# Containment Domains: a Full System Approach to Computational Resiliency

An Initial Framework for State Preservation/Restoration and Application-Tunable Resiliency

**Michael Sullivan**
mbsullivan@mail.utexas.edu

**Doe Hyun Yoon**
doehyun.yoon@gmail.com

**Mattan Erez**
mattan.erez@mail.utexas.edu


**Locality, Parallelism and Hierarchy Group**
**Department of Electrical and Computer Engineering**
**The University of Texas at Austin**
**1 University Station (C0803)**
**Austin, Texas 78712-0240**

**TR-LPH-2011-001**
**January 2011**

This research was, in part, funded by the U.S. Government . The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

# Containment Domains: a Full System Approach to Computational Resiliency

**Abstract**

Operating the Echelon system at optimal energy efficiency under a wide range of environmental conditions and operating scenarios requires a comprehensive and flexible resiliency solution. Our research is focused in on two main directions: providing mechanisms for proportional resiliency which use application-tunable hardware/software cooperative error detection and recovery, and enabling hierarchical state preservation and restoration as an alternative to non-scalable and inefficient generic checkpoint and restart. As an interface to these orthogonal research areas, we are developing and evaluating programming constructs based on the concept of *Containment Domains*. Containment Domains enable the programmer and software system to interact with the hardware and establish a hierarchical organization for preserving necessary state, multi-granularity error detection, and minimal re-execution. Containment domains also empower the programmer to reason about resiliency and utilize domain knowledge to improve efficiency beyond what compiler analysis can achieve without such information. This report describes our initial framework and focuses more on the aspects relating to enabling efficient state preservation and restoration.

## 1 Introduction

Today's prevailing system design philosophy is to compartmentalize error-tolerance mechanisms and, if possible, hide them from the user. As a result, current reliability techniques have a fixed and high overhead, leading to significantly lower performance for a given hardware cost and energy budget. This is particularly true in the case of systems that rely on global checkpoint-rollback, where the overhead of preserving state is non-scalable and the cost of recovery is very high. In contrast to prior schemes, we advocate a cooperative and flexible approach to error resiliency. The requirements of an error detection and correction system are, by nature, application specific. Not all computations require the same amount and type of error protection. Some algorithms can be protected with low-cost algorithm-based fault tolerance [1, 2] or sanity checks [3, 4, 5], and others are naturally fault tolerant [6, 7] or can tolerate some imprecision [8, 9]. Protecting all instructions to the same degree carries either a large associated cost or a hidden risk of catastrophic failure or silent data corruption. In addition, the storage hierarchy exhibits a large, application-specific degree of natural redundancy—the vertical and inclusive hierarchy present in most high-performance machines naturally copies state through regular operation; also, computation patterns can mirror data horizontally across machines. By intelligently utilizing the natural redundancy of the storage hierarchy, many of the state preservation costs associated with traditional checkpoints can be avoided.

We introduce the concept of *Containment Domains*, programming constructs with transactional semantics which enable the scalable, efficient, and application-aware protection of programs against many types of faults. At its core, a containment domain indicates that all data generated within the domain must be checked for correctness before being communicated outside of the domain. Containment domains are nested and hierarchical, and provide a means to preserve and restore state in an optimal way within the storage hierarchy. In addition, Containment Domains also provide a mechanism for controlling the type of error detection used, which allows for the application-aware, need proportional hardening of programs.

The rest of this report is organized as follows. Section 2 defines and describes the concept of a containment domain. Section 3 gives examples of the mechanisms and granularities for state preservation and restoration that would be appropriate for different failures; a mini-benchmark is explored in detail. Section 4 investigates some related concepts and some concluding remarks are given in Section 5.

# 2 Containment Domains

Each Containment Domain has four components. The *preserve* component is called first to locally preserve state for recovery. This preservation process need not be complete—unpreserved state, if needed, will be recovered or rematerialized from elsewhere in the system. The *body* is then executed, followed by a *detect* routine to identify potential faults. Detection is always performed before the outputs of a Containment Domain are committed. Detection determines the containment scope and prevents a fault from becoming an error from the application perspective. If a fault is detected, the *recover* routine of the Containment Domain is initiated to restore the necessary preserved state and recover from the error.

Containment Domains may be hierarchically nested; failures in an inner domain are encapsulated and recovered by the domain in which they occur—therefore, they are *contained*. A specific error may be too rare, costly, or unimportant to handle at a fine granularity. For this reason, an inner Containment Domain can *elevate* certain types of errors to its parent. Figure 1 gives the organization of a hierarchy of nested Containment domains, with their four components shown. Containment Domains enable errors to be handled at the most efficient granularity, taking into account characteristics of both the running application and the underlying machine.

