Cost-effective Safety and Fault Localization using Distributed Temporal Redundancy

Brett H. Meyer[†*], Benton H. Calhoun[‡], John Lach[‡], and Kevin Skadron[†]
[†]Computer Science and [‡]Electrical and Computer Engineering Departments
University of Virginia
Charlottesville, VA 22904 USA
[†]{bhm, skadron}@cs.virginia.edu, [‡]{bcalhoun, jlach}@virginia.edu

ABSTRACT

Cost pressure is driving vendors of safety-critical systems to integrate previously distributed systems. One natural approach we have previous introduced is *On-Demand Redundancy* (ODR), which allows safety-critical and non-critical tasks, traditionally isolated to limit interference, to execute on shared resources. Our prior work has shown that *relaxed dedication* (RD), one ODR strategy which allows non-critical tasks (NCTs) to execute on idle critical task resources (CTRs), significantly increases NCT throughput. Unfortunately, there are circumstances under which, in spite of this opportunity, it is difficult to effectively schedule NCTs.

In this paper, we introduce *distributed temporal redundancy* (DTR), which allows critical tasks, which traditionally execute in lockstep, to execute asynchronously. In doing so, DTR increases scheduling flexibility, resulting in systems that achieve much closer to the optimal NCT throughput than with relaxed dedication alone; in one set of experiments, DTR schedules no less 93% of the theoretical NCT cycles across a variety of synthetic benchmarks, outperforming RD by over 11%, on average. Furthermore, by distributing all redundant tasks across different resources, triple-modular redundancy, and therefore fault localization, can be achieved. We demonstrate that this can be accomplished with little additional cost and complexity: in practice, relatively few DTR tasks are in flight simultaneously, limiting the additional buffering needed to support DTR.

Categories and Subject Descriptors

B.8.1 [**Hardware**]: Performance and Reliability—*Reliability, testing, and fault-tolerance*; C.3.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Real-time and embedded systems*

General Terms

Design, Reliability

Keywords

Safety-critical, system-level design, on-demand redundancy

1. INTRODUCTION

Manufacturing process scaling has made it possible to employ computer systems in almost every aspect of our lives; few devices today operate without the assistance of computers. As a natural consequence, computer systems are increasingly responsible for the control of safety-critical systems in a variety of market segments, such as medical equipment and automobiles. Manufacturing scaling, however, has also brought with it a variety of reliability challenges: the rate of single-event upsets due to radiation, for example, is expected to increase exponentially as transistors shrink [1], exacerbated by the emergence of single-event, multi-bit upsets [2]. To ensure that no harm comes to human users when such errors occur, faults must be quickly detected, and better yet, automatically corrected.

Unfortunately, fast error detection is expensive, and correction more so. For example, dual-modular redundancy (DMR) [3], a special case of n-modular redundancy (nMR), duplicates resources so that divergent architectural state (resulting from a fault) can be quickly detected. In an embedded DMR system, it is typical that pairs of critical tasks (CTs) execute on dedicated critical task resources (CTRs) resources, and in lockstep; in this way, DMR systems can, without interference from non-critical tasks (NCTs), quickly identify mismatches in the results of safety-critical computations (*e.g.*, detected at the interface to memories). While restricting task assignment and scheduling in this way makes it possible to detect failures quickly, failure localization and correction is not possible without additional costs, in the forms of more redundancy or diagnosis.

To reduce the cost of redundancy, we proposed *on-demand redundancy*, which improves NCT throughput by relaxing the requirements that CTs execute (a) on dedicated resources and (b) in lockstep [4]. One form of ODR is *relaxed dedication* (RD), which relaxes assignment restrictions so that NCTs are allowed to execute on CTR pairs when they are not performing safety-critical computations. While RD significantly increases NCT throughput in many cases, we observed that the scheduling constraint imposed by lockstep

*Meyer is now with the ECE dept. at McGill University.

CASES'11, October 9–14, 2011, Taipei, Taiwan.

execution prevents systems employing relaxed dedication from taking full advantage of the theoretical opportunity to schedule NCTs [4].

We observe that if the scheduling restrictions imposed by lockstep execution are relaxed along with the assignment restrictions of resource dedication, then system performance improves. For this purpose, we have developed a novel application of *temporal redundancy* (TR). When scheduling using temporal redundancy, two redundant copies of the same task are often scheduled on the same resource at different times, *e.g.*, so they execute one after the other. The second execution of the task is used to check the result of the first execution. When there is slack in the schedule for a multiprocessor system, this can often be accomplished with minimal impact on schedule length, extending reliability while minimizing cost increases or performance degradation [5].

We therefore propose another form of ODR, *distributed* TR (DTR). DTR relaxes the lockstep scheduling constraint, allowing redundant tasks to be arbitrarily scheduled, while constraining assignment such that each of three redundant tasks are scheduled on different resources. The third copy is scheduled to begin after the first two tasks have completed, and is only invoked if there is a mismatch; in this case, the third task executes, and voting (using values buffered from the first two executions of the task) is performed to determine (a) the location of the fault and (b) the correct output value. When the third task is not needed (the common case), NCTs can execute during its reservation.

We hypothesize that this approach increases

- scheduling flexibility relative to a system employing lockstep execution, exposing opportunity to increase utilization, or reduce cost accordingly; and,
- reliability by using DMR hardware to achieve triple-modular redundancy (TMR) at little additional cost.

In this paper, we estimate the performance and cost tradeoffs of DTR. To quantify the performance advantages of DTR, we developed a novel assignment and scheduling approach for mixes of critical and non-critical tasks, and used it to compare the performance of DTR with techniques in the literature. In one set of experiments, we observe that unlike relaxed dedication, whose behavior depends on the complexities of the underlying architecture and application, DTR consistently achieves within 93% of the optimal NCT performance, outperforming relaxed dedication by 11%.

The additional consistency and performance of DTR also comes at very low cost. We quantify the cost of DTR by examining its buffering requirements, and find that even when DTR is naïvely applied (*i.e.*, scheduling does not consider cost as a constraint), buffering for four in-flight tasks is sufficient to cover 92% of execution on average for a system with one CTR pair and two NCTRs. Buffering for six in-flight tasks increases this coverage to 98%. This suggests that it may be possible to perform cost-constrained scheduling to reduce the maximum required buffering (from eight tasks) without significantly compromising the resulting performance.

2. RELATED WORK

The reader is referred to the literature for surveys of the fundamental structures of fault-tolerant computing [6, 7, 8] and transient errors and architectures to mitigate them [9].

As multicore architectures have emerged, a number of fault-tolerance techniques have emerged to reduce the performance impact of such systems. TRUSS introduces a distributed shared memory architecture with no single point of failure [10]. To avoid common-mode failure, redundant operations are carried out by cores on different chips; however, this leads to performance losses due to long delays waiting for data to be checked. Another proposal, which filters checks from the critical path when possible and decouples checking from coherence, addresses these challenges [11]. Subramanyan, *et al.* reduce throughput losses in a CMP when a redundant thread lags the leading thread by forwarding loaded values and branch outcomes [12]. DDMR uses fingerprinting to support a technique which dynamically forms pairs of redundant processors. Sloan and Kumar developed a framework which distributes voting logic to support efficient, dynamic nMR group formation in chip multiprocessors (CMPs) [13]. Fingerprinting hashes state changes to reduce the quantity of state that is compared during checking to a single 16-bit word [14]. While our research also focuses on multi-core systems, we focus on safety-critical systems where critical tasks have hard deadlines.

