Optimal Checkpoint Selection with Dual-Modular Redundancy Hardening

Shin-Haeng Kang, Hae-woo Park, Sungchan Kim, *Member, IEEE*, Hyunok Oh, *Member, IEEE*, and Soonhoi Ha, *Senior Member, IEEE*

Abstract—With the continuous scaling of semiconductor technology, failure rate is increasing significantly so that reliability becomes an important issue in multiprocessor system-on-chip (MPSoC) design. We propose an optimal checkpoint selection with task duplication hardening to tolerate transient faults. A target application is specified in a task graph, and the schedule/checkpoint placements are determined at design time. The proposed optimal algorithm minimizes the checkpoint overhead with a latency constraint. Experimental results show that the proposed algorithm effectively reduces the minimum end-to-end latency to perform a fault-tolerant schedule. In addition, the proposed algorithm dramatically decreases the checkpointing overhead on uniprocessor and multiprocessor systems compared with a greedy approach and an equidistant algorithm.

Index Terms—Checkpoint, task graph, multiprocessor, reliability, optimal algorithm

1 Introduction

As the technology scaling continues, more processing cores are being integrated into a single chip to meet the growing computing demand of modern embedded applications. On the other hand, shrinking geometries, lower voltages applied, and higher heat dissipation increase the probability of both permanent and transient faults on the system components [1], [2]. Thus reliability becomes an important issue in MPSoC design to attract extensive research attention recently.

A transient fault is usually caused by unexpected energy injection to flip the state of bits in an unpredictable way. Since transient faults will occur much more frequently than permanent faults [3], any reliable system should facilitate an efficient scheme to tolerate transient faults in the future. Although a bit-flip error is transient from the hardware point of view, the affected data value is not reliable any more, which may have a dramatic impact on overall system. We, therefore, should recover the lost data to tolerate the transient fault.

Three types of redundancies have conventionally been applied to increase the reliability of a system against transient faults. The first type is time-redundancy based on checkpointing and re-execution of tasks in case of faults [4], [5]. It is applicable only after a fault, somehow, is detected. The overhead of local fault detection is not free since it is hard to achieve high quality transient fault detection and transient faults might lead to a wrong output from the task, without causing the whole application to crush. The second type is space-redundancy based on placing redundant replicas of task executions on different cores. In these replication-based hardening techniques, a majority voting procedure is generally used to tolerate faults; if the values delivered by multiple replicas are different, majority makes the decision. To make a majority vote the number of replicas should be an odd number so that triple modular redundancy (TMR) is widely used [6], [7]. Although TMR is simple and predictable for tasks with deadlines, it requires more resources and energy than dual modular redundancy (DMR). On the other hand, DMR is only capable of detecting error since it is not possible to decide which one is correct. The last type is the information redundancy based on coding. Various error detection and recovery codes have been adopted for memory [8], [9], for data-path [10], and even for control logic [11], [12].

In this paper, we adopt a hardening technique that combines DMR and checkpointing as [13] assumes. We deploy checkpointing to serve two purposes. The first purpose is fault-detection, which is achieved by executing two replicas on two processors, and comparing the tasks/channels' states at checkpoints [14]. The second is to reduce the fault-recovery time by saving an intermediate correct state, thus rolling back to the latest checkpoint in case an error is detected.

In addition to hardware cost to implement hardware replication for DMR, the non-negligible overhead in terms of performance, memory space and power/energy consumption should be paid when the current states of tasks are tested and copied to a reliable storage, such as error-correction-code-equipped memory, at each checkpoint. Hence it is not desirable to perform too many checkpoints. On the other hand, if there are too few checkpoints, we may have to roll back too far to satisfy a given latency constraint. Consequently, this paper aims at determining the optimal placements of checkpointing for the execution of an application.

- *S.-H. Kang and S. Ha are with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-744, Korea. E-mail: {shkang, sha}@iris.snu.ac.kr.*
- *H. Park is with Google, Korea LLC, Seoul, Korea. E-mail: haewoo@google.com.*
- *S. Kim is with the Department of Computer Science and Engineering, Chonbuk National University, Jeonbuk 561-756, Korea. E-mail: sungchan.kim@chonbuk.ac.kr.*
- *H. Oh is with the College of Engineering, Hanyang University, Seoul 133-791, Korea. E-mail: hoh@hanyang.ac.kr.*

Manuscript received 25 Sept. 2013; revised 29 July 2014; accepted 4 Aug. 2014. Date of publication 18 Aug. 2014; date of current version 10 June 2015. Recommended for acceptance by D. Atienza.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2349492

We assume that an application is specified by a synchronous dataflow graph (SDFG) that is widely used for signal processing applications [15]. It enables us to estimate the performance and the resource requirements from the static scheduling result at design time. The proposed checkpoint algorithm in this paper is not only applicable to other decidable dataflow models [16], [17], but also to task graphs that can be scheduled at design time. It is also possible to convert a code written in imperative/functional languages to a dataflow model [18]. Even though the proposed technique is not applicable to all applications, there are a wide set of applications that can be specified by task graphs with a static schedule of tasks.

In this paper, we propose an optimal algorithm based on recurrence relations, which minimizes the checkpointing overhead under a given latency constraint. We will compare the proposed solution with a current state of the art and a greedy approach with synthetic benchmarks and real-life applications.

The remainder of this paper is organized as follows. In Section 2 we review the related work and summarize the unique contributions of this paper. The checkpoint selection problem is defined in Section 3. After we present the proposed optimal algorithm for uniprocessor systems in Section 4, we show how to extend it for multiprocessor systems in Section 5. Section 6 shows experimental results and Section 7 concludes the paper.

2 Related Works

There are two subjects relevant to this paper. One is DMR for error detection and the other is checkpoint/recovery technique for error correction.

For error detection several SW-based techniques such as signatures [19] and assertions [20] check the functional correctness at the high level with extra programming effort. DMR has been used to detect transient faults when proper software detection is not available. In the proposed technique, thus DMR with checkpointing is used to detect and correct transient faults.

For example, COFTA [21] introduces fault detection capabilities in an application by adding software assertions, if available, or duplicate-with-compare (DWC) tasks. It uses a clustering algorithm for the placement of assertions and checking tasks to reduce performance overhead. The optimal placement of voters and checkers by using task grouping is also explored in [22]. These works, however, are different from ours since DMR and checking are only used to support a fault detection, and the possibility of re-execution after comparing replicas is not considered. Ziv and Bruck [14] used a similar approach of DMR with checkpointing to ours. They use two types of checkpoints: compare-checkpoints (comparing the states of the redundant tasks to detect faults) and store-checkpoints (storing the states to reduce recovery time). Separating comparison and store operations allows to choose the optimal interval for each operation to improve the performance of checkpointing schemes. They, however, assumed equi-distance checkpointing where the interval between checkpoints is uniform. As the experiments will show, the equi-distance checkpointing does not guarantee good performance in terms of checkpointing overhead unless a few of tasks request extremely long WCET compared to others. In addition, they assumed a single task running on the uniprocessor. Thus their approach is not generally applicable for a modern complex multicore system.

The checkpoint/recovery scheme has a long history of extensive research to build a fault-tolerant system. An optimal decision of checkpoints is based on the assumed failure patterns. One approach is to assume a single [23] or a known upper bound of transient failures in a specific period [24]. Another is to adopt statistical failure models [25], [26]. The proposed technique belongs to the former approach. The proposed approach tolerates at most $maxF$ transient fault events and the corresponding rollbacks during a single iteration of the task graph. According to [27], Failures In Time (FIT) rate[1] of a 16 nm technology processor would be between 10^6 and 10^9, which implies that user-visible fault rate could be as high as one failure per day. Therefore, it is reasonable to consider the $maxF$ number of transient fault events at most within the end-to-end latency of an target application.

As for the checkpoint interval, most papers that find an optimal checkpointing scheme assume equidistant (periodic) checkpointing [24], [26]. Since an exact failure probability cannot be known in advance and may change at run time, the quasi-static schedules have been proposed to adjust the checkpointing frequency on-line [28], [29]. Previous works also commonly assume that checkpoints can be set at any time during the execution, not taking into account the volume of the states to be saved. In the proposed technique, however, we restrict the position of checkpoints to task boundaries. Since the execution time of a task may vary, the checkpoint interval may also vary at runtime. Checkpointing at task boundaries is also explored in previous works [30], [31]: Cummings and Alkalaj assume a coarse-grained dataflow graph as an execution model [30]. Like ours, they also save the edge buffers at the checkpoints by utilizing the characteristics of the dataflow model. However they assume that checkpoints are manually set by the programmer while we aim at finding the optimal checkpoints in a systematic way. Farquhar and Evripidou also assume a data-driven execution model where data tokens carry their own context information [31]. In their approach, however, checkpointing is performed at every actor invocation and checkpointing overhead is not considered.

