. 52, no. 2, 2003.

[22] F. Liberato, S. Lauzac, R. Melhem, and D. Mosse, "Fault Tolerant Real-time Global Scheduling on Multiprocessors," 1999, pp. 252–259.

[23] S. Ghosh, R. Melhem, and D. Mossé, "Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 3, pp. 272–284, 1997.

[24] Y. Oh and S. H. Son, "Scheduling Real-Time Tasks for Dependability," *The Journal of the Operational Research Society*, vol. 48, no. 6, pp. 629–639, 1997. [Online]. Available: http://www.jstor.org/stable/3010227

[25] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *RTSS '89: Proceedings of the IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1989, pp. 166–171.

[26] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-Based Composition Tool for Real-Time Systems," in *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Toronto, Canada: IEEE Computer Society, 2003, pp. 58–69.

[27] Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software," in *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Toronto, Canada: IEEE Computer Society, 2003, pp. 78–85.

[28] D. de Niz, G. Bhatia, and R. Rajkumar, "Model-Based Development of Embedded Systems: The SysWeaver Approach," in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–242.

[29] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," in *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS)*, San Francisco, CA, Apr. 2009, pp. 118–127.

[30] F. Wolf, J. Balasubramanian, A. Gokhale, and D. C. Schmidt, "Component Replication based on Failover Units," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '09)*, Aug. 2009, pp. 99–108.

[31] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation Algorithms for Bin Packing: A Survey," in *Approximation algorithms for NP-hard problems*. Boston, MA, USA: PWS Publishing Co., 1997, pp. 46–93.

[32] S. Dhall and C. Liu, "On a Real-time Scheduling Problem," *Operations Research*, pp. 127–140, 1978.

[33] J. M. López, M. García, J. L. Díaz, and D. F. García, "Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling," *Real-Time Syst.*, vol. 24, no. 1, pp. 5–28, 2003.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[35] S. Dhall and C. Liu, "On a Real-time Scheduling Problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.