A variety of techniques have been developed to address the cost of hardware redundancy. Baleani, *et al.* have investigated the trade-offs of lockstep and more loosely coupled redundant execution [3]. When cost limits redundancy to duplicated hardware or less, reliability can be categorized by their recovery mechanism: limited hardware or software replication [15, 5], re-execution [16], checkpointing [17], or some combination [18]. These efforts all reclaim or reduce the cost of explicit hardware redundancy, either working with it or replacing it; unlike any of these efforts, our research specifically focuses on the interaction of critical and non-critical tasks, reducing the overhead of redundancy by increasing NCT execution.

Several reliability techniques use sets of static schedules to dynamically respond to failure [4, 16, 19, 20]. Another approach adjusts a static schedule at runtime to enhance reliability, allowing tasks to re-execute by moving the execution of other tasks [21]. More recent research has investigated employing dynamic voltage and frequency scaling to improve energy efficiency in the average case when task re-execution is not necessary [22]. Other research statically schedules retry slots so tasks can re-execute as needed without affecting the execution of other tasks in the system [20]. We use static schedules, but aggressively schedule non-critical tasks during critical task retry reservations, pre-empting NCTs in the rare event that a failure occurs [4].

Our work is not the first to consider systems executing a mix of critical and non-critical tasks. Izosimov, *et al.* presents a fault-tolerant scheduling technique that enforces hard deadlines and selectively enforces soft deadlines in the presence of failures on a single processing node [23]. Our research is focused on the relationships between task assignment and scheduling restrictions and performance in fault-tolerant multi-core systems.

3. BACKGROUND

Traditional DMR architectures achieve reliability at great cost. When a DMR system executes a mix of critical and non-critical tasks, critical task resources are not only duplicated, but also isolated from non-critical task resources. In this case, the load and store addresses and store data can be

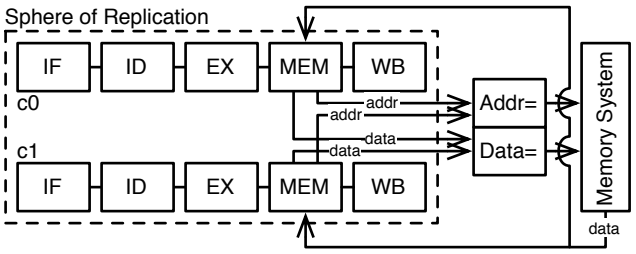


Figure 1: When tightly coupled cores execute in lockstep, each load/store address and store datum can be compared before it is passed to the memory system to be fulfilled.

compared prior to being passed on to the memory system to be fulfilled, as illustrated in Figure 1. Operations executed by CTRs c0 and c1 are assumed to be error free provided that (a) all duplicated signals entering the *sphere of replication* are equal, and (b) all duplicated signals leaving the *sphere of replication* are also equal.

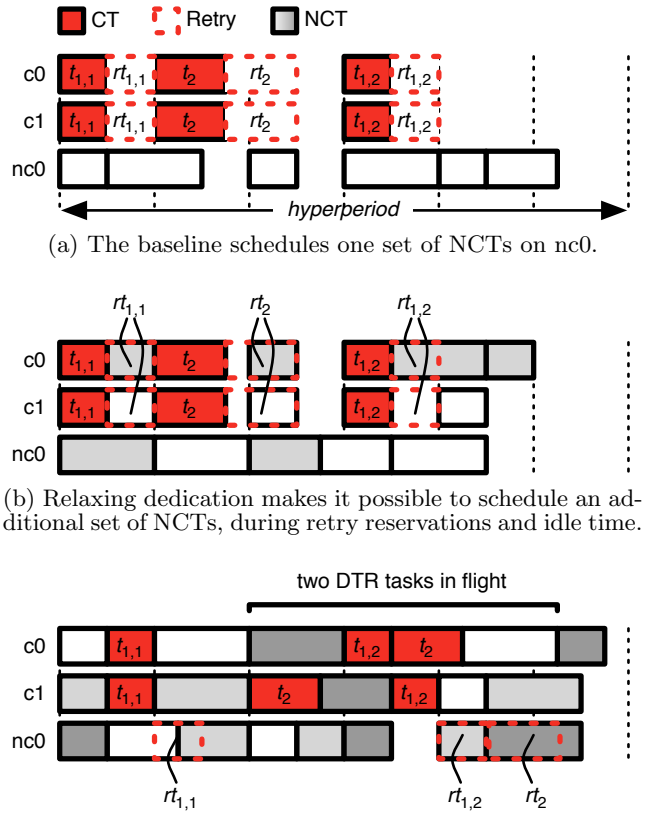
An example schedule for such a system is illustrated in Figure 2(a). All tasks assigned to the c0 and c1 are either critical tasks (filled red boxes) or retry reservations (red dashed boxes). In the baseline, NCTs (white boxes) are only assigned to NCTR nc0. nc0 is not pictured in Figure 1 since the critical and non-critical subsystems have been logically isolated to prevent interference.

If retry reservations are statically scheduled immediately following critical tasks, when a mismatch is detected (*e.g.*, due to a single event upset), tasks can be immediately re-executed [4, 16, 20, 22]. For example, if there is a failure in either copy of $t_{1,1}$, the retry pair $rt_{1,1}$ immediately re-execute. Ensuring that re-execution completes before the original task's deadline is a simple matter of enforcing this constraint at scheduling time.

Waste results in this case when the assignment restrictions that isolate tasks result in poor resource utilization. In fact, when CTRs reserve time to retry in the event of a failure, dedicated CTRs are *always* underutilized. In our example, while the NCTR nc0 is able to execute one set of NCTs, a lot of idle time remains on CTRs c0 and c1.

We have previously proposed *relaxed dedication*, which relaxes the requirement that critical task resources be used only to execute critical tasks, and shown that this optimization can substantially improve NCT throughput [4]. The schedule for such a system is illustrated in Figure 2(b). Critical tasks and retry reservations are scheduled first, to ensure that lockstep execution is preserved (as each task in each pair must execute at the same time). By scheduling NCTs (white and light gray boxes) during (a) idle time and (b) retry reservations on c0 and c1, significant opportunity to increase NCT execution is exposed. In the event a failure does occur (even with increased vulnerability due to scaling, failures are expected to be rare relative to the time scales of a single hyperperiod), NCTs that either completely or partially overlap with retry reservations (*e.g.*, $rt_{1,1}$ and rt_2, respectively) are preempted to allow the retry to execute.

In our example, this system is able to execute two (equivalent) sets of NCTs when relaxed dedication is applied, doubling the NCT throughput. However, when CT utilization is high and NCTs are long, relative to CTs, relaxed dedication is restrictive, making it difficult to schedule NCT tasks; in these cases, waste dominates once more.



(a) The baseline schedules one set of NCTs on nc0.

(b) Relaxing dedication makes it possible to schedule an additional set of NCTs, during retry reservations and idle time.

(c) Relaxing lockstep makes it possible to schedule yet another set of NCTs, through co-scheduling.