Each preservation and recovery component can be tailored for a particular algorithm and machine configuration to enable optimal efficiency and performance. Recovery is initiated via an active message call back, and again can be optimized by the compiler or programmer using domain knowledge to minimize preservation and restoration overhead. Examples of optimized preserve/restore/recover include restoring data from neighboring processing elements or nodes which already have a copy of the data for algorithmic reasons. Figure 2 shows three different examples of possible optimized preserve/restore/recover routines, given three different failures.

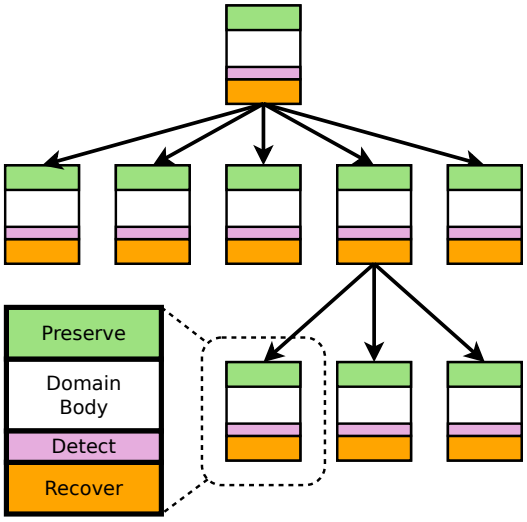
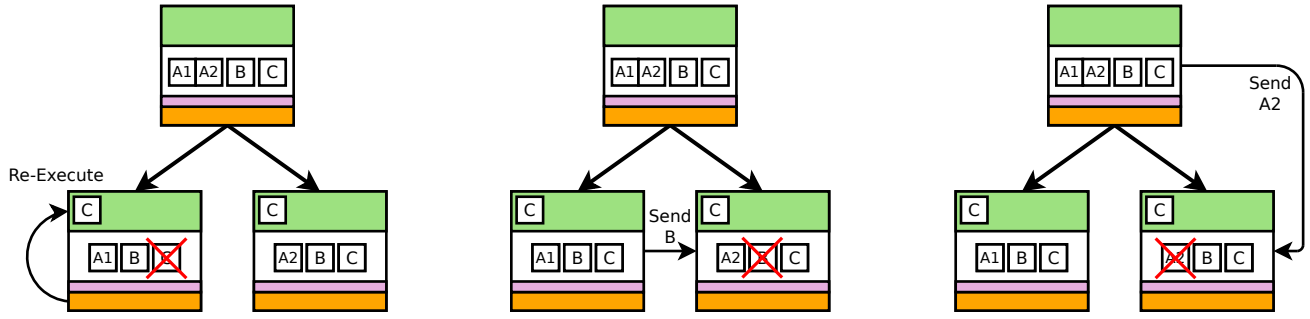



Figure 1: The organization of hierarchical Containment Domains. Each domain has four components, shown in color. The relative time spent in each component is not to scale.

The flexibility of Containment Domains allows the correct balance to be struck between the overhead of preserving state at each hierarchy level and recomputing or refetching data which was not locally preserved. Optimizing this tradeoff is critical for avoiding excessively preserved state and wasting valuable memory, bandwidth, and energy resources.

Containment Domains are both amenable to automatic analysis and code generation as well as incorporating domain knowledge. At the finest granularity, for example, each arithmetic operation can be considered a Containment Domain with detection performed through redundant execution. This overhead can be significantly reduced if algorithmic based techniques can be employed at a coarser domain which includes many arithmetic operations.




(a) Local Recovery (b) Transfer from a Neighbor (c) Transfer from a Parent

Figure 2: Examples of recovery after the detection of a failure in some input. Both local restoration (a) and optimized recovery through restoring data from non-local domains (b and c) are shown.

Containment Domains are designed and intended to work with any error detection and correction schemes present in a system. There are a variety of concurrent error detection schemes which provide low-overhead error detection for important hardware structures, given assump-

tions about the errors themselves. Further utility of containment domains presents itself when the built-in error detection mechanisms fail to provide the desired error coverage, when some coarse-grained software alternative provides a more efficient solution, or when some computations require less error protection than is built into the system. The flexible hardware-software cooperative error detection and recovery offered by Containment Domains offers a mechanism to provide sufficient resiliency to all workloads, regardless of their needs.