Although the earlier techniques assume a uniprocessor system, the reliability problem becomes more severe on multiprocessor environments. As the number of processors increases, the probability of failure increases proportionally. Moreover, checkpointing in a multiprocessor system faces a new challenge: *consistency problem*. Depending on how to tackle the problem, checkpointing techniques for multiprocessors are categorized into a coordinated approach and an uncoordinated approach. A coordinated checkpointing approach [32], [33] guarantees a system-wide consistent state at all checkpoints. The system-wide consistent checkpoint prevents a task from proceeding with a wrong message that

1. The number of failures that can be expected in one billion (10^9) device-hours of operation.

the message sender has not sent. On the other hand, an uncoordinated approach assumes that the checkpoints of each task are executed independently of other tasks and no further information is stored on a reliable medium. Thus, a well-known domino effect [34] is likely to appear when a subset of tasks, which have to be resumed after a failure, roll back unboundedly to reach a set of mutually consistent checkpoints. Hence the cost of a fault is unknown and there is a chance of losing the whole execution. Our work belongs to the coordinated approach in that we store the checkpointed data to a reliable shared memory to maintain the global consistency. Unlike these techniques that assume distributed systems, we assume a multiprocessor system that has a shared memory to store global states. Thus the problem of managing the globally consistent states is different. It is left as a future work to apply the proposed technique to distributed systems considering non-negligible overhead of managing the consistent states.

On the other hand, multiple bit upsets (MBUs) per fault event become more prominent and no longer negligible [35]. Rossi et al. [9] analyzes the impact of several error detection codes on area overhead and memory access time for different codeword sizes and code-segment sizes, as well as their correction ability as a function of codeword/code-segment sizes. However, the number of tolerable bit errors is still limited unlike the proposed technique.

Compared with the related work, the major contributions of this paper can be summarized as follows:

1) We present an optimal algorithm to find the checkpoints with minimum checkpointing overhead under a real-time constraint, assuming that the application is specified by a task graph and task scheduling is given at compile-time. Unlike most previous work that assumes constant or negligible checkpointing overhead, we assume that the overhead depends on the volume of checkpointed global states.
2) The proposed technique is applicable for multiprocessor systems by identifying a set of consistent checkpoints for a multiprocessor schedule. We propose a tree-structured graph to describe the chronological relation between consistent checkpoints. On the other hand, most previous work on DMR and checkpointing assumes uniprocessor systems or makes an unrealistic assumption that equidistance checkpointing has no problem of consistency.
3) In the proposed technique, checkpointing is performed at the task execution boundary. This restriction eliminates uncertainty caused by temporary states, and reduces the problem space. Through the extensive experiments, we show that the proposed technique outperforms the previous state-of-art techniques which allow to do checkpoints during the task execution.
4) Last, the proposed DMR with checkpointing scheme can correct multiple bit upsets, even up to entire bits of local data memory. In addition, it can endure multiple events occurring during the application execution with the overhead proportional to the maximum number of tolerable fault events.

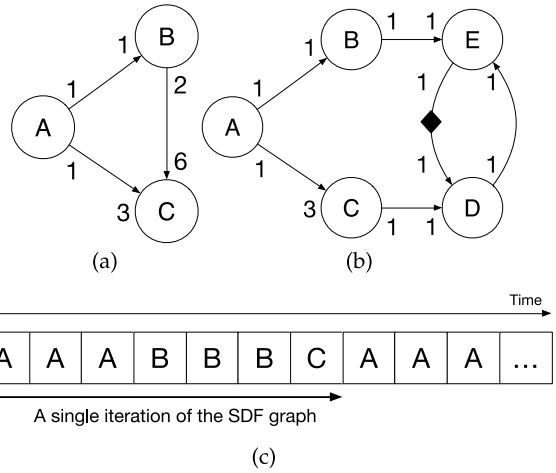


Fig. 1. (a) An SDFG, (b) another SDFG with an initial token, (c) a Fig. (a)' s scheduling: AAABBBC.

3 Problem Definition

3.1 Application Model

An application is represented as *a task graph representation*. Among various types of task graph representations, we choose an SDFG, $G(V, E)$; two simple examples are presented in Fig. 1. In an SDFG, each node $v \in V$ represents a computation task and each edge $e(v, v) \in E$ models a FIFO channel.

The number of tokens produced or consumed per task execution, which is called a sample rate, is annotated on each edge. An SDFG may have an initial token on an edge, which is depicted by a diamond shape (probably with the number of tokens) as shown on the edge (E, D) in Fig. 1b. The initial tokens indicate that there are tokens before the graph starts to execute. A task becomes executable if every edge accumulates no fewer number of tokens than the specified sample rate on every input edge. For example of Fig. 1b, the execution of task D consumes one token from the edge (C, D) and one token from the edge (E, D). Even though task D has enough tokens on edge (E, D) at the beginning of the iteration, task D is required to wait for one token on edge (C, D). If the condition is met, task D consumes two tokens, performs specified computation, and generates one token on the edge (D, E).

Determining the order of task executions on the processors is called a scheduling, and a scheduling of SDFG can be statically determined [36]. We use a static schedule based on the worst case execution time (WCET) [37], [38], [39] of the task to satisfy the real-time constraint [40]. Suppose that all tasks of Fig. 1a are mapped to a single processor and the given schedule is AAABBBC as shown in Fig. 1c. The schedule will be repeated with the streams of input tokens to the application.

An SDFG reveals task-level parallelism explicitly by specifying the true dependency only between tasks. Thus parallelizing an application is readily performed by mapping tasks to processing elements if it is specified in an SDFG.

The global state of the SDFG is constituted by the following two parts: 1) Internal states of each task, and 2) buffer states between tasks. An internal state of a task can be expressed by a feedback edge associated with the task. So we assume that an internal state of each task is saved in the form of a feedback buffer. Thus, the global state of the application is uniquely defined by the edge buffers of the tasks,

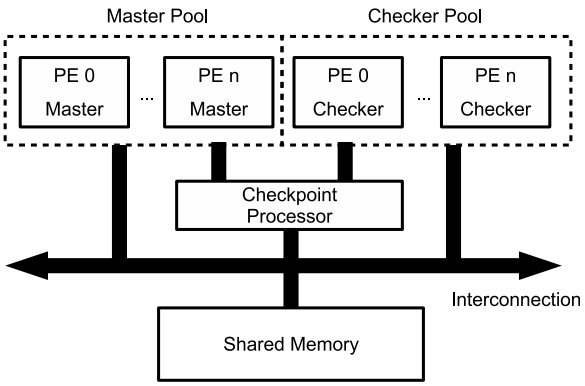


Fig. 2. A logical view of the system architecture assumed in this paper.

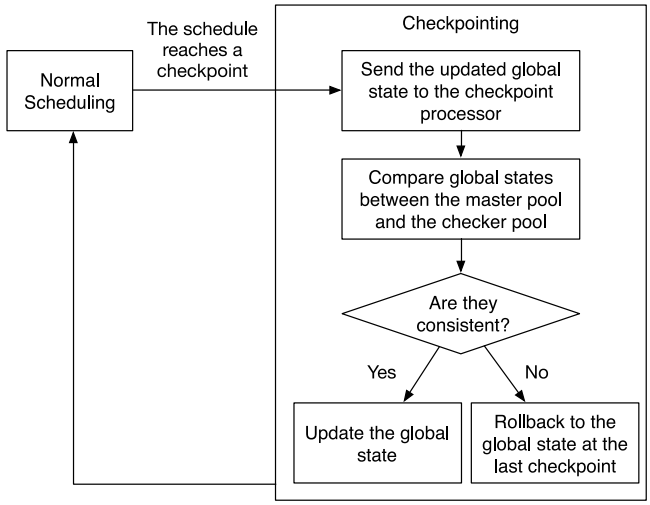


Fig. 3. Flowchart at a checkpoint.

and saving the current buffers is enough to capture the current global state of the application. If we make a checkpoint during a task execution, we need to save all temporary states such as stacks and registers for the task. In most previous works, it is also assumed that checkpointing can be performed at any time ignoring the volume of the states to be saved. On the other hand, we choose to set the checkpoints only at the task execution boundary. Then, the volume of states to be saved is determined at design-time and the overhead of checkpointing and recovery is accurately modeled to obtain optimal points.

3.2 Architecture

Implementation of a system which uses DMR and checkpointing can be done in various ways. We extend *the dual processor architecture* [41] to support a multiprocessor environment, namely *the dual multi-processor architecture*, as depicted in Fig. 2. The system consists of two distinct processor pools, while each pool contains a set of processing elements. Two processor pools are completely identical in terms of types of processing elements, topology, and other properties. Each processing element contains a processor and local (private) memory, either in a distributed memory system or part of a shared memory system. The system has a *reliable shared memory*, which is reachable from all processors, that is used to store the states of the application at checkpoints. Since the maximum size of checkpointing can be determined at compile time, we reserve the enough dedicated space for checkpointing. As long as memory space accommodates the checkpointing space, checkpointing does not affect the normal program execution in our model.

A special logical processor called a *checkpoint processor* or *monitor* is deployed to coordinate the execution of an application. The checkpoint processor detects failures by comparing the global states of the application at checkpoints, and coordinates the recovery according to the recovery scheme used. It is well known that DMR cannot detect common-mode failures (CMFs) if the same error occurs to both processors. Typical DMR compares the results of duplicated functions, while we compare all system states to be checkpointed. Since the possibility of CMF for all multiple bits of duplicated states is extremely low, CMF does not threaten usability of the proposed technique, we believe.