Figure 2: NCT throughput is increased over (a) the baseline by relaxing CT (b) resource dedication, and (c) lockstep.

4. DISTRIBUTED TEMPORAL REDUNDANCY (DTR)

We have developed a new technique, *distributed temporal redundancy* (DTR), which cost-effectively addresses this problem and substantially improves the consistency with which and extent to which NCT cycles can be utilized. DTR relaxes both (a) the assumption that CTs execute in lockstep, and that (b) CTs execute on CTRs.

DTR has two important results for system performance and reliability. First, when CTs need not execute in lockstep, they can be easily co-scheduled with NCTs. The resulting scheduling flexibility means that a greater fraction of available cycles can be easily utilized to execute NCTs.

Second, when a third task is executed on a third resource (*e.g.*, dedicated to non-critical tasks in the baseline), TMR is achieved without additional resources. This makes it possible to determine the origin of incorrect calculations and identify components that may be wearing out when failures recur; without TMR or complex self-test mechanisms, it is not possible to distinguish transient errors due to external events (which occur randomly in time and space) and intermittent failures due to manufacturing variability or wear-out (which recur deterministically).

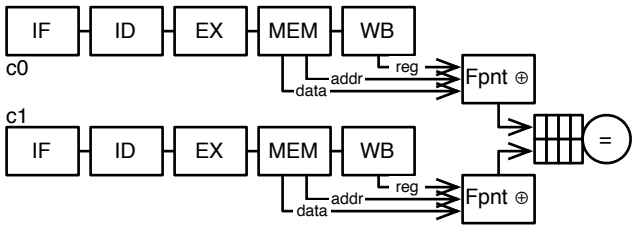


Figure 3: Changes to registers, memory, and memory addresses are accumulated in a single CRC *fingerprint*; fingerprint comparison has a high probability of detecting failures independent of the number of accumulated changes.

4.1 Performance via Flexibility

An example of the sort of schedule that can result for such a system is illustrated in Figure 2(c). Like the baseline and relaxed dedication scenarios, a pair of critical tasks always execute on c0 and c1 (*e.g.*, $t_{1,1}$). Unlike the previous cases, these tasks need not execute in lockstep (*e.g.*, $t_{1,2}$).

Relaxing the scheduling requirement that critical tasks execute in lockstep makes it possible to jointly optimize CT and NCT schedules, producing efficiencies. When CTs and NCTs are co-scheduled, it becomes possible to accommodate larger NCTs that would not otherwise not fit in the gaps between CTs, which are scheduled before NCTs under relaxed dedication. In our example, this system is able to execute three (equivalent) sets of NCTs, tripling the NCT throughput of the baseline, and improving upon relaxed dedication by 50%. We present results in this paper exploring the extent to which DTR improves upon relaxed dedication for a variety of synthetic benchmarks.

New architectural structures are needed to expose this increased flexibility. Since redundant critical tasks do not necessarily execute at the same time, the results of calculations must be buffered until all copies of a critical tasks have completed. In our example, at one point two sets of critical tasks are in-flight: results for $t_{1,2}$ and t_2 must be simultaneously buffered while the second (and possibly, third) copy of each task finishes. We present results in this paper exploring the amount of buffering required by DTR.

4.2 DTR using Fingerprinting

While DTR is not dependent on any particular underlying implementation for result buffering and comparison, *fingerprinting* is one particularly promising method for low-overhead comparison [14]. Fingerprinting uses a cyclic redundancy check (CRC) to compress (a) changes to architectural registers, (b) new memory values, and (c) effective load and store addresses, over the course of a number of instruction executions. For example, a 16-bit CRC has a $1 - 2^{-16} = 0.99998$ probability of detecting an error; the length of instruction sequence or number of updates to the CRC have no effect on the probability of error detection. When greater confidence is needed, a 24- or 32-bit CRC can be employed to achieve detection probabilities of seven nines and nine nines respectively. Not all changes to state need to be accumulated in the CRC fingerprint, as any errors that that would eventually propagate to state outside of the core must do so through registers or memory accesses. While error detection latency increases under fingerprinting, this is mitigated by scheduling redundant tasks so they complete before the original tasks' deadline.

We assume a fingerprinting implementation of DTR that collects fingerprints from the two main critical task resources in a buffer, as illustrated in Figure 3. The buffer stores the fingerprints of completed copies of a critical task until each copy completes, at which point the fingerprints for each are compared. If the fingerprints match, one copy of changes to external state (*e.g.*, buffered in a store buffer or cache, not pictured) can be released; the other copy can be discarded. Otherwise, the execution of a third task is triggered on a third resource (not pictured), pre-empting any NCTs that have been aggressively scheduled. Once the third task completes, buffered data stored by either c0 or c1 can be released, depending on which fingerprint matches that of the third redundant task. Though the buffers added to support DTR are themselves vulnerable to transient upset, we anticipate that (a) only a limited number of DTR tasks will ever be in-flight simultaneously, and (b) what buffering is needed could be protected using ECC.

When NCTs execute on CTRs, fingerprints are not generated or compared. CT/NCT interference (*e.g.*, in the form of an NCT modifying data relating to a CT) can be prevented using standard memory protection mechanisms; static scheduling prevents timing interference.

5. PERFORMANCE ESTIMATION

In order to determine the relative costs and benefits of distributed temporal redundancy compared to (a) traditional dual-modular redundancy, (b) mission-monitor pairs [24], and (c) implementations of (a) and (b) using relaxed dedication, we developed a novel static scheduling technique to determine the extent to which each of these redundancy techniques may take advantage of opportunity to execute non-critical tasks. To estimate the number of NCT cycles that can be scheduled, we have developed a framework that (a) uses simulated annealing to iteratively permute NCT assignment and (b) subsequently performs iterative list scheduling to schedule as many NCTs as possible. As our purpose is performance estimation, the assignment/scheduling process is optimized for generating schedules with the highest utilization possible while respecting task timing constraints. Unlike traditional scheduling, in which the input task set is fixed, we schedule using a task pool which is effectively infinite; whatever technique results in the system that schedules the most NCTs while respecting the deadlines of critical tasks and NCTs alike, achieves the best performance.

5.1 Problem Definition

We assume as input

- an architecture, with M CTR pairs and N NCTRs,
- a finite set of periodic critical tasks $cts = \{ct_1, ct_2, \ldots, ct_n\}$ with hyperperiod hp,
- an infinite set of periodic non-critical tasks $ncts = \{nct_1, nct_2, \ldots, nct_i, nct_{i+1}, \ldots\}$,
- a set of restrictions on what constitutes a legal assignment of tasks $cts \cup ncts$ to processors r (*e.g.*, "non-critical tasks only to non-critical task resources"), and
- a set of restrictions on what constitutes a legal schedule (*e.g.*, "critical task pairs in lockstep").

Each periodic task $t \in cts \cup ncts$ has a period p, and is itself composed of task instances, $t = \{t_1, \ldots, t_o\}$, $o = hp/p$.