## 2.1 Operation within the Storage Hierarchy

Centralized storage bandwidth is at its limit for meeting the needs of current systems, and will not be able to effectively and efficiently satisfy the state preservation needs of Echelon and other future systems [10]. Containment Domains are powerful enough to preserve and restore state in an optimal way within the storage hierarchy, effectively utilizing small amounts of fast preserved state to provide superior bandwidth to a faulty running program. In addition, Containment Domains have the flexibility to run on machines with a variety of memory hierarchies, including those with supplemental non-volatile RAM. Use of NV memory can protect even long-running programs from critical and time-sensitive failures. Using NV memory carelessly, however, is not advisable because of the extra costs associated with wear-out failure avoidance [11, 12, 13]. Therefore, the intelligence, domain knowledge, and hardware-software cooperatively enabled by Containment Domains is crucial for exploiting the state preservation potential of NV memory.

In addition to the ability of Containment Domains to fully utilize the memory hierarchy and minimize the average restoration latency, the recovery routine built into Containment Domains offers an orthogonal area of potential performance optimization. Taking into account the relative availability of memory and CPU bandwidth for a given failure, an intelligent recovery routine may be able to tradeoff the advantages of rolling back the inputs of a faulty structure versus rematerializing those inputs from a reduced amount of state.

Many of the most time-intensive optimizations required for peak performance are amenable to automatic compiler and runtime analysis. Prior work has been successful at using compiler and runtime analysis to reduce checkpoint sizes [14, 15], to dynamically find producers for a faulty unit at runtime [16], and to weigh the options of preserving state versus rematerializing it [17].

## 2.2 Time and Space-Based Redundancy through Containment Domains

In the absence of any sufficient built-in mechanism for error detection, or for computations where high error coverage is critical, redundant Containment Domains may be used as a periodic error detection mechanism. Both time and space-based redundancy are possible through Containment Domains; examples of each are shown in Figure 3.

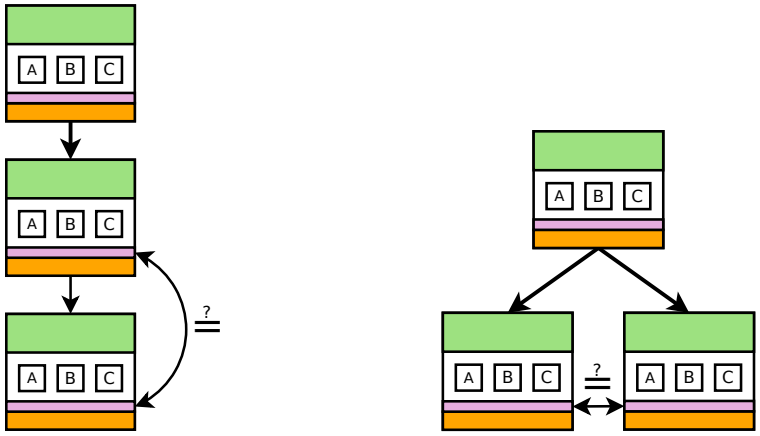



Figure 3: Two ways of providing periodic error detection with redundant Containment Domains. The outputs of each are compared at the end of execution; if an error is detected, a likely recovery mechanism would re-execute given fresh inputs from the parent.

Space-based redundancy may also be effective at protecting very critical computations from insidious errors, because Containment Domains may be mapped to physically different nodes. An example of such an error would be a multi-bit local cache error, which can occur with low probability [18], but may not be detected by a SEC-DED error correcting code.

# 3 Usage Examples

The interface between the programmer and Containment Domains is relatively simple. In order to protect a program, the programmer hierarchically partitions the program into Containment Domains, and assigns preservation, detection, and recovery mechanisms to each domain. We briefly investigate the forms that Containment Domains are likely to take. Section 3.1 describes appropriate detection and recovery mechanisms for a variety of errors, and Section 3.4 gives an example of partitioning realistic workloads through an example.

## 3.1 Detection Mechanisms for Likely Errors

Errors in computation can originate in many different parts of a machine due to many different reasons. Containment Domains are general enough to utilize a variety of error detection mechanisms at various granularities. By tailoring the level of error detection and the granularity at which each type of error is detected to application-specific demands, performance can potentially be improved.