In such an architecture, two distinct processor pools are referred to as *the master pool* and *the checker pool* respectively, executing the same schedule being strictly synchronized at each checkpoint. The master pool has an access to the shared system memory and drives all system outputs, while the checker pool continuously executes the same task graph fetched by the master pool. The global states produced by both pools are provided to the checkpoint processor at the checkpoints. The detection of a disagreement on the value of any pair reveals the existence of a fault. The checkpoint processor is not capable of detecting interconnection and shared memory errors. Therefore checkpoint processor, bus, and shared memory must be protected against faults by using proper techniques such as the time-triggered architecture [42] or parity bits/error correcting codes (ECC) [8]. It is generally assumed that failures from microprocessor and local memory would be the dominant reason of a system failure due to relatively high soft error rate [2], [43].

3.3 Checkpoint Model

At each checkpoint, the checkpoint processor performs operations necessary to achieve fault detection and recovery, following the flowchart in Fig. 3. After receiving the updated global states from the master pool and the checker pool, the checkpoint processor compares them.

If no fault is detected or a detected fault is tolerable without any side-effect [44], [45], the global states at the checkpoint are updated to the stable storage, and both pools continue the next task execution. Otherwise the checkpoint processor rolls back to the lastly stored checkpoint, and sends the restored global states to both pools to resume task execution from the checkpoint.

Since the checkpointing is performed at the task boundary, the global state to be saved is uniquely defined by the buffer states of all edges. The shared memory in the target architecture should maintain the consistent global state. Note that we determine the live buffers at each task boundary from the static schedule. At a checkpoint, each processor may send only the difference between the current buffer states and the previous buffer states saved at the last checkpoint. If no data is produced or consumed in an edge buffer between two adjacent checkpoints, the data need not be stored at the checkpoint. This scheme is called *incremental checkpointing*. At the first checkpoint, we need to store all buffer states. In the example of Fig. 1a, since the example SDFG has no initial token on

all edge buffers and all buffers are empty, we do not have to store any buffer before starting execution.

Recovery policy is to fetch the entire states from the last checkpoint in the shared memory. The main advantage of deploying this policy is that the system can tolerate a transient fault event that simultaneously affects multiple memory cells or latches.

The checkpointing/recovery overhead would be a function of the data volume that should be transferred between the local memory and the reliable shared memory to generate/restore the global checkpoint image. For brevity of explanation, we assume that the overhead is linearly proportional to the data volume in Sections 4 and 5. For the experiment, we use a more elaborated realistic model.

There are two extreme checkpointing schemes. One is to save buffers at every node invocation. In this case, the accumulated checkpoint overhead is maximized and the rollback distance is minimized. The other extreme is to checkpoint only once during the graph execution. In this case, the accumulated checkpoint overhead is minimized, but the rollback distance is maximized. In the example of Fig. 1a, there is no checkpointing overhead if a single checkpointing is performed at the beginning of the iteration due to an empty buffer. However, the worst case rollback distance increases as the schedule length increases. The constraint may not be satisfied if the checkpointing overhead and recovery overhead are too large. In summary, we need to consider both checkpointing and recovery overhead to determine the optimal checkpoint.

The checkpointing overhead affects the throughput of the program at a normal condition. Hence, the checkpointing overhead is minimized with satisfying the given latency constraint while $maxF$ faults events are allowed during an iteration. The recovery overhead is considered when the locations of checkpoint is determined to satisfy the latency constraint. At the normal condition in the absence of transient faults, the reserved worst-case recovery overhead will become the slack time that can be utilized in various ways. For instance we may reduce the energy consumption by lowering the processor speed utilizing the slack time for the next iteration of graph execution. The detailed explanation how to exploit such slack time is out of scope of this paper.

3.4 Problem Description

Now we summarize the problem tackled in this paper as follows:

Target architecture.

- A dual multi-processor architecture that has an ECC-enabled shared memory, fault-free checkpoint processor, and fault-tolerant interconnection network.

Input information.

- An application specified by an SDFG.
- Static mapping and scheduling of the tasks to processors.

Fault model.

- At most $maxF$ transient fault events may occur during an iteration of a given SDFG. Each fault event may spoil multiple, possibly entire, processor states.

TABLE 1
Notations for Algorithm Description

Notation	Description
n	The number of possible checkpoint positions
E	Total execution time to execute a single iteration without faults
D	Deadline on the execution. $i.e, E \leq D$
t_i	Elapsed time from the beginning of the schedule to the checkpoint position c_i without faults.
$CO(i,j)$	Checkpointing overhead at c_j assuming that the previous checkpoint is made at c_i
$CD(i,j)$	Size of data transferred for checkpointing at c_j from c_i
$RBO(i,j)$	Rollback overhead from c_j to c_i.
$REO(i,j)$	Re-execution overhead from c_i to c_j, i.e., $t_j - t_i$
$RBD(i)$	Size of data to be transferred for recovering to c_i.
$RCO(i,j)$	Recovery overhead from c_j to c_i, including rollback overhead, re-execution overhead, and re-checkpointing overhead i.e., $RBO + REO + CO$
$MARCO(i,j)$	Maximum accumulated recovery overhead when the latest checkpoint was set at c_i before c_j, and there are $k-1$ faults.
$aco(i,j)$	Minimum accumulated checkpointing overhead when the latest checkpoint was set at c_i before $c_j, i.e, aco(i,j) = CO(i,j) + ACO(j)$
$ACO(j)$	Minimum accumulated checkpointing overhead at c_j
$CL(j)$	Checkpoint position list until c_j (possibly including j).

Constraints.

- The total execution time, including the checkpoint and recovery overheads, to execute a single iteration with faults must be equal to or less than the given deadline.

Problem.

- Decide the checkpoint positions to minimize the checkpointing overhead while satisfying the latency constraint.

4 Optimal Checkpoint Selection For Uniprocessor Implementation

In this section, we first explain the basic idea of the proposed algorithm based on recurrence relations with the simplest form of the problem. We solve the problem 1) for the uniprocessor case where each pool has a single processor on which the entire SDFG is executed. We further assume that 2) there is no initial state and 3) 1 fault in one iteration is allowed. We generalize the problem in three orthogonal ways in the following sections.

4.1 Basic Idea

Since we set the checkpoints at the task boundary, the finite possible checkpoint positions are fixed for a given schedule. Suppose that a given scheduling of Fig. 1a is $AAABBBC$. There are seven possible checkpoint positions as denoted by the binary values $\{c_i\}$ in the augmented schedule $c_0Ac_1Ac_2Ac_3Bc_4Bc_5Bc_6Cc_7$ (c_0 and c_7 refer to the same state before/after an iteration). If a binary value is 1, we define a checkpoint at that position. Then the problem is reduced to determine the values of $\{c_i\}$. In addition, we summarize variables and constants used in the formulation of the proposed technique in Table 1.

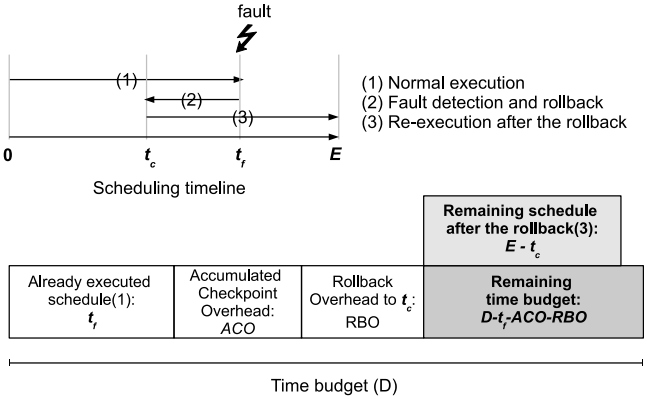


Fig. 4. Illustrative examples for Equation (1). The remaining time budget at any point ($D - t_f - ACO - RBO$) must be no less than the time required to finish the remaining schedule after the rollback operation ($E - t_c$).

To meet the latency constraint, the remaining time budget at any point must be no less than the time required to finish the remaining schedule after the rollback operation. Let us consider the timeline and time budget as shown in Fig. 4. Let t_c be the latest checkpoint time, t_f be the time instant when a fault is detected, where $t_c < t_f$, $ACO(f)$ be the accumulated checkpointing overhead until t_f, and $RBO(c, f)$ be the rollback overhead to the latest checkpoint at t_c.

Then, the following inequality should be satisfied to meet the latency constraint.

$$E - t_c \leq D - t_f - ACO(f) - RBO(c, f). \quad (1)$$

The left side of Equation (1) is the time to finish the graph after restarting the execution from the latest checkpoint while the right side is the remaining time to the deadline.

For example, suppose that $E = 10$, $D = 15$, $ACO(f) = 1$, $RBO(c, f) = 1$, and a fault is detected at $t_f = 8$. From the inequality, we obtain $5 \leq t_c$, meaning that the system would miss the deadline unless a checkpoint is performed after time 5.

$CO(i, j)$ depends on the size of data transferred for checkpointing at c_j from c_i, which is denoted by $CD(i, j)$. Note that we need to save the buffers updated during two checkpoints. If the next checkpoint is made at c_2 while the initial checkpoint is made at c_0, two edge buffers on channels AB and AC are updated. Hence $CD(0, 2)$ becomes the sum of two edge buffers on (A, B) and (A, C) at c_2. In the similar manner, we can define $RBO(i, j)$ as the rollback overhead from c_j to c_i. It also depends on the size of data to be transferred for recovering to c_i, which is denoted by $RBD(i)$. $RBD(i)$ will be the sum of the total edge buffers at c_i, regardless of c_j.