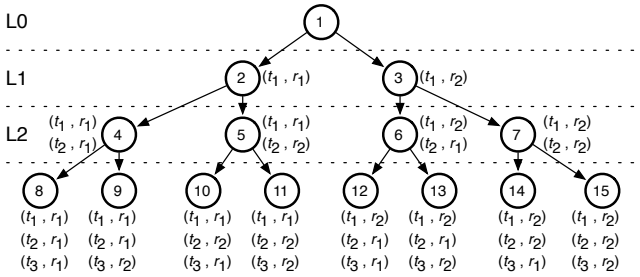


Figure 4: Complete assignment tree for three tasks, $\{t_1, t_2, t_3\}$, and two processors $\{r_1, r_2\}$. Complete assignments only appear in the last level of the tree.

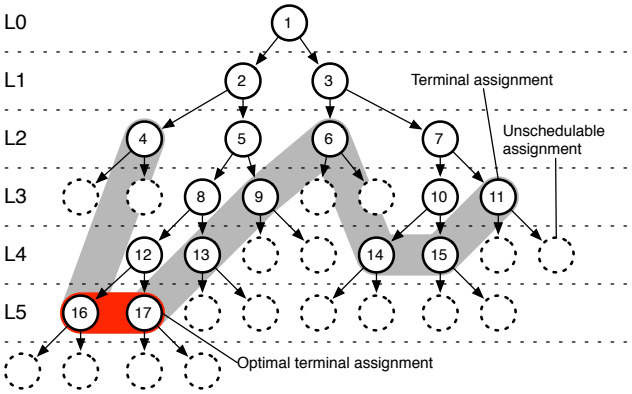


Figure 5: Performance estimation assignment tree for two resources. Unlike typical assignment trees, there are no leaf nodes; search halts when an assignment is not schedulable.

The activation time for task t_i is $i \cdot p$; the deadline for task t_i is $i \cdot p + p$. Each copy of each instance of each periodic task $t \in cts \cup ncts$ must begin after its activation time and complete prior to its deadline. The retry reservations for each CT instance must be scheduled after the CT instance finishes, and finish prior to the deadline of that CT instance. We also assume that the assignment of CT instances, their redundant pair, and their corresponding retry reservations or temporally redundant triple, are fixed (as defined by the above assignment restrictions).

In this context, our goal is to find the assignment of task instances $t_{i,j} \in ncts$ to processors in r such that the number of NCTs s that are schedulable is maximized, where $s = |ncts_s|$, $ncts_i = \{nct_1, nct_2, \ldots, nct_i\}$. Since $nct_i \subset nct_{i+1}$, incremental increases in the set of scheduled NCTs (and thus s) represent incremental increases in scheduled NCT cycles; whichever architecture maximizes s therefore maximizes the number of scheduled NCT cycles.

5.2 Assignment Annealing

We have developed an assignment technique that performs simulated annealing on a growing set of NCTs in order to maximize the number of successfully scheduled NCT tasks.

While optimal task assignment is in general NP-hard, significant progress has been made in the development of heuristics, such as branch-and-bound, which constrain the search space sufficiently to produce good results. Branch-and-bound (BNB) approaches search an *assignment tree*, illustrated in Figure 4. The root (node 1 at L0 in Figure 4) represents an empty assignment. At level i, task $t_i \in \{t_1, \ldots, t_n\}$ is assigned to a processor from the processor set $r = \{r_1, \ldots, r_m\}$. Any assignment at level $i < n$ is a partial assignment; leaf nodes (at level n) represent complete assignments. BNB techniques reduce the complexity of assignment search by aggressively pruning branches of the assignment tree by estimating (bounding) the cost of an incomplete assignment; any incomplete assignment which can be shown to produce only invalid schedules or be sub-optimal (*e.g.*, in terms of schedule length) need not be further pursued. In this way, assignment and scheduling effort is limited to those paths likely to produce good—or in some cases, even optimal—results, and very few leaf nodes are evaluated [25].

In our assignment tree, however, there are no leaves; any downward search only terminates at level i when, given a particular assignment of tasks to processors, list scheduling fails to produce a valid schedule (*i.e.*, there exists some task instance which cannot be scheduled such that its activation time and deadline are respected). Instead of attempting to limit the depth to which any particular path is explored, our technique focuses exploration on pushing forward the boundary of *terminal assignments* in search of an *optimal terminal assignment*, as illustrated in Figure 5. The children of a *terminal assignment* are all unschedulable; the assignment of any additional task results in the violation of at least one performance constraint. The level i of an *optimal terminal assignment* is greater than or equal to the level of all other terminal assignments, ensuring it has scheduled at least as many NCT cycles as any other terminal assignment.

We employ simulated annealing to permute the assignment of NCTs to processors, using a cooling schedule from the literature [26]. Because the set of NCTs is larger than what can be scheduled, annealer effort focuses on the (changing) interface in $ncts$ between successfully scheduled tasks ($ncts_i$, tasks at levels i and lower) and that just beyond ($ncts_{i+1} \setminus ncts_i$, the task at level $i+1$). For each candidate task assignment, we perform iterative scheduling to determine how many of the assigned tasks can be scheduled [4].

5.2.1 Initialization

Our annealing process begins by initializing the assignment of all critical and non-critical tasks. First, all CTs are deterministically assigned; these assignments are not subject to permutation during annealing. Restrictions on CT assignment vary from technique to technique, and are detailed in subsequent subsections.

Second, all NCTs are randomly assigned. In practice, $|ncts|$ is not infinite; in our experiments, $ncts$ is only large enough such that it is never completely scheduled. During initialization, all tasks in $ncts$ are assigned, though many tasks in $ncts$ are never successfully scheduled. Restrictions on NCT assignment also vary from technique to technique; specific details follow in subsequent sections.

5.2.2 Permutation

After initialization, our annealer repeatedly permutes and evaluates the assignment of NCTs to processors, probabilistically accepting permutations that increase assignment cost (by reducing the number of schedulable tasks).

As our approach is designed to maximize the size of the set of schedulable NCTs, $s = |ncts_s|$, our permutations focus on

exploring the interaction of currently schedulable tasks $ncts_i$ and the first task currently not schedulable $ncts_{i+1} \setminus ncts_i$. Each instance of an NCT must be schedulable for the task to be schedulable; $ncts_{i+1} \setminus ncts_i$ may therefore contain a number of tasks whose assignment can be changed. Half of all permutations change the assignment of an instance of a schedulable task in the hope of making room for an unschedulable NCT. The other half change the assignment of an instance of the unschedable NCT.

5.2.3 Evaluation

To evaluate each candidate assignment, we employed an iterative scheduling technique we previously developed [4]. Given an assignment of tasks to processors, a binary search is performed for $i = |ncts_i|$, such that the set $\{ncts_{i+1}\}$ is not schedulable. This is accomplished with repeatedly list scheduling [27], using a binary search on i, $0 \leq i < |ncts|$, to quickly identify the largest schedulable set of NCTs. i is sole variable in the objective function.

5.2.4 Cooling Schedule

Our annealer adopts the cooling schedule and related parameters proposed by Huang *et al.* [26]. This approach first performs a number of permutations to estimate the standard deviation of the objective function, which is in turn used to identify a starting temperature (in simulated annealing, the temperature determines the likelihood that a permutation that increases cost is accepted). At each temperature, the annealer performs permutations until "equilibrium" is achieved: if a minimum number of moves have been accepted, and a certain fraction of those fall within an interval around the average cost, then equilibrium has been achieved and the system can be cooled.