### 3.1.1 Soft Memory Errors

Transient memory errors are relatively common and can be very problematic in large machines [19, 20]. Error checking codes are typically used for the reliable detection of memory errors, though some algorithmic error detection may be possible.
**Detection:** ECC (SEC-DED) for single, double, and some multiple-bit errors. Sanity checks or ABFT programs also detect some errors of all intensities. In addition, time-based redundancy through Containment Domains (Section 2.2) can be used to emulate single-threaded software based compiler directed fault tolerance algorithms. Space-based redundancy through Containment Domains can be used to provide a high level of memory error coverage, if desired.

**Recovery:** ECC (SEC-DED) for single-bit errors, restoration or rematerialization for multiple-bit errors.

### 3.1.2 Hard Memory Errors

In addition to temporary errors caused by environmental stimula or uncommon operating inputs, "stuck at 0" or "stuck at 1" errors can occur in memory.
**Detection:** Same as the detection of soft memory errors. We also assume the presence of a supervisory system (this can be in software, the OS, or a hardware monitoring circuit) to register the repeated failure of a memory location. Alternatively, built-in-self-test (BIST) hardware, if it exists on chip, may be used to diagnose hard memory errors [21].

**Recovery:** Remap the memory location to a different chip, if this is supported by the machine. If there is still enough memory available given the loss of the faulty chip, then recovery proceeds as it did with a soft memory error. Else, elevate the error up to the parent Containment Domain.

### 3.1.3 Soft Compute Errors

Traditionally, protection of the memory system has taken precedence over any sort of arithmetic protection, due to the higher transient error rate in memory. However, error protection in the compute logic of a processor is becoming more important; soft error rates are rising with the number of transistors on chip, and the error vulnerability of combinational logic may be increasing faster than that of memory [22, 23, 24]. Furthermore, soft errors in the execution path are the most likely to produce fail-silent data violations, which are more insidious and may be more dangerous than visible software malfunctions [25].

**Detection:** The detection of arithmetic errors is a widely studied topic, and low-cost methods of detecting many types of errors exist. The area, time, and power overheads of these methods, and the overall error coverage they provide, differs. Detection of arithmetic errors can be

provided through some built-in resiliency mechanism, such as arithmetic error coding, if it exists. Alternatively, coarse grained replication (Section 2.2) can be used to detect computational errors. Sanity checks or ABFT techniques also apply to some errors in some programs.

**Recovery:** Recovery in the presence of a transient arithmetic error normally involves the re-execution of the offending code. In the case of error detection through algorithm-based fault tolerance, recovery may be performed at a coarser granularity in an algorithm-specific manner.

### 3.1.4 Hard Compute Errors

As is the case with memory, logic is susceptible to permanent "stuck at 0" and "stuck at 1" errors.

**Detection:** Analogous to hard memory detection; soft-error detection mechanisms and a supervisory system may be used to detect hard compute errors[1]. Alternatively, BIST hardware may be used, if present.

**Recovery:** Retire the functional unit which has failed, if this is supported by the machine. If the failing node still retains all functionality (albeit at reduced capacity), then recovery proceeds as it did with a soft compute error. Else, elevate the error up to the parent Containment Domain.

## 3.2 Control Errors

Errors in the decoder or control logic are especially insidious, and are likely to crash the running program [24]. In lieu of a catastrophic failure, control errors can create severe silent data errors that are unlikely to be detected by low-cost concurrent detection mechanisms for computer arithmetic [26].

**Detection:** There exist a number of mechanisms for the low-cost detection of control errors [27, 28, 29, 30]. Any built-in technique may be used for control error detection. Alternatively, the structure of Containment Domains makes them amenable to the software-based signature or assertion-based detection of control errors [28, 29, 30], and the natural constraints of memory isolation defined by Containment Domains limit the amount that a control error may perturb the executing algorithm. In addition, sanity checks, ABFT techniques, or dedicated watchdog hardware [31] may ensure that reasonable progress is being made towards the desired goal. A supervisory system may be used to detect permanent faults in the control logic.

[1] Soft-error detection methods based on time replication can not generally be used for hard compute errors, since permanent errors, by definition, will affect multiple repeated runs on the same hardware.

**Recovery:** The usual response to a control error should be the re-execution of the Containment Domain in which the error was detected. A permanent fault in control logic should elevate an error up to the parent Containment Domain.

## 3.3 System-Level Faults

Due to an unhandled error in the network, an external failure (e.g., power failure), an error at the interface of a node, or other high-level system faults, a node may become completely unresponsive at some time during operation. In this case, the system should handle the decommissioning of dead nodes, and then either map in a spare node if one has been allocated or gracefully degrade to be able to run most workloads at diminished capacity despite the presence of failed nodes. Correctly operating with fewer resources is left for software. We expect automatic analysis to be able to introduce code to achieve this in some cases, whereas explicit code will be required in other cases.