Finally, the objective function is to minimize accumulated checkpointing overhead $ACO(n)$ where n is the number of total task invocations in the schedule. For example, in Fig. 1a, n is 7. To minimize $ACO(n)$, we propose an optimal algorithm based on recurrence relations as presented in Algorithm 1.

Note that there is no state to be saved in the beginning since there is no initial token at any edge buffer. Therefore we can always assume that the first checkpoint is made before the graph starts execution; initially $ACO(0)$ is zero (line 3). To compute the accumulated checkpointing overhead at c_j, we consider each case separately that the latest checkpoint is made at c_i where i is less than j, and prune out infeasible candidate by using constraint $E - t_i \leq D - t_j - CO(i, j) - ACO(i) - RBO(i, j)$ based on Equation (1) (line 6). Then we compare the overhead to select the minimum value (line 8). When $argmin_i(aco(i, j))$ has more than single index, the minimum i, implying the earliest checkpoint, is picked for $CL(j)$ to make the number of checkpoints as small as possible while the checkpointing overhead is guaranteed (line 9). Through recurrence relations, we obtain the final result, $ACO(n)$, which is optimal (line 15). Since the time complexity to compute each $ACO(j)$ is proportional to j, the total time complexity to compute $ACO(n)$ is $O(n^2)$. We can find the positions of optimal checkpoints $CL(n)$ at $ACO(n)$.

Algorithm 1. An Optimal Checkpointing Algorithm for a Uniprocessor Schedule of an SDFG with No Initial Token.

Result: Minimum accumulated checkpointing overhead $ACO(n)$, and checkpoint position list $CL(n)$
variable $i, j \in \{0, 1 \ldots, n\}$
function $aco(i, j) \leftarrow CO(i, j) + ACO(i)$
1: **for** $j \leftarrow 0$ *to* n **do**
2: **if** $j = 0$ **then**
3: $ACO(0) \leftarrow 0$
4: $CL(0) \leftarrow \{0\}$
5: **else**
6: $I = \{i | E - t_i \leq D - t_j - CO(i, j) - ACO(i) - RBO(i, j)$ and $ACO(i)$ is not inf$\}$
7: **if** $I \neq$ *empty* **then**
8: $ACO(j) \leftarrow \min(aco(i, j))$ *s.t.* $i \in I$
9: $CL(j) \leftarrow CL(\min(argmin_i(aco(i, j)))) \cup \{j\}$ *s.t.* $i \in I$
10: **else**
11: $ACO(j) \leftarrow \inf, CL(j) \leftarrow$ *empty*
12: **if** $ACO(n) = \inf$ **then**
13: There is no feasible solution
14: **else**
15: return $ACO(n), CL(n)$

The proof of optimality of the given algorithms is straightforward. Since Algorithm 1, as well as the following algorithms, is based on recurrence relations, we can directly apply mathematical induction: "$ACO(0)$ is optimal" holds (the basis); if "$ACO(i)$ is optimal $s.t\ \forall i < j$" holds, then the statement "$ACO(j)$ is optimal" also holds[2] (the inductive step). The same logic can be applied to the subsequent algorithms since proposed algorithms share the same structure.

Fig. 5 illustrates how the proposed algorithm works for the example of Fig. 1c with the following assumptions for simple illustration; the execution times of all tasks are 10, the deadline is given as 120, all token have unit size, the

2. Suppose that there exists a checkpoint trace $CL(k) \cup \{j\}$ that minimizes the checkpoint overhead, where $ACO(j) = aco(k, j) < \min(aco(i, j))$ *s.t.* $i \in I$, and $k \notin I$. To determine set I, most components, $E, t_i, D, t_j, CO(i, j), RBO(i, j)$, are constant if the application model is determined. Although $ACO(i)$ is dependent on checkpoint traces to t_i, $ACO(i)$ represents the minimum checkpoint overhead by the induction. Therefore set I contains all checkpoint candidates that can satisfy the time constraint, regardless of the checkpoint traces. Since $k \notin I$, $CL(k) \cup \{j\}$ is an infeasible checkpoint list, which is a contradiction.

j	$ACO(j)$	$CL(j)$	
0	0	{0}	
1	2	{0,1}	min ($CO(0,1)$+$ACO(0)$) = min (2+0) = 2
2	4	{0,2}	min ($\underline{CO(0,2)+ACO(0)}$, $CO(1,2)+ACO(1)$) = min (4+0, 2+2) = 4
3	6	{0,3}	min ($\underline{CO(0,3)+ACO(0)}$, $CO(1,3)+ACO(1)$, $CO(2,3)+ACO(2)$) = min (6+0, 4+2, 2+4) = 6
4	7	{0,4}	min ($\underline{CO(0,4)+ACO(0)}$, $CO(1,4)+ACO(1)$, $CO(2,4)+ACO(2)$, $CO(3,4)+ACO(3)$) = min (7+0, 6+2, 4+4, 2+6) = 7
5	9	{0,4,5}	min ($CO(2,5)+ACO(2)$, $CO(3,5)+ACO(3)$, $\underline{CO(4,5)+ACO(4)}$) = min (6+4, 4+6, 2+7) = 9
6	11	{0,4,6}	min ($CO(4,6)+ACO(4)$, $CO(5,6)+ACO(5)$) = min (4+7, 2+9) = 11
7	9	{0,4,5,7}	min ($\underline{CO(5,7)+ACO(5)}$, $CO(6,7)+ACO(6)$) = min (0+9, 0+11)= 9

Fig. 5. Application of Algorithm 1 to our uniprocessor schedule of Fig. 1.

checkpointing overhead $CO(i,j)$ is equal to $CD(i,j)$, and the recovery overhead $RBO(i,j)$ is $2 \times RBD(i)$. Underlined terms form optimal checkpoints to fulfill the minimum overhead. Thus the set of optimal checkpoints is $\{c_0, c_4, c_5, c_7\}$ and the minimal checkpointing overhead is 9.

4.2 Multiple Fault Events

In the previous section, the accumulated recovery overhead is ignored because we do not need to consider previous transient faults and corresponding recovery overheads. In this section, we assume that maximum $maxF$ fault event may occur in one iteration of the schedule. Here, we introduce new term, the maximum accumulated recovery overhead, $MARCO(c,f)$ on the right side of the Equation (1). $MARCO(c,f)$ denotes the total time overhead consumed for a series of recoveries from the beginning to the current time t_f when the latest checkpoint was set at t_c before t_f. Then the remaining time to the deadline should be reduced by $MARCO(c,f)$. Incorporating $MARCO(c,f)$, Equation (1) is revised as $E - t_c \leq D - t_f - ACO(f) - RBO(c,f) - MARCO(c,f)$.

If there have been more than $maxF-1$ faults during the current iteration, no fault is allowed any more. Therefore we need to consider only $maxF-1$ previous recovery operations in the equation. Thus $MARCO(c,f)$ can be found when $maxF-1$ faults occur at the point incurring the maximum recovery overhead until t_f. Note that the recovery overhead $RCO(i,j)$ is composed of rollback overhead $RBO(i,j)$, re-execution overhead $REO(i,j)$ and re-checkpointing overhead $CO(i,j)$. The maximum recovery overhead is computed from the following theorem.

Theorem 1. $MARCO(c,f) = (maxF-1) \times \max RCO(i,j)$ *where* $i,j \in CL(f)$, *and* $\forall c_x = 0$ *where* $i < x < j$.

To perform a checkpoint at c_j, the previous checkpoint index i should satisfy the following formulation: $E - t_i \leq D - t_j - CO(i,j) - ACO(i) - RBO(i,j) - MARCO(i,j)$. It is apparent that $MARCO(i,j)$ is dependent on the previous checkpoint trace to c_i. In a single fault case, we need to maintain single optimum checkpoint trace at each point. Due to the presence of $MARCO(i,j)$ in the formulation for multiple faults, the checkpoint trace to minimize only $ACO(i)$ would not lead to feasible solutions. Therefore, we maintain all checkpoint traces to minimize a pair of $ACO(i)$ and $MARCO(i,j)$ to guarantee the optimal solution.

Given the checkpoint position list, the worst case complexity of finding the maximum recovery overhead is $O(n)$ where n is the total number of task invocations in a schedule. Note that the average complexity is much less in the ordinary case unless the distance between consecutive checkpoints is extremely tight.

4.3 An SDFG with Initial Tokens

If an SDFG has initial tokens, the checkpointing overhead is not zero any more at the beginning of the graph. Hence, we cannot guarantee that the optimal set of checkpoints includes c_0 as in Algorithm 1. Instead, we find the minimum checkpointing overhead by considering each of possible checkpoints as a candidate starting point in Algorithm 1 in an iterative fashion.

Algorithm 2. An Optimal Checkpointing Algorithm for a Uniprocessor Schedule of a General SDFG with Initial Tokens and/or Internal States.