The annealer adaptively cools based on the standard deviation of the system cost during the prior iteration; the smaller the standard deviation, the faster the annealer cools. Annealing terminates after n iterations without significant change in the cost function; $n = 10$ in our experiments.

5.3 Baseline Systems

We consider two baseline systems. The first employs dual-modular redundancy (DMR). In DMR systems, if $t_{i,j} \in t \in cts$ is assigned to resource r, $t_{i,j}$'s redundant pair is assigned to r's redundant pair r_{DMR}, and a pair of retry reservations are assigned to r and r_{DMR} as well. Task scheduling is also restricted: $t_{i,j}$ and its redundant pair must begin execution on r and r_{DMR} at the same time; and, the corresponding two retry reservations must also begin at the same time. Retry reservations are scheduled such that they begin immediately after the original redundant tasks ends.

The second baseline is a system employing mission-monitor (MM) pairs [24]. The mission core executes the critical task, while a tightly-coupled monitor core replicates just enough of the execution of the mission core to ensure that failures are detected. The principal advantage of MM pairs is that system cost is reduced by replicating in the monitor core only that functionality which is needed to ensure the safe operation of the mission core. Toshiba reports that in its implementation, the monitor core is 58% smaller than the mission core [24]. The disadvantage of MM is that the tightly coupled monitor core cannot be used to execute NCTs.

In MM systems, there are no pairs of redundant tasks or retry reservations; a single CT instance $t_{i,j}$ is scheduled on the mission core r, and a single retry reservation is assigned to the same processor. We assume that each MM system, by virtue of achieving redundancy more cost-effectively than an equivalent DMR system, benefits from having an additional half-performance, half-area NCTR [4].

For the baseline systems, CTs and their retry reservations are scheduled first on CTRs (using list scheduling), in order to ensure that scheduling restrictions related to lockstep execution can be satisfied. Each time a CT is scheduled, a retry reservations is scheduled to immediately follow it on the same resource. CT list scheduling is performed in a single pass. Since all CTs must be schedulable for the schedule to be legal, iterative list scheduling is unnecessary.

NCTs are scheduled on NCTRs in a subsequent step. For each attempted assignment, iterative list scheduling determines the maximum number of NCTs that can be scheduled.

5.4 Relaxed Dedication

When evaluating systems with *relaxed dedication*, assignment restrictions are relaxed: while CTs and retry reservations remain assigned to CTRs, NCTs can be assigned to any resource (except the monitor in MM).

For systems with relaxed dedication, CTs and their retry reservations are still scheduled first, and in the same way as for the baseline systems. When NCTs are subsequently scheduled, they can assigned to CTRs and can be scheduled either during (a) idle time, or (b) retry reservations.

5.5 Distributed Temporal Redundancy

When evaluating systems with *distributed temporal redundancy*, assignment restrictions are changed relative to the baselines: CTs remain assigned to CTRs; for each CT, a single retry reservation (rather than a pair) is made on a single NCTR. This assignment is deterministic: the same NCTR is used for all retries for a given pair of CTRs. This reduces the cost of DTR by limiting the number of resources which require the result buffering and fingerprint generation logic described in Section 4.2. By assigning each redundant copy of a task to a different resource, majority voting can isolate the source of the failure.

For DTR systems, CTs, their retry reservations, and NCTs are scheduled simultaneously in a single, iterative list scheduling step. The assignment annealer can only change the assignment of NCTs (all other assignments are fixed). For each assignment, list scheduling attempts to co-schedule all tasks: CT pairs may not be scheduled to start at the same time, and there may be gaps between the end of CT tasks and the beginning of the retry reservation. NCTs can be scheduled at any time. Co-scheduling CTs, retry reservations, and NCTs significantly increases scheduling complexity, while exposing opportunity to improve schedule efficiency.

6. EXPERIMENTAL SETUP

We conducted experiments to compare the performance of distributed temporal redundancy, approaches from the literature [24], and our prior work [4]. We employ the assignment and static scheduling approach in Section 5 to both (a) estimate the relative performance of each technique under a variety of usage scenarios, and (b) estimate the cost of DTR by measuring the number of in-flight DTR tasks.

In our experiments, we assume a fixed system of four core equivalents, where $M = 1$ (the number of CTR pairs), and $N = 2$ (the number of NCTRs). Experimenting with het-

erogenous systems is the subject of future work. We select a mission-monitor implementation as the baseline (MM), and compare traditional dual-modular redundancy (DMR), DMR with relaxed dedication (DMR+RD), and MM with relaxed dedication (MM+RD), and our proposed approach, distributed temporal redundancy (DTR).

We randomly generated a number of benchmarks using Task Graphs for Free (TGFF) [28] in order to determine how ctf and $ctrl$ influence the relative performance of DTR and comparison approaches. In our experiments, we varied the fraction of CTR execution dedicated to critical tasks (critical task fraction), $ctf \in [0.1, 0.48]$. We also experimented with two ratios of average CT length to average NCT length (critical task length ratio), $ctlr \in \{0.4, 1\}$. Experimenting with real applications is the subject of on-going research.

6.1 Critical Task Sets

The cycles *theoretically* available to relaxed dedication to execute NCTs changes as a function of (a) redundancy type (DMR vs. MM) and (b) ctf. In our prior work, DMR with relaxed dedication performed better for low ctf, with MM achieving parity for high ctf. To explore the relative performance of DTR under variable ctf, we generated 10,000 sets of critical tasks, from which we selected 20 such that $ctf \in \{0.1, 0.12, \ldots, 0.48\}$. These different sets of tasks capture a range of reasonable applications, from those dominated by NCTs where safety is periodically monitored (*e.g.*, a tire-pressure monitoring system), to those dominated by safety-critical tasks (*e.g.*, an anti-lock braking system). Each set of CTs is composed of 8 tasks, with a task length of 20 ± 10 (uniformly distributed), and period multipliers in $\{1, 2, 5, 10, 20\}$. The CT benchmarks are composed of from 19 to 95 task instances, 47.8 on average.

6.2 Non-critical Task Sets

The cycles *practically* utilizable by RD changes as a function of (a) redundancy type (DMR vs. MM) and (b) $ctrl$. In our prior work, DMR with relaxed dedication performed better for high $ctrl$ (CTs longer than NCTs), with MM performing substantially better for low $ctrl$ (CTs shorter than NCTs). DTR is expected to perform well independent of $ctrl$, since CTs and NCTs can be simultaneously scheduled.

To explore the relative performance of DTR under variable $ctrl$, we considered two scenarios, $ctrl = 1$ (CTs are as long on average as NCTs) and $ctrl = 0.4$ (CTs are 40% as long on average as NCTs). Period multipliers are selected from the same set as for CTs. In our experiments, NCTs are independent of the CTs, under the assumption that making a CT dependent on an NCT would pose a safety risk.

When $ctrl = 0.4$, the task length of an NCT is 50 ± 10 (uniformly distributed); in this case, NCTs are *always* too long to be scheduled in retry reservations alone, allowing us to explore the relative advantages of DTR in a case when relaxed dedication is expected to perform poorly.