**Detection:** Detection of dead nodes may be provided by a runtime system. A heartbeat can be used to guarantee that each node is still responsive.

**Recovery:** Given a node failure, the parent node should map out the failing node, re-execute, and redistribute the running tasks among the remaining healthy processors if the application supports it. If no spare node is available, and the application has not been written or compiled to operate with reduced resources, the application will fail. Containment domains help minimize such failure cases by providing a framework to easily map in spare nodes and provide code for redistributing work.

## 3.4 Partitioning Programs into Containment Domains

Our work over the past semester further developed the concept of Containment Domains using the concrete examples of the Echelon mini-apps. We have so far qualitatively evaluated the application of Containment Domains to the iterative sparse matrix-vector multiply mini-app. We have identified multiple opportunities for cooperative resiliency and for utilizing Containment Domains as a programming construct for expressing the required information.

We consider the mini-benchmark corresponding to a parallel sparse matrix-vector multiplication for a sparse matrix held in a block-distributed compressed sparse row format. The computation consists of iteratively multiplying a constant matrix by an input vector. The result vector is then used as the input for the next iteration. This simple application demonstrates how Containment Domains can be used to express efficient hierarchical state preservation and restoration. We also provide an example of including application-specific error detection based

on an assertion of data structure integrity. The example also shows how complex state restoration can be expressed with little additional code.

While we have not yet specified the syntax for how to integrate Containment Domains within a programming language, we provide sample code below to anchor the discussion. The code is written in the style of Sequoia [32] (`sequoia.stanford.edu`) and targets a hierarchical machine. The root-level task in the code performs the iterative computation by hierarchically decomposing the matrix multiplication. The hierarchy is formed by recursively calling SpMV to form a tree of tasks. The leaves of the tree perform sub-matrix multiplications that are then reduced into the final result. Each compute task is a Containment Domain with its own preserve and recover routines. The leaf compute tasks also includes a nested Containment Domain to verify data structure integrity. Note that the tasks also include a `restore_for_child` code block, which expresses a routine which is executed by a parent task on behalf of a child task that is recovering from a fault.

The inner tasks of the task tree specify that only the matrix should be preserved locally, ideally in non-volatile memory. The matrix does not change, so it only needs to be preserved once. Each node has its own unique portion of the matrix, and thus if a node (at any level) is lost, the matrix can only be recovered from preserved if the matrix is not preserved, then any data corruption may be unrecoverable. This is not true for the vector though, which is distributed in such a way that multiple copies of the vector are created. This is because there are NB$\times$NB sub-blocks of the matrix, but only NB sub-blocks of the vector. If a fault occurs that requires recovering a vector for a task, the vector is simply copied from another task that holds it. This second task is determined by the parent of the faulting task, as shown in the `restore_for_child`.

The resiliency of this application is described below. Global default detect and recover routines are provided, which include duplicating instructions to verify arithmetic correctness (each instruction is an implicit Containment Domain), as well as runtime detection of memory errors and of faulty (non-responsive) nodes and processing elements. When a fault is detected, the default recovery procedure is triggered so that re-execution can occur. First, the type of fault is determined. If the fault is at an instruction-level containment domain, the instruction is retried. If this retry fails then the fault is not a soft error and recovery must occur. Recovery starts by identifying a new node (or processin element) on which the containment domain can be re-executed. This is potentially the same node, if the fault is in memory and there is still sufficient capacity to run the domain. Alternatively, a spare can be mapped by the system, or rely on the application to provide a means of remapping the problem to a smaller number of nodes (often simple to do). The state is then restored onto the new node, which is done using a default handler. The handler restores the matrix data from the explicitly preserved local state and requests all other state from the parent. The parent then provides the needed data using the `restore_for_child` routine, which in this case simply copies over the input vector data from a neighboring task. At this point all inputs have been restored and re-execution occurs. Note that this procedure is entirely distributed, and the computation of other tasks is not interrupted. This