Result: Minimum accumulated checkpointing overhead $minACO$, and Checkpoint position list $minCL$
variable $i,j,q \in \{0,1\ldots,n\}$
function $aco_q(i,j) = CO(i,j) + ACO_q(i)$
//First Phase
for $q \leftarrow 0$ *to* $n-1$ **do**
  for $j \leftarrow q$ *to* n **do**
    if $j = q$ **then**
      $ACO_q(j) \leftarrow 0$
      $CL_q(j) \leftarrow \{j\}$
    else
      $I = \{i \mid E - t_i \leq D - t_j - CO(i,j) - ACO_q(i) - RBO(i,j)\ s.t.\ q \leq i$ and $ACO_q(i)$ is not inf$\}$
      if $I \neq empty$ **then**
        $ACO_q(j) \leftarrow \min(aco_q(i,j))\ s.t.\ i \in I$
        $CL_q(j) \leftarrow CL_q(\min(argmin_i(aco_q(i,j)))) \cup \{j\}\ s.t.\ i \in I$
      else
        $ACO_q(j) \leftarrow \inf, CL_q(j) \leftarrow empty$
//Second Phase
for $q = 0$ *to* $n-1$ **do**
  $I = \{i \mid E - (t_i - t_q) \leq D - E - CO(i,n+q) - ACO_q(i) - RBO(i,n+q)\ s.t.\ q \leq i$ and $ACO_q(i)$ is not inf$\}$
  if $I \neq empty$ **then**
    $ACO_q(q) \leftarrow \min(aco_q(i,n+q))\ s.t.\ i \in I$
    $CL_q(q) \leftarrow CL_q(\min(argmin_i(aco_q(i,n+q))))\ s.t.\ i \in I$
  else
    $ACO_q(q) \leftarrow \inf, CL_q(q) \leftarrow empty$
//Final Phase
$minACO \leftarrow min_q(ACO_q(q))\ s.t.\ 0 \leq q \leq n-1$
$minCL \leftarrow CL_q(\min(argmin_q(ACO_q(q))))\ s.t.\ 0 \leq q \leq n-1$
if $MinACO = \inf$ **then**
  There is no feasible solution
else
  return $minACO, minCL$

q \ j	0	1	2	3	4	5	6
0	6	4	4	4	5	5	6
1		9	2	3	4	4	5
2			inf	1	2	2	3
3				inf	1	1	2
4					inf	1	1
5						inf	1

Fig. 6. $ACO_q(j)$ computation. The gray-shaded cells are computed at the first phase, and the white cells are computed at the second phase. The minimum value among values of white cells is selected as the minimum checkpoint overhead at the final phase. Note that the black cells are invalid, and there is no feasible solution if $q \geq 2$.

We replace $ACO(j)$ with $ACO_q(j)$ which indicates the minimum checkpointing overhead accumulated *after* the initial checkpoint c_q until c_j. It also implies that there is no checkpoint from c_0 to c_{q-1}, i.e., $c_0 = c_1 = \cdots = c_{q-1} = 0$, since c_q is assumed to be the first checkpoint position. $aco_q(i,j)$ and $CL_q(j)$ are also redefined in the same way. If a fault occurs between c_0 and c_{q-1}, we need to roll back to the last checkpoint of the previous iteration. Algorithm 2 is extended from Algorithm 1 to consider the initial tokens.

Algorithm 2 consists of three phases: the first phase (lines 1-12) computes the checkpointing overhead during $(c_q, c_n]$, which is similar to Algorithm 1 except that the starting checkpoint is not c_0, but c_q (line 2). The second phase (lines 13-19) computes the checkpointing overhead at c_{n+q} for each q, from the previous iteration. The last phase (lines 20-25) chooses the minimum value of the checkpointing overhead among all values of $ACO_q(q)$. The time complexity of Algorithm 2 becomes $O(n^3)$ with maximum 1 fault in one iteration because Algorithm 2 basically repeats Algorithm 1 n times.

For the illustration of Algorithm 2, consider the SDFG of Fig. 1b with a given schedule, $ABCCDE$. Then the augmented schedule by candidate checkpoints becomes $c_0Ac_1Bc_2Cc_3Cc_4Dc_5Ec_6$. We make the same assumption on the task execution times, the token size, the checkpointing and the recovery overheads as the case of Fig. 5, and we assume that the deadline D is 100.

Fig. 6 shows the entire computation results from which $ACO_0(0)$ is picked as the minimum overhead solutions. $ACO_0(0)$ indicates that the optimal checkpoints become $\{c_0, c_3, c_4\}$ or the optimal schedule becomes (checkpoint) ACB (checkpoint) C (checkpoint) DE, and the overhead is 6.

5 Optimal Checkpoint Selection For Multiprocessor Implementation

The key challenge for multiprocessor implementation is to guarantee the consistency of the checkpoints. In a uniprocessor case, we can safely go to the last checkpoint when a fault is detected. It is not true anymore in a multiprocessor system. If a fault is detected, all processors go to the appropriate checkpoints that define *the system-wide consistent state*.

Consider a partial schedule graph in Fig. 7 on two processors. A scheduling graph represents 1) the partitioning and the order of task execution, and 2) the dependency between task instances on a multiprocessor schedule.

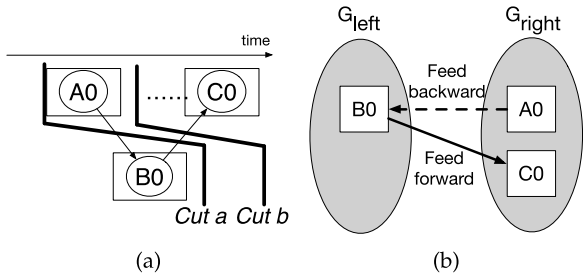


Fig. 7. Valid checkpoint cut: (a) A partial scheduling graph with valid/invalid checkpoint cuts, and (b) two subgraphs divided by *Cut a*.

Cut a and *Cut b* indicate where checkpoints are made. Suppose that checkpoints are made before the invocation $A0$ on a processor, and after invocation $B0$ on another processor as *Cut a*. When an error is detected during execution of task instance $C0$, the both processors will roll back to *Cut a* if they both roll back to the last checkpoints. Unfortunately, *Cut a* does not maintain a consistent system state; if we restart from *Cut a*, the output token produced by invocation $A0$ will not be consumed by $B0$ in the current iteration since $B0$ will not be re-executed. It may cause a buffer overflow or incorrect result in the next iteration.

Therefore we have to define a *checkpoint cut* that is a set of synchronized checkpoints on all processors. When we make a rollback, we resume from a checkpoint cut. In the previous example, *Cut a* is not a valid checkpoint cut.

5.1 Valid Checkpoint Cuts

Based on the scheduling result, we generate valid checkpoint cuts that represent checkpoint position candidates. In the example of Fig. 7, *Cut b* defines a valid checkpoint cut while *Cut a* is not. In order to build a valid checkpoint cut on multiprocessors, we should ensure that the cut does not yield any dangling data after restart. We define a *feed-forward cut* as follows.

Definition 1 (Feed-forward Cut). *A checkpoint cut divides a scheduling graph into two subgraphs. Let two bipartite graphs be G_{left} and G_{right} where tasks in G_{left} need not be re-executed, and tasks in G_{right} should be re-executed after rollback. If there exists no edge from G_{right} to G_{left}, then the cut is defined as a feed-forward cut; otherwise it is a feedback cut.*

For instance in Fig. 7, *Cut a* partitions the graph into $G_{left} = \{B0\}$ and $G_{right} = \{A0, C0\}$. Since there is an edge from G_{right} to G_{left}, edge $(A0, B0)$, it is not a feed-forward cut. On the other hand, *Cut b* is a feed-forward cut since there is no edge to $G_{left} = \{A0, B0\}$ from $G_{right} = \{C0\}$. A feed-forward cut that does not generate any dangling data after rollback is a valid cut.

5.2 *Checkpoint Cut Lattice*

In order to consider all possible checkpoint cuts, we create a new data structure called a checkpoint cut lattice, abbreviated as *CCL*. Given an SDFG in Fig. 1a and its associated multiprocessor schedule in Fig. 8a, the example of *CCL* is shown in Fig. 8b. Note that each vertex in a *CCL* is one-to-one mapped to a valid checkpoint cut. A vertex is represented as a set of tasks included in the left subgraph G_{left} stated in Definition 1. For example, vertex $\{A0, A1\}$ means that *Cut c2* is chosen as a checkpoint cut,

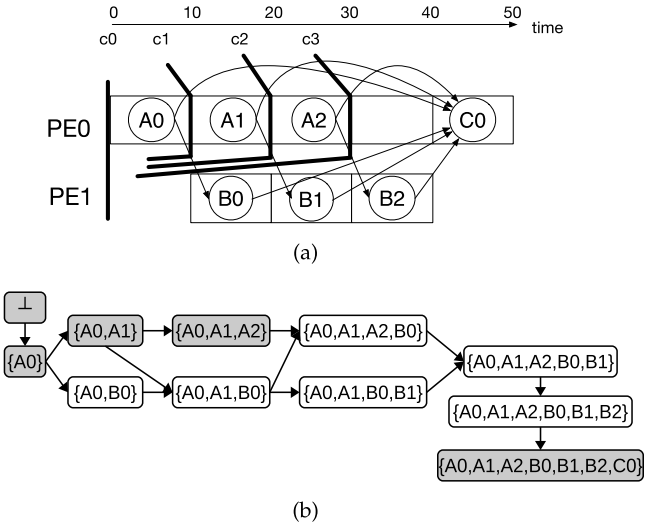


Fig. 8. *Checkpoint cut lattice (CCL)*: (a) a given scheduling graph for Fig. 1 a with a optimal checkpoint cut (thick solid lines), and (b) the generated CCL.

after $A1$ on PE0 and before $B0$ on PE1. An edge between two vertices represents the inclusion relation between G_{left}. Relation $A \prec B$ denotes that vertex B has all elements of vertex A. This relation holds only if there is a path from vertex A to vertex B in the lattice. The *CCL* is constructed from initial vertex $\perp$ that means a checkpoint cut at the beginning of the iteration.