6.3 CT-NCT Benchmarks

In order to account for the interaction of individual sets of CTs and NCTs (where one ill-sized NCT can prevent further NCTs from being scheduled during our iterative approach), we match each CT set with n randomly generated sets of NCTs from each pool, $n \in [33, 132]$. Each NCT set from the same pool has the same parameters, but different sets can result in significantly different assignment and scheduling outcomes. The performance of a system for a given value of ctf is derived by averaging the number of cycles utilized by NCTs across each of the n samples. Larger n is used when 95% confidence intervals are large, and $n = 33$ is insufficient to distinguish the approaches statistically.

Table 1: Theoretical NCT cycles

General Eqn.		Normalized Eqn. $M = 1$, $N = 2$
W_{DMR}	$= Nft$	$= 4/5$
W_{MM}	$= \frac{Mft}{2} + Nft$	$= 1$
W_{DMR+RD}	$= 2Mft(1-c) + W_{DMR}$	$= 8/5 - 4c/5$
W_{MM+RD}	$= Mft(1-c) + W_{MM}$	$= 7/5 - 2c/5$
W_{DTR}	$= W_{DMR+RD}$	$= 8/5 - 4c/5$

7. RESULTS

The results of our two scheduling scenarios are illustrated in Figures 6 and 7. In the compilation of these results, we completed over 10,000 individual trials. Depending on the number of tasks scheduled, a single trial executed for anywhere from less than an hour to a few days (on an AMD Opteron™ 242 with 3 GB RAM); as task count increases, both assignment and scheduling complexity increase. When $ctf = 1$, the baseline and relaxed dedication approaches scheduled from 319 to 491 NCT instances on average, and up to 512 and 982 respectively. Because DTR co-schedules (a) NCTs with (b) each redundant copy of each CT, it scheduled more task instances: 694 on average, and up to 1092.

Figure 6(a) plots the normalized scheduled NCT cycles, across a variety of ctf values, when $ctrl = 1$ (CTs and NCTs are the same length, on average). Each value is normalized to the theoretical number of cycles available to MM. The error bars in Figure 6(b) indicate the theoretical maximum NCT cycles that could be exposed.

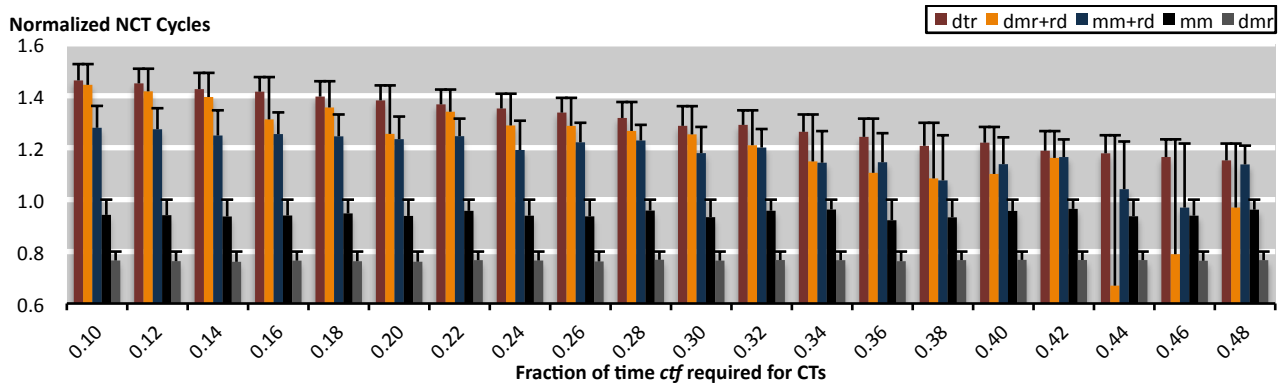
The theoretical maximums for work W (cycles) done by each technique are summarized in Table 1 [4]. The general equations are functions of the number of CTR pairs M, the number of NCTRs N, the (homogeneous) clock frequency of each resource f, the hyperperiod length t, and critical task fraction c (cft elsewhere). While DTR has the same theoretical maximum as DMR+RD, we will observe that relaxing scheduling constraints makes it easier for DTR to take advantage of opportunity to schedule NCTs. As in Figure 6(a), the normalized equations are normalized to MM.

The absolute number of NCT cycles is plotted in Figure 6(b). Different sets of columns have different magnitudes (*e.g.*, $ctf = 0.3$ and $ctf = 0.32$) because the hyperperiods of the CT sets are different. In this figure, the error bars correspond to the 95% confidence interval.

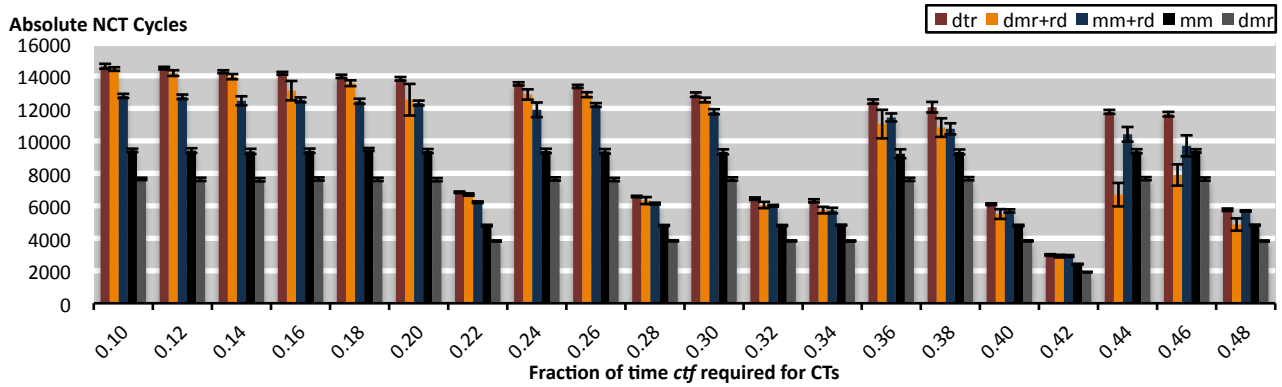
Figures 7(a) and (b) plot the normalized and absolute cycles respectively, when $ctrl = 0.4$ (CTs are 40% as long as NCTs on average).

7.1 Baseline Performance

We first observe that the baselines (DMR, MM) experience consistent, if consistently lower, NCT performance than the other techniques. All NCT cycles are scheduled on dedicated resources, significantly simplifying the assignment and scheduling process. DMR and MM capture 96% and 94% of the opportunity respectively, when $ctrl = 1$ (Figure 6). In



(a) NCT cycles normalized to those theoretically achievable by a baseline mission-monitor system. The error bar corresponds to the NCT cycles theoretically available [4].



(b) Absolute NCT cycles scheduled. Fewer cycles are scheduled for some ctf values as a result of shorter hyperperiods. Error bars correspond to the 95% confidence interval.

Figure 6: $ctrl = 1$, CTs and NCTs are the same length on average. Distributed temporal redundancy outperforms the alternative approaches across all considered mixes of critical and non-critical tasks, consistently exposing a greater share of the potential NCT cycles than the alternatives.