Listing 1: Sequoia-like code for sparse matrix-vector computation with Containment Domains.

```
// This is Sequoia code for doing the sparse matrix-vector multiplication from the
//   Echelon Shoch Hydrondynamics mini-app, which applies this repeatedly by copying
//   the result into the input of next iteration.

// Assume pre-partitioned input; partitioned all the way to finest size,
//   but still go through hierarchy for locality and resiliency
// Data structure has CSR representation for each block.
// Mapping to global vector maintained by caller.
//
// The partitioning is into NBxNB blocks, with each block having a maximum of
// NNZ_B non-zeroes and N_B vector elements. Note that these numbers and sizes
// are different for each level of hierarchy, but will be multiples of the finest
// level.

void task<root> IterativeSpMV(in  double data[NB][NB][NNZ_B],
                              in  uint   colIdx[NB][NB][NNZ_B],
                              in  uint   rowStart[NB][NB][N_B+1],
                              in  double input[NB][N_B],
                              out double result[NB][N_B]) {

  mapseq(int i=0 : NITER) { // unroll by 2 technically. NITER is number of iterations
    SpMV(data, colIdx, rowStart, input, result);
    input = result;
  }
}

void task<inner> SpMV(in  double data[NB][NB][NNZ_B],
                      in  uint   colIdx[NB][NB][NNZ_B],
                      in  uint   rowStart[NB][NB][N_B+1],
                      in  double input[NB][N_B],
                      out double result[NB][N_B]) {

  tunable blockSize; // size of block for this level of hierarhcy (num fine-grained blocks)

  mappar(int i=0 : NB/blockSize) { // assume NB divisible by blockSize, mappar == forall
    // Containment domain: Implicit liveness checks to make sure each of the reductions completes.
    // If any fault exposed to this level then respawns entire set of reduce tasks from
    //   inputs (or preserved inputs if preferable).
    mapreduce(int j=0 : NB/blockSize) { // like mappar, but with reduction at end of each task
      //  Containment domina: implicit liveness check for completion of each task
      //   If any fault exposed then respawn task from inputs:
      //     - 'data', 'colIdx', and 'rowStart' restored from local preserved data
      //     - 'input' has to be re-read from level above, but parent can provide data
      //       a neighbor to minimize storage and bandwidth overhead.
      // using explicit contain directive here because of specialized recovery
      contain {
        SpMV(data[i*blockSize;blockSize][j*blockSize;blockSize][NNZ_B],
             colIdx[i*blockSize;blockSize][j*blockSize;blockSize][NNZ_B],
             rowStart[i*blockSize;blockSize][j*blockSize;blockSize][N_B+1],
             input[j*blockSize;blockSize][N_B],
             reducearg<result[i*blockSize;blockSize][N_B],VectAdd>);
      }
      preserve { // explicit preserve block for this domain
        preserve_once_NV(data);     // hint to preserve in NV-RAM because state never changes
        preserve_once_NV(colIdx);   // system implicitly runs this block only once because state
        preserve_once_NV(rowStart); // valid for entire application after that
      }
      restore_for_child { // override default handler for restoring data for child
        restore(input,input[j*blockSize;blockSize][N_B]); // explicitly states input
        // can come from any place that stores this particular portion of the vector
        // specific location cannot be determined by programmer, but is set by the
        // compiler during mapping
      }
    }
  }
}

void task<leaf> SpMV(in  double data[NNZ], // number of non-zeros
                     in  uint   colIdx[NNZ],
                     in  uint   rowStart[N+1], // number of rows
                     in  double input[N], // number of input vector elements
                     out double result[N],
                     in  bool   firsttime) { // Containment A (nothing really)
  // compute the matrix-vector multiplication
  contain { // to express explicit preservation of matrix
    for (uint r=0; r<N; r++) {     // iterate over the rows of the result
      // Implicit checks of all arithmetic by default. SpMV BW bound so not an issue
      result[r] = 0.0;
      prev_col = 0; // for assertion on data structure integrity
      for (uint c=rowStart[r]; c<rowStart[r+1]; c++) {
        // Implicit checks for all arithmetic
        // Explicit additional check for data structure consistency
        // Not very meaningful here, but demonstrates the concept
        contain {
          //iterate over the nonzeroes in row r
          result[r] += data[c] * input[colIdx[c]]; // accumulate each scalar product into the result
          // can replicate arithmetic operations automatically in compiler or HW
          // If arithmetic fails check just redo last instruction
          // Assumption that hardware protects stores well. If not, need to expose faulting store up
        }
        detect { // additional detection for data structure integrity
          default_detect(); // do default and add to it
          assert(colIdx[c] >= prev_col);
          prev_col = colIdx[c];
        }
      }
    }
  }
  preserve { // explicit preserve block for this domain
    preserve_once_NV(data);     // hint to preserve in NV-RAM because state never changes
    preserve_once_NV(colIdx);   // system implicitly runs this block only once because state
    preserve_once_NV(rowStart); // valid for entire application after that
  }
}

void task<leaf> VectAdd(in double v1[N], inout double v2[N]) {
  for (int i=0; i<N; i++) {
    // implicit check of arithmetic
    v2[i] += v1[i];
  }
}
```

occurs automatically because tasks do not see partial results of other tasks and will block for the correct results to be available (after detection returns no errors).