An algorithm to build *CCL* is described in Algorithm 3. Tracing through the schedule, we find all feed-forward cuts and create a new vertex for each feed-forward cut.

Algorithm 3. Checkpoint Cut Lattice Construction Algorithm.

Data: Scheduling graph SG
Result: Checkpoint cut lattice CCL
struct cut $\{children, tasklisk\}$
//children contains a list of children cuts in CCL, and tasklist contains a list of scheduled tasks at the cut
function $|cut_i|$ is the number of elements in $cut_i.tasklist$
1: $S \leftarrow \{\}; new\_S \leftarrow \{\}$
2: $cut_0 \leftarrow (children = \{\}, tasklist = \{\})$
//create initial vertex $\perp$
3: $CCL \leftarrow \{cut_0\}$
4: $SRCINST \leftarrow \{t_j \mid t_j$ is an initial instance of source tasks and t_j has no earlier invocation on the target processor$\}$
5: **for** $i \leftarrow 1$ *to* $|SG|$ **do**
6: **if** $i = 1$ **then**
7: **for** *all* $t_j \in SRCINST$ **do**
8: $new\_S \leftarrow \{t_j\}$
9: generate and add $cut_{new}(children = \{\}, tasklist = new\_S)$ to CCL
10: $cut_0.children.add(cut_{new})$
//set $\perp$ as the parent nodes
11: **else**
12: **for** all cut_j in CCL s.t. $|cut_j| = i - 1$ **do**
13: $S \leftarrow cut_j.tasklist$
14: **for** all $t_j \in S$ **do**
15: **for** all t_k *s.t.* t_k is later than t_j on the same processor or t_k has incoming edge from t_j or $t_k \in SRCINST$ **do**
16: **if** all incoming edges of t_k are directed from the actor invocation $\in S$ and not $t_k \in S$ **then**
17: $new\_S \leftarrow S \cup \{t_k\}$
18: find $cut_{new}(tasklist = new\_S, \ldots)$ from CCL
19: **if** cut_{new} does not exist **then**
20: generate and add $cut_{new}(children = \{\}, tasklist = new\_S)$ to CCL
21: $cut_j.children.add(cut_{new})$

After creating initial vertex $\perp$ (line 2-3), each first-tier vertex ($|cut| = 1$) is created from a single task instance that satisfies two conditions: 1) The task instance is the initial instance of a source task and 2) the first schedule on each processor (lines 6-10). And then we generate vertexes with multiple task instances by using an inclusion relation with existent vertexes (lines 11-21). Since the order of space to store the *CCL* is $O(2^{|SG|})$, the worst case complexity of *CCL* construction or $O(|SG|^3|CCL|^2)$ becomes $O(|SG|^3 4^{|SG|})$, where $|SG|$ is the total number of task instances in a schedule, and $|CCL|$ is the number of checkpoint cut in the *CCL*. While the worst-case complexity is exponential to $|SG|$, it occurs if there is no dependency between task invocations in an SDFG and all task invocations are mapped on the different processors.

Unless there is a path from checkpoint cut_x to cut_y, both cut_x and cut_y cannot be taken together. For example, cuts $\{A0, B0\}$ and $\{A0, A1, A2\}$ do not have a path between them. Cut $\{A0, B0\}$ saves the state after the first invocation of task B while cut $\{A0, A1, A2\}$ saves the state before task B is invoked. It is a contradiction.

Algorithm 4. An Optimal Checkpointing Algorithm for Multiprocessor Schedules without an Initial Sample.

Result: Minimum accumulated checkpointing overhead $ACO(cut_{sink})$, and Checkpoint position list $CL(cut_{sink})$
variable $cut_i, cut_j \in CCL$
function $|cut_i|$ is the number of elements in $cut_i.tasklist$
variable
cut_{sink} is the sink checkpoint cut $s.t.\ |cut_{sink}.children| = 0$
function $aco(cut_i, cut_j) = CO(cut_i, cut_j) + ACO(cut_i)$
function $finishtime_p(cut_i)$
is the last finish time on processor p among $cut_i.tasklist$
function $cp(cut_i)$ is the execution time of the critical path from cut_i to the end of the schedule
1: **for** $cut_j \in CCL$ in the increasing order by size of $|cut_j|$ **do**
2: **if** $|cut_j| = 0$ **then**
3: $ACO(cut_j) \leftarrow 0$
4: $CL(cut_j) \leftarrow \{cut_j\}$
5: **else**
6: $I = \{cut_i \mid cp(cut_i) \leq min_p(D - finishtime_p(cut_j)) - CO(cut_i, cut_j) - ACO(cut_i) - RBO(cut_i, cut_j)\ s.t.\ cut_i \prec cut_j$ and $ACO(cut_i)$ is not inf$\}$
7: **if** $I \neq empty$ **then**
8: $ACO(cut_j) \leftarrow \min(aco(cut_i, cut_j))\ s.t.\ cut_i \in I$
9: $MinCutList \leftarrow argmin_{cut_i}(aco(cut_i, cut_j))\ s.t.\ cut_i \in I$
10: $cut_i \leftarrow argmin_{cut_x}|cut_x|\ s.t.\ cut_x \in MinCutList$
11: $CL(cut_j) \leftarrow CL(cut_i) \cup \{cut_j\}$
12: **else**
13: $ACO(cut_j) \leftarrow$ inf, $CL(cut_j) \leftarrow$empty
14: **if** $ACO(cut_{sink}) =$ inf **then**
15: There is no feasible solution
16: **else**
17: return $ACO(cut_{sink}), CL(cut_{sink})$

After constructing the *CCL*, an optimal checkpoint algorithm for multiprocessors is devised similarly to Algorithms 1 and 2 for the uniprocessor case. The key difference between Algorithms 1 and 4 is that Algorithm 1 uses an integer index while Algorithm 4 utilizes a CCL vertex to indicate the checkpoint position. Algorithm 4 is given for multiprocessor schedules without initial sample. If there is no initial token on all edges, a checkpoint cut is built at the beginning with no checkpointing overhead (line 2-4). Note that the algorithm based on recurrence relations proceeds following all paths in the *CCL*. Similarly to the uniprocessor case, $CO(x, y)$ is defined as the state update cost of cut_y after cut_x. Other terms are similarly defined based on the vertex index instead of schedule position index. In case there are initial tokens on any edge, we have to consider all possible positions of the first checkpoint as Algorithm 2 does.

The latency constraint formulation to check the checkpoints becomes more complicated on the multiprocessor schedule than the uniprocessor schedule (line 6). The left side of the inequality should be the maximum execution time from the checkpoint cut to the end of the iteration and the right side is the minimum remaining time from the current cut to the deadline. At line 6, $cp(cut_i)$ indicates the remaining execution time from cut_i to the end of the execution, and $min_p(D - finishtime_p(cut_j))$ denotes the remaining time from cut_j to the deadline. For example, suppose that each invocation of all tasks requires the same amount of time in the scheduling graph as shown in Fig. 8a. Checkpoint cut $c3$ corresponds to valid lattice vertex $\{A0, A1, A2\}$. For the checkpoint cut, the remaining time to deadline (right-side term) is computed from $A2$ while the remaining time to be executed for the schedule (left-side term) is computed from $B0$.

The vertex which is the farthest one from vertex cut_j among $MinCutList$ (lines 9-10) is used to minimize the number of checkpoints under the same minimum checkpointing overhead (line 11).

With the same assumption on the task execution times, the token sizes, and the checkpointing/recovery overhead as the case of Fig. 5, the optimal checkpoint overhead of the example of Fig. 8 with deadline $D = 90$ is 6. The optimal solution is obtained with four checkpoint cuts: $\{\}$, $\{A0\}$, $\{A0, A1\}$ and $\{A0, A1, A2\}$ as shown in Fig. 8a.

6 Experiment

The proposed algorithm guarantees the optimal solutions if a checkpoint is allowed only at the task boundary. To our best knowledge, there is no previous work to optimize the same problem. For the purpose of comparison, we devise two baseline design-time algorithms: *Greedy* and *Equi-distance* algorithms.

Greedy algorithm determines a checkpoint cut whenever the time constraint inequalities are not satisfied. First, checkpointing is performed at the beginning of the iteration. Tasks are executed without checkpointing unless the time constraint is violated. Once the time constraint is violated after the task execution, the execution is rolled back to satisfy the time constraint by introducing a new checkpoint. Then we resume the schedule from the latest checkpoint. Note that although it is allowed to perform a checkpoint at the task boundaries, *Greedy* may conduct the checkpoint during the task execution on a multiprocessor system when a task finishes its execution on a processor and tasks are still working on the other processors.

On the other hand, checkpoints of *Equi-distance* algorithm are equally distributed throughout the execution time of the application. The algorithm to find the optimal checkpoint interval is widely studied in many literatures [24], [26], [46]. In this experiment, *Equi-distance* algorithm performs an exhaustive search to obtain the optimal distance between consecutive checkpoints.