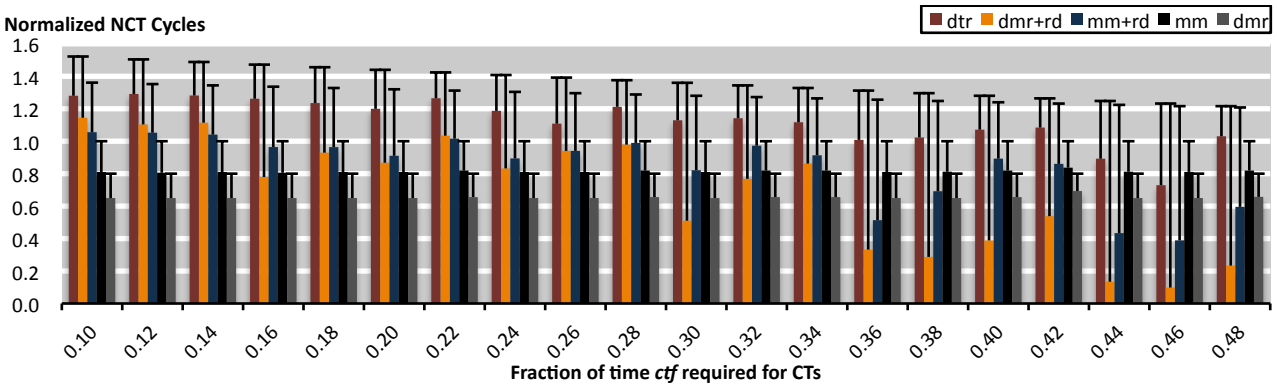
this case, the relative cost advantage of MM translates into higher NCT performance, 22% on average. When $ctrl = 0.4$ (Figure 7), it is harder to fully utilize resources dedicated to NCTs: longer tasks are more difficult to pack effectively. In this case, DMR captures 82% of the opportunity, while MM captures 81%; MM's advantage grows slightly to 26%.

Next, we observe that when DMR and MM are extended with relaxed dedication, significantly more cycles are available for NCT execution, confirming what we have previously observed [4]. When $ctrl = 1$, DMR+RD is able to schedule 56% more NCT cycles than DMR on average; MM+RD is able to schedule 25% more NCT cycles than MM. Compared across all possible values of ctf, DMR+RD and MM+RD are approximately equivalent on average, with DMR+RD having an advantage for smaller ctf (because of greater theoretical opportunity) and MM+RD having an advantage for smaller ctf (because of scheduling challenges): DMR+RD performs 8% better on average than MM+RD for $ctf < 0.3$ and 7% worse for $ctf \geq 0.3$.

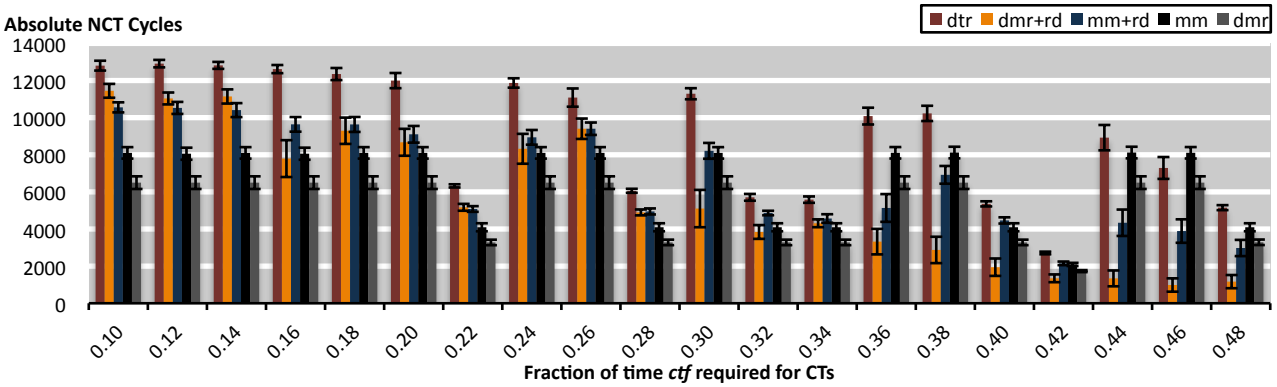
As the fraction of time devoted to critical tasks grows, both approaches struggle to schedule NCT tasks in some cases. DMR+RD performs notably poorly when $ctf = 0.44$, scheduling fewer NCT cycles than even DMR. This illustrates that while there may be more scheduling opportunity when ctf is large, it is difficult enough to find a good schedule that the added scheduling flexibility actually becomes a liability. MM+RD likewise performs poorly when $ctf = 0.46$; though the average for MM+RD is higher in this case for MM, the confidence interval around MM+RD includes the average for MM (Figure 6(b)).

DMR+RD and MM+RD perform even worse when $ctrl = 0.4$: DMR+RD schedules just 6% more than DMR on average; MM+RD schedules 4% more than MM on average. As was the case when $ctrl = 1$ and $ctf = 0.44$, it is often difficult for DMR+RD and MM+RD to take advantage of the opportunity to schedule NCTs. When $ctf \geq 0.3$, DMR+RD schedules 49% more NCTs on average than DMR; however, when $ctf < 0.3$, DMR+RD schedules 37% *less* on average. MM+RD faces a similar challenge, with a 21% advantage over MM when $ctf \geq 0.3$, and a 13% penalty otherwise.

Unlike when $ctrl = 1$, DMR+RD and MM+RD are not equivalent on average, and for essentially the same reason. DMR+RD schedules 1% fewer NCT cycles when $ctf < 0.3$, and 46% less when $ctf \geq 0.3$, on average. Because NCTs cannot be scheduled during retry reservations (on account of being too long), it is more difficult to schedule NCTs (a) in general, (b) under relaxed dedication, and (c) under DMR with relaxed dedication in particular.



(a) NCT cycles normalized to those theoretically achievable by a baseline mission-monitor system. The error bar corresponds to the NCT cycles theoretically available [4].



(b) Absolute NCT cycles scheduled. Fewer cycles are scheduled for some ctf values as a result of shorter hyperperiods. Error bars correspond to the 95% confidence interval.

Figure 7: $ctrl = 0.4$, CTs are 40% the length of NCTs on average. Even when NCTs are large relative to CTs, DTR performs well, outperforming the alternative approaches across almost all considered mixes of CTs and NCTs.

7.2 DTR Performance

Unlike the approaches employing just relaxed dedication, DTR performs consistently no matter the fraction of time dedicated to critical tasks. When $ctrl = 1$, DTR schedules NCTs in 95% of the available cycles on average, compared with 87% and 92% for DMR+RD and MM+RD respectively. Notably, DTR never takes advantage of less than 93% of the available cycles, compared with 54% and 80% for DMR+RD and MM+RD respectively. This advantage in consistency translates to an advantage in NCT scheduling of 12% and 11% compared with DMR+RD and MM+RD respectively.

When $ctrl = 0.4$, DTR schedules NCTs in 82% of the available cycles on average, compared with 49% and 65% for DMR+RD and MM+RD respectively. In this case, DTR schedules 1.38× more NCT cycles than DMR+RD, and 39% more than MM+RD. It is interesting to note that DTR's advantage over the baseline approaches is comparable; DTR schedules 72% more NCTs than DMR, and 53% more than MM. In one case, when $ctf = 0.46$, DTR actually performs slightly (11%) worse than MM; this is the one observed case when DTR fails to outperform all comparison approaches.