In total, the modifications to partition iterative matrix-vector multiplication into Containment Domains add about 15 lines of code—two explicit preserve statements and the code for restoring data from a neighbor task. These 15 lines of code allow the system to perform optimal hierarchical and distributed state preservation and restoration.

# 4 Previous Literature

The concept of containment domains is closely related to a large body of prior work. This includes the distributed, hierarchical checkpointing used in large-scale compute clusters as well as programming languages which use hierarchical transactional semantics to interface with checkpointed state.

## 4.1 State Preservation and Restoration

Checkpointing is a generic state preservation and restoration mechanism. Large-scale compute clusters such as supercomputers have widely adopted periodic checkpointing of long-running workloads to tolerate frequent failures. Most current systems take a global checkpointing approach, establishing a synchronized program state of every node in a centralized array of disks. Global checkpointing, however, is not feasible in future systems; the time required to checkpoint the system state increases, because the application working set size is increasing, while I/O bandwidth for transferring data to a centralized non-volatile location does not scale. Oldfield et al. [33] predict that a 1-peta FLOPS system can potentially spend more than 50% of its time checkpointing global state.

Researchers have proposed checkpointing to distributed or local storage in order to overcome the inefficiencies associated with centralized global checkpoints. We first review a generic hybrid local/global checkpointing mechanism, then describe further examples including checkpointing to distributed storage and multi-level checkpointing. Local checkpointing stores the state of each node locally in non-volatile storage. It is faster and more scalable than global checkpointing, because data is transferred only within each node, and each node can checkpoint state in parallel. A naïve implementation of local checkpointing, however, cannot recover from permanent node failures. Hence, global checkpointing is often combined with local checkpointing to tolerate such failures. A typical hybrid strategy is to take local checkpoints frequently to deal with most common case failures, such as soft errors or timing violations, while infrequent and expensive global checkpoints provide a safetynet against node failures. Generic reliability models for two-level local/global checkpointing are studied in [34, 35].

Chiueh et al. propose an alternative diskless checkpointing mechanism [36] that uses DRAM

for storing both local and global checkpoints. Their mechanism splits the DRAM memory in each node into four segments and employs three fourths of the memory to make checkpoints. Every node's memory has three redundant copies in the system: one locally in its own memory and the remaining two stored in neighboring nodes. Plank and Li [37] suggest a similar, but different, diskless checkpointing mechanism using mirroring and parity.

Dong et al. use phase-change memory (PCRAM) for fast local/global checkpointing [12]. Their mechanism uses a 3D stacked PCRAM die on top of the DRAM die connected via Through-Silicon-Vias (TSV). TSV provides a high data transfer rates between the DRAM and 3D stacked PCRAM. As a result, local PCRAM checkpointing has only a negligible impact on system efficiency. The non-volatile nature of the stacked local PCRAM also lowers the power consumption of the system as a whole.

Moody et al. further extend local/global checkpointing into multi-level checkpointing [38]. Multi-level checkpointing employs multiple types of checkpoints with different costs and different levels of resiliency in a single run.

The basic rationale of containment domains is similar to that of two-level (local/global) or multi-level checkpointing. Each approach employs different levels of state preservation and restoration such that a failure is constrained within a domain. This domain is an explicit level of the storage hierarchy for containment domains and an implicit time interval between checkpoints in two- or multi-level checkpoints. The use of multiple levels of state preservation and restoration allows for light-weight state preservation and restoration to handle the most common failures; less common, but more severe, faults are recovered using more expensive state preservation and restoration mechanisms.

Containment Domains have the potential to be more efficient than two-level or multi-level checkpoints because the domains themselves exploit the vertical storage hierarchy which is natural in modern computing systems. In addition, Containment Domains are flexible and allow the programmer and software system to interact with the hardware to preserve necessary state, provide multi-granularity error detection, and use minimal re-execution to maximize the performance of the system. Generic checkpointing approaches, on the other hand, only have the notion of time intervals between state preservation and restoration; the programmer has little control over state preservation and restoration, thus making these checkpoints inefficient and inflexible to application needs. An orthogonal advantage of containment domains are their potential for adaptive and tunable reliability due to their hardware/software interaction and dynamic behavior.