For checkpointing overhead CO and rollback overhead RBO, more elaborated memory read/write model is introduced. A memory write operation takes (transferred data size (byte) / 4) cycles for an infinite size write buffer, while a memory read operation requires 250 cycles for the first memory chunk, and 20 cycles per subsequent memory chunk. The size of memory chunk is set to 8 bytes.

The first set of tested SDFGs consists of 20 synthetic SDFGs generated by SDF3 [47] (*S0-0 - S1-9*). The number of tasks in an SDFG and the number of task instances per iteration are 5-10 and 10-30, respectively. The second set of graphs is composed of five practical applications, including *h263 encoder/decoder*, *mp3 decoder*, *modem*, and *FFT* applications. The first four SDFGs are from SDF3, and the last SDF is from StreamIt benchmark [48]. Benchmarks from SDF3, except *Modem*, have the information of WCET and data/state size, which are extracted from the real implementation on *ARM7TDMI* core, and synthetic values are utilized for *Modem*. For *FFT* application, we use profiling data from *CELL BE* Processor.

To perform a checkpoint during the task execution, it is required to save the intermediate image of the all running tasks. For synthetic examples, we assume that the intermediate image of the task consists of the process meta-information and the current stack. The process control block (PCB) size of MicroC/OS-III, one of commonly used as real-time lightweight OS, is 136 bytes. We assume that the maximum stack size would be bounded by the data consumed and produced per task execution. In the experiment the volume of the stack is the sum of input and output rates of the task, multiplied by the token size.

For each benchmark, we use Algorithm 2 for a uniprocessor system and Algorithm 4 for a multiprocessor system with four processors maximally. Note that we assume $maxF = 1$ if $maxF$ is not explicitly stated.

We have implemented the algorithms in Java 1.7. All experiments have been conducted on a desktop machine with Intel Core i7-4770 processor clocked at 3.4 GHz, 16 GB main memory, and running Ubuntu Linux 64-bit.

6.1 The Minimum Feasible Deadline

In this section, we will show that the proposed approach produces feasible solutions with a tighter deadline while the based algorithms cannot.

Figs. 9a and 9b show the minimum deadline with which each algorithm can find a feasible solution. The proposed algorithm generates the feasible solution for tighter deadlines by 9.42 percent (uniprocessor) and 5.11 percent (multiprocessor) on average, than the *greedy* and the *equi-distance* algorithms. On the other hand, *MP3 decoder* on the multiprocessor

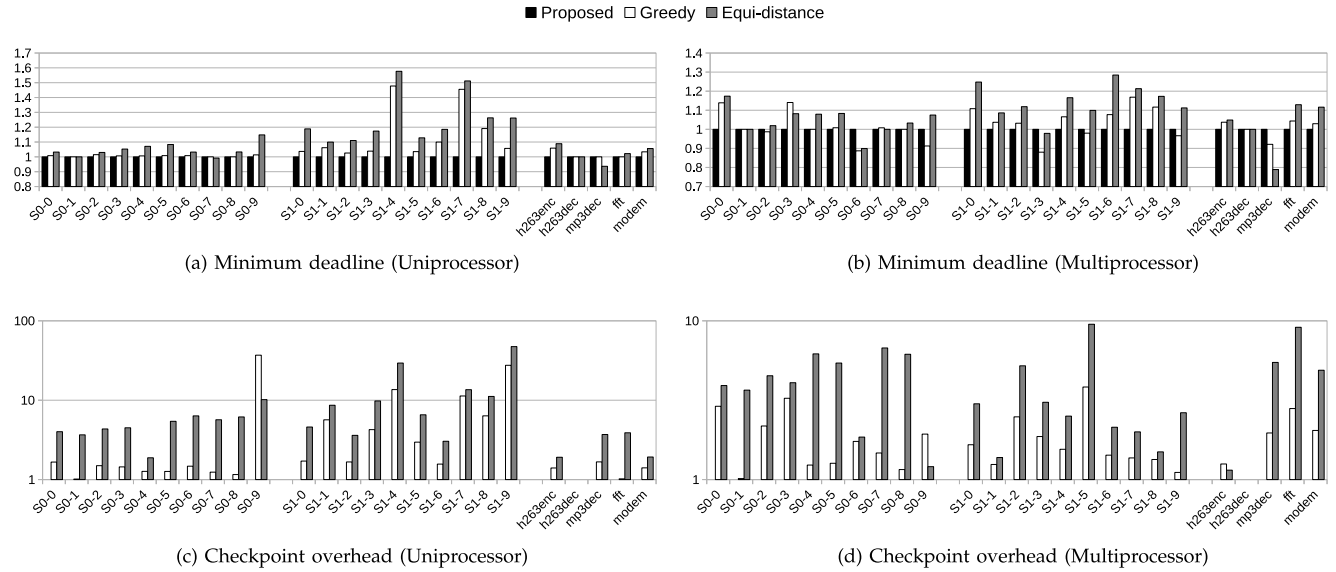


Fig. 9. For uniprocessor and multiprocessor schedules, we measure that 1) the minimum deadline in which each technique finds the feasible checkpoint placement, and 2) the checkpoint overhead under the minimum deadline that all techniques find a feasible solution. Note that all values are normalized by the value from proposed technique.

system, the proposed algorithm is defeated by the baseline algorithms since the execution time of a task becomes too large. While the end-to-end latency of *MP3 decoder* is 8,318,404 cycles, WCETs of both tasks "p10" and "p11" are 1,866,138 cycles which occupy about 45 percent of the end-to-end latency of the application. In this case, checkpointing during the execution of tasks with high WCET provides a better solution than the boundary based checkpoint algorithm.

6.2 The Checkpoint Overhead

We compare the checkpoint overhead when every algorithm can find a feasible solution in Figs. 9c and 9d. The experiment results show that the proposed algorithm reduces the checkpoint overhead by 6.68 times (uniprocessor) and 2.86 times (multiprocessor) on average compared with the baseline algorithms.

We observe that the proposed algorithm does not outperform the baseline algorithms for *H263 decoder* benchmark. Since the data size is too large for *H263 decoder*, the checkpoint overhead becomes dominant compared with the task execution time. Therefore all algorithms generate the same results for the application with a long deadline resulting in a single checkpoint per iteration.

From the static schedule, we can compute the maximum space requirement for checkpointing data. By reusing the buffer spaces if their life times are not overlapped, we could reduce the checkpointing space. For *H263 encoder*, *H263 decoder*, and *MP3 decoder* on a uniprocessor system, the sizes of dedicated checkpointing spaces are 745, 313, and 32 KB, respectively.

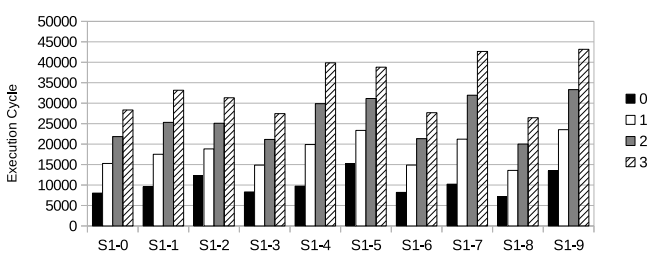


Fig. 10. The minimum end-to-end latency of fault-tolerant schedule with various $maxF$ for s1-0 to s1-9.

6.3 Effect of the Number of Tolerable Faults per Iteration

As the number of tolerable faults per the iteration increases, more checkpoints and more relaxed deadline are required. Deadlines are prolonged by 1.62 times to tolerate a single fault, 2.27 times for two faults, and 2.91 times for three faults on average. The minimum feasible deadlines for *s1-0* to *s1-9* for a uniprocessor system are shown in Fig. 10.

6.4 Solving Time

The proposed optimal algorithm is quite fast although it requires more execution time than the baseline algorithms. The solving times for most benchmarks, except for *H263 decoder*, are less than one second. For the biggest benchmark of *H263 decoder*, it takes about 300 and 1,385 seconds to find the solution on uni- and multiprocessor targets, respectively. Since the CCL is reusable for other deadlines once CCL is constructed, the execution time is not quite large for the complex application on the multiprocessor target. Even when it requires a long solving time the execution time is affordable since the computation is performed at compile time, We believe that the proposed optimal algorithm is practically applicable for large real-life examples.

7 Conclusion

In this paper, we utilize DMR and checkpointing to tolerate transient faults that may corrupt the result for a given application specified in a task graph that can be statically scheduled, and presents an optimal checkpoint selection problem. We devise an optimal algorithm based on recurrence relations for uniprocessor implementation. Also, we extend the algorithm to multiprocessor implementation by

using a data structure called checkpoint cut lattice. Experimental results show that the proposed algorithm effectively reduces the minimum end-to-end latency to perform a fault-tolerant schedule. In addition, the proposed algorithm dramatically decreases the checkpointing overhead on uniprocessor and multiprocessor systems compared with a greedy approach and an equidistant algorithm.

Acknowledgments

This work was supported by the Ministry of Science, ICT & Future Planning (MSIP), Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA) (NIPA-2014-H0301-14-1018), by Basic Science Research Programs through the National Research Foundation of Korea (NRF) funded by the MSIP (2013R1A1A1012715, 2013R1A1A1013384, 2013R1A2A2A01067907), by Center for Advanced Image of Chonbuk National University, and by IT R&D program MKE/KEIT (No. 10041608, Embedded system Software for New-memory based Smart Device). The ICT at Seoul National University provides research facilities for this study. The corresponding author is Hyunok Oh.