7.3 DTR Cost

Though DTR is not free, even when naïvely applied its hardware cost is low. At present, our performance estimation technique does not constrain the number of DTR tasks that may be in flight at any given time. In practice, scheduling must be performed under such a constraint, since buffering and comparison logic must be implemented in hardware for the sake of efficiency. We estimate the cost of DTR here by measuring the number of buffers required to support in-

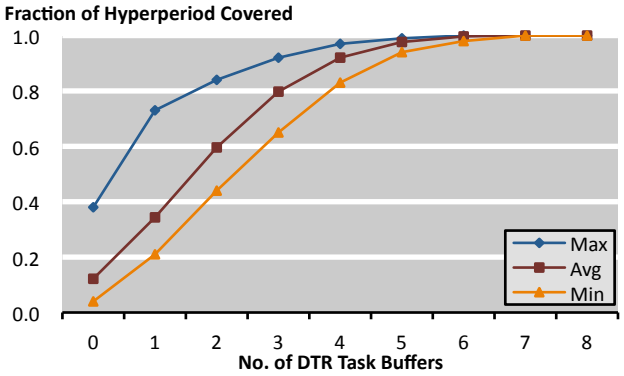


Figure 8: In the worst-case, buffering is needed for eight DTR tasks, though the majority of time four suffice.

flight CTs, under the assumption that DTR area overhead will scale approximately linearly with the number of buffers; a detailed micro architectural evaluation, while beyond the scope of this paper, is the subject of future work.

In our experiments, there were at most seven or eight DTR tasks in flight at once. On average, however, far less buffering is needed. Across all values of *ctf* and *ctrl*, the average number of tasks in flight varied from 1.17 to 2.9, with an overall average of 2.25. Figure 8 illustrates, averaged across all values of *ctf* and *ctrl*, the maximum, average, and minimum hyperperiod coverage afforded by different amounts of buffering. For example, on average (Avg), buffering for four sets of tasks is sufficient for 92% of all execution time. Even in the worst case (Min), buffering for six sets of tasks covers 98% of execution. The results charted in Figure 8 imply that there could be significant opportunity to constrain the cost of DTR buffering without dramatically reducing the benefits of DTR. Investigating this, by adding constraints to the scheduling process, is also the subject of future work.

8. CONCLUSIONS

We introduced *distributed temporal redundancy*, a form of *on-demand redundancy*. To goal of ODR is to increase NCT throughput in safety-critical systems by relaxing the traditional assumptions of (a) resource dedication and (b) lockstep execution, which restrict task assignment and critical task scheduling. Like *relaxed dedication*, DTR relaxes assignment restrictions to make use of unutilized cycles on critical task resources for executing NCTs. DTR, however, makes two important improvements over RD. First, by relaxing scheduling restrictions such that critical tasks can execute out of lockstep, DTR achieves greater scheduling flexibility, making it practical to schedule a greater fraction of the theoretically available NCT cycles. Second, by allowing retry reservations to execute on non-critical task resources, it is to determine the difference between transient and intermittent or permanent failures, but without the costs of triple-modular redundancy or sophisticated self-check logic.

Our experiments demonstrate that not only does DTR offer significant performance advantages over RD and other approaches, it does so with relatively small increases in cost and complexity. In one case, DTR schedules no less 93% of the theoretical NCT cycles across a variety of synthetic benchmarks, outperforming an approach that only relaxes assignment restrictions by over 11% across a variety of usage scenarios. While the use of fingerprints already substantially reduces the burden of redundancy checking, the fingerprint buffering required by DTR is also not significant. In the worst-case, when cost restrictions are not imposed during scheduling, we observed that buffering is required for up to eight in-flight critical tasks. However, on average far less buffering is required, with 92% of execution time covered by buffering for four tasks, on average. This suggests that even when the number of in-flight critical tasks is restricted to meet hardware constraints, significant opportunity to improve performance remains.

9. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers; their comments and insights have made the paper better. This work is supported by the Semiconductor Research Corporation through contract 2009-HJ-2042.

10. REFERENCES

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE MICRO*, vol. 25, no. 6, 2005.

[2] Nishant J. George, *et al.*, "Transient fault models and AVF estimation revisited," in *DSN'10*, June 2010.

[3] M. Baleani, *et al.*, "Fault-tolerant platforms for automotive safety-critical applications," in *CASES'03*, 2003.

[4] B. H. Meyer, *et al.*, "Reducing the cost of redundant execution in safety-critical systems using relaxed dedication," in *DATE'11*, March 2011.

[5] Y. Xie, *et al.*, "Reliability-aware co-synthesis for embedded systems," in *ASAP'04*, September 2004.

[6] V. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, July 1990.

[7] V. Prasad, "Fault tolerant digital systems," *IEEE Potentials*, vol. 8, February 1989.

[8] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, 1991.

[9] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan-Kaufmann, 2008.

[10] B. T. Gold, *et al.*, "TRUSS: a reliable, scalable server architecture," *IEEE Micro*, vol. 25, Nov.-Dec. 2005.

[11] B. T. Gold, *et al.*, "Chip-level redundancy in distributed shared-memory multiprocessors," in *PRDC'09*, 2009.

[12] P. Subramanyan, *et al.*, "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," in *DATE'10*, 2010.

[13] J. Sloan and R. Kumar, "Towards scalable reliability frameworks for error prone CMPs," in *CASES'09*, 2009.

[14] J. C. Smolens *et al.*, "Fingerprinting: bounding soft-error detection latency and bandwidth," in *ASPLOS-XI*, 2004.

[15] B. P. Dave and N. K. Jha, "COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance," *IEEE Transactions on Computer*, vol. 48, April 1999.

[16] N. Kandasamy, *et al.*, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Trans. Comput.*, vol. 52, Feb. 2003.

[17] S. Punnekkat, *et al.*, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, 2001.

[18] P. Eles, *et al.*, "Synthesis of fault-tolerant embedded systems," in *DATE'08*, 2008.

[19] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, "Off-line real-time fault-tolerant scheduling," in *PDP'01*, 2001.

[20] V. Izosimov, *et al.*, "Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems," in *DATE'06*, 2006.

[21] G. Fohler, "Adaptive fault-tolerance with statically scheduled real-time systems," in *Euromicro Real-Time Systems Workshop*, 1997.

[22] Y. Liu, *et al.*, "Scheduling for energy efficiency and fault tolerance in hard real-time systems," in *DATE'10*, 2010.

[23] V. Izosimov, *et al.*, "Scheduling of fault-tolerant embedded systems with soft and hard timing constraints," in *DATE'08*, 2006.

[24] C. Holland, "Toshiba MCU gains SIL3 and ASILD safety approval." `http://www.embedded.com/products/integratedcircuits/222400364`, January 2010.

[25] D.-T. Peng, *et al.*, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE Trans. Softw. Eng.*, vol. 23, no. 12, 1997.

[26] M.D. Huang, *et al.*, "An efficient general cooling schedule for simulated annealing," in *ICCAD'86*, October 1986.

[27] R. P. Dick and N. K. Jha, "Mocsyn: multiobjective core-based single-chip system synthesis," in *DATE'99*, 1999.

[28] R. P. Dick, *et al.*, "TGFF: Task graphs for free," in *CODES*, 1998.