References

[1] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, Jul./Aug. 2003.

[2] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *Proc. 50th Annu. Des. Autom. Conf.*, May 2013, pp. 1–10.

[3] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 7, pp. 956–965, Jul. 2009.

[4] S. Punnekkat and A. Burns, "Analysis of checkpointing for schedulability of real-time systems," in *Proc. 4th Int. Workshop Real-Time Comput. Syst. Appl.*, Oct. 1997, pp. 198–205.

[5] N. Kandasamy, J. Hayes, and B. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 113–125, Feb. 2003.

[6] R. Lyons and W. Vanderkulk,"The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Develop.*, vol. 6, no. 2, pp. 200–209, 1962.

[7] S. Mitra and E. McCluskey, "Word-voter: A new voter design for triple modular redundant systems," in *Proc. IEEE 18th Very Large Scale Integr. Test Symp.*, 2000, pp. 465–470.

[8] C. Chen and M. Hsiao,"Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124–134, 1984.

[9] D. Rossi, N. Timoncini, M. Spica, and C. Metra, "Error correcting code analysis for cache memory high reliability and performance," in *Proc. Des., Autom. Test Eur. Conf. Exhib.*, 2011, pp. 1–6.

[10] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3-GHz fifth-generation SPARC64 microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 1896–1905, Nov. 2003.

[11] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.

[12] C. Metra, D. Rossi, M. Omana, A. Jas, and R. Galivanche, "Function-inherent code checking: A new low cost on-line testing approach for high performance microprocessor control logic," in *Proc. 13th Eur. Test Symp.*, May 2008, pp. 171–176.

[13] D. Pradhan and N. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Trans. Comput.*, vol. 43, no. 10, pp. 1163–1174, Oct. 1994.

[14] A. Ziv and J. Bruck, "Performance optimization of checkpointing schemes with task duplication," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1381–1386, Dec. 1997.

[15] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[16] H. Oh and S. Ha. (2004). Fractional rate dataflow model for efficient code synthesis. *J. VLSI Signal Process. Syst. Signal, Image Video Technol.* [Online], *37(1)*, pp. 41–51. Available: http://dx.doi.org/10.1023/B

[17] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static data flow," in *Proc. Int. Conf. Acoust., Speech, Signal Proces.*, May 1995, vol. 5, pp. 3255–3258.

[18] H. Hwang, T. Oh, H. Jung, and S. Ha, "Conversion of reference C code to dataflow model H.264 encoder case study," in *Proc. Asia South Pac. Des. Autom. Conf.*, Jan. 2006, pp. 24–27.

[19] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[20] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. IEEE 18th Int. Symp. Defect Fault Tolerance Very Large Scale Integr. Syst.*, Nov. 2003, pp. 581–588.

[21] B. Dave and N. Jha, "COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance," *IEEE Trans. Comput.*, vol. 48, no. 4, pp. 417–441, Apr. 1999.

[22] C. Bolchini and A. Miele, "Reliability-driven system-level synthesis for mixed-critical embedded systems," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2489–2502, Dec. 2013.

[23] C.-C. Han, K. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Trans. Comput.*, vol. 52, no. 3, pp. 362–372, Mar. 2003.

[24] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 3, pp. 389–402, Mar. 2009.

[25] Z. Zhang, D.-cheng Zuo, Y. wei Ci, and X.-zong Yang, "The checkpoint interval optimization of kernel-level rollback recovery based on the embedded mobile computing system," in *Proc. IEEE 8th Int. Conf. Comput. Inf. Technol. Workshops*, 2008, pp. 521–526.

[26] N. Chen and S. Ren, "Adaptive optimal checkpoint interval and its impact on system's overall quality in soft real-time applications," in *Proc. ACM Symp. Appl. Comput.*, Mar. 2009, pp. 1015–1020.

[27] S. Feng, S. Gupta, A. Ansari, and S. A. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proc. 15th Archit. Support Program. Lang. Oper. Syst.*, 2010, pp. 385–396.

[28] D. Nikolov, U. Ingelsson, V. Singh, and E. Larsson, "On-line techniques to adjust and optimize checkpointing frequency," in *Proc. IEEE Int. Workshop Rel. Aware Syst. Des. Test*, Bangalore, India, Jan. 7-8, 2010, pp. 29–33.

[29] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling of fault-tolerant embedded systems with soft and hard timing constraints," in *Proc. Des., Autom. Test Eur.*, 2008, pp. 915–920.

[30] D. Cummings and L. Alkalaj, "Checkpoint/rollback in a distributed system using coarse-grained dataflow," in *Proc. 24th Int. Symp. Fault-Tolerant Comput.*, Jun. 1994, pp. 424–433.

[31] W. Farquhar and P. Evripidou, "Fault detection and recovery in a data-driven real-time multiprocessor," in *Proc. 8th Int. Parallel Process. Symp.*, Apr. 1994, pp. 769–774.

[32] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

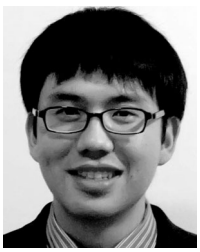
[33] D. Rai, L. Schor, N. Stoimenov, and L. Thiele, "Distributed stable states for process networks: Algorithm, analysis, and experiments on intel SCC," in *Proc. 50th Annu. Des. Autom. Conf.*, May 2013, pp. 1–10.

[34] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Softw. Eng.*, vol. SE-1, no. 2, pp. 220–232, Jun. 1975.

[35] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *Proc. IEEE Int. Rel. Physics Symp.*, 2011, pp. 5B.4.1–5B.4.7.

[36] S. Bhattacharyya, P. Murthy, and E. Lee,"Synthesis of embedded software from synchronous dataflow specifications," *J. Very Large Scale Integr. Signal Process.*, vol. 21, no. 2, pp. 151–166, 1999.

[37] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem–Overview of methods and survey of tools," *ACM Trans. Embedded Comput.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.
[38] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for WCET analysis," *Real-Time Syst.*, vol. 34, no. 3, pp. 195–227, 2006.
[39] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proc. IEEE 28th Int. Real-Time Syst. Symp.*, Dec. 2007, pp. 239–243.
[40] P. Puschner and A. Burns,"Guest editorial: A review of worst-case execution-time analysis," *Real-Time Syst.*, vol. 18, no. 2, pp. 115–128, 2000.
[41] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2003, pp. 170–177.
[42] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.
[43] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Mateo, CA, USA: Morgan Kaufmann, 2010.
[44] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *Proc. Conf. Des., Autom. Test Eur. Conf. Exhib.*, 2013, pp. 202–207.
[45] M. Shafique, B. Zatt, S. Rehman, F. Kriebel, and J. Henkel, "Power-efficient error-resiliency for H.264/AVC context-adaptive variable length coding," in *Proc. Des., Autom. Test Eur. Conf. Exhib.*, 2012, pp. 697–702.
[46] A. Bertossi and L. Mancini, "Scheduling algorithms for fault-tolerance in hard-real-time systems," *Real-Time Syst.*, vol. 7, no. 3, pp. 229–245, Nov. 1994.
[47] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," in *Proc. Appl. Concurrency Syst. Des.*, 2006, pp. 276–278.
[48] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *Proc. Parallel Archit. Compilation Techn.*, 2010, pp. 365–376.



Shin-Haeng Kang received the BS degree in computer science and engineering from Seoul National University in 2010. He is currently working toward the PhD degree of the MS-PhD integrated program at Seoul National University. His research interests include design and simulation methodologies for parallel embedded systems.

Hae-woo Park received the BS degree in computer science and engineering and the PhD degree in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 2004 and 2011, respectively. He is currently a software engineer at Google. From 2011 to 2014, he was a research staff member at Samsung Advanced Institute of Technology (SAIT). His research interests include embedded system design methodology and big data processing.

Sungchan Kim received the BS degree in material science and engineering, the MS degree in computer engineering, and the PhD degree in electrical engineering and computer science from Seoul National University, Seoul Korea, in 1998, 2000, and 2005, respectively. He is currently an assistant professor at Chonbuk National University, Korea. His research interests embedded systems, parallel computing, and big data processing. He is a member of the IEEE.

Hyunok Oh received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Seoul, Korea, in 1996, 1998, and 2003, respectively. He is currently an associate professor in the Department of Information Systems, Hanyang University, Seoul, Korea. His research interests include design automation, non-volatile memory, storage systems, parallel processing, multimedia, and embedded system design. He is a member of IEEE.

Soonhoi Ha received the bachelor's and master's degrees in electronics engineering from Seoul National University, and the PhD degree in electrical engineering and computer science from the University of California, Berkeley, in 1985, 1987, and 1992, respectively. He is a full professor in the School of Computer Science and Engineering at Seoul National University. From 1993 to 1994, he worked for Hyundai Electronics Industries Corporation. He has worked on the Ptolemy project and the PeaCE (development of a HW/SW codesign environment) project. He is currently leading the HOPES (development of an embedded S/W design environment for MPSoC) project. His research interests include hardware-software codesign, design methodology for embedded systems, and embedded S/W. He is a senior member of the IEEE.