Orthogonal to local checkpointing, researchers have suggested distributed checkpointing mechanisms. In distributed checkpointing, each node takes checkpoints independently. This uncoordinated distributed checkpointing, however, may result in the *domino effect* [39]—given an arbitrary set of interacting processes, each with its own checkpoint, a single error local to one process can cause all processes to use up all of their checkpoints. Common approaches to

eliminate the domino effect include log-based approaches [40, 41] and coordinated checkpointing based on communication history [42, 43]. These mechanisms, however, often sacrifice a certain degree of process autonomy and incur run-time and extra message overheads [44].

## 4.2 Software Interface to Preserved State

Randell proposed a software fault-tolerant programming interface, *Recovery Blocks* [39]. Each program is decomposed hierarchically into nested elements called Recovery Blocks. Each Recovery Block is implemented in two independent yet equivalent routines: the primary alternative (AP), and the secondary alternative (AS). The AP contains the body of the Recovery Block; it implements the functionality of the block in the most optimized manner. The execution of the AP is checked by a pre-defined checker, the acceptance test (AT). If the outcome of AP does not pass the AT requirements, then the system state is rolled back to the beginning of the Recovery Block, and the system executes the subordinate secondary alternative (AS). The AS implements the same functionality as the AP, but it is simpler and less optimized than the AP. Proper implementation of the AS may tolerate hard, soft, or software errors that prevent the AP from passing the AT requirements. If the AS also fails, the Recovery Block reports an error. Figure 4 shows an example of a Recovery Block which might be used to protect a sort routine. The programmer can design hierarchical Recovery Blocks such that an outer (higher level) Recovery Block can resolve the error from an inner (lower level) Recovery Block.

Conceptually, Containment Domains are similar to Recovery Blocks. Both Containment Domains and Recovery Blocks constrain the detection and correction of errors to a local boundary, and are able to pass uncorrectable errors upwards to be handled by the parent domain or block. Unlike Recovery Blocks, Containment Domains are able to utilize any available hardware error detection mechanisms. Also, Containment Domains offer more aggressive optimization by allowing for the partial preservation of state, application-aware recovery and rematerialization, and by mapping naturally to the storage hierarchy.

```
recovery_block sort (S) {
   S_backup = S; // preserve the state for restoration
                 // this was originally achieved through "recursive cache" structure
   quicksort(S); // execute AP
   if( sum(S) != sum(S_backup)  ) { // test against AT
                                    // the sums of the original and the sorted vectors should match
      S_backup = S; // restore the system state
      bubblesort(S); // execute AS
      if( sum(S) != sum(S_backup)  ) { // test against AT
         return ERROR; // report errors if neither AP nor AS passes
      }
   }
   return OK;
}
```

Figure 4: Recovery block example – a sort function in a C-like syntax. *Note: Randell's paper originally used Algol or PL/I language as the baseline representation.*

The construct of *transactions* has been primarily developed for controlling concurrency in data

base systems [45], but they can also be used to enforce reliability. Transactions, if so defined, can be nested to any depth, constructing a *tree*. The nested transaction structure is naturally used to localize the effects of failures within the closest possible level of nesting in the transaction nesting tree. As a result, a failed transaction has no effect on the data or on other transactions. This style of programming, including [46, 47, 48], is a generalization of the recovery block to the domain of concurrent programming.

Spheres of Control are a logical construct proposed by Davies which are designed to aid in the control, auditing, and error recovery of distributed programs [49]. Spheres of Control are nested and transactional in nature, and can rollback any errors which occur before a transaction has been committed. In addition, Spheres of Control are able to backtrack and recover errors which occur *after* the end of a sphere. The mechanism which allows for error recovery past domain boundaries requires all writes to be journaled; accordingly, Spheres of Control incur hefty implementation overheads [50]. Containment Domains are unlike Spheres of Control in that they strictly enforce containment within domain boundaries. Also, Containment Domains enable the partial preservation of state, and more flexibly map to differing machine and application requirements. Containment Domains, as such, should be more implementable than pure spheres of control, with a higher potential for performance optimization.

Argus is a distributed programming language with inherent hard fault tolerant features [51, 48]. The semantics of Argus are nested and transactional, and deal with reliability through the application of an abstract object called a Guardian. Guardians encapsulate resources on a per-node basis, and associate additional handler routines with any communication to and from the encapsulated resources. The purpose of a guardian is to preserve sufficient information about a resource to non-volatile state such that a node failure may be tolerated. Upon a temporary node failure, a recovery routine, associated with each Guardian, is used to recover the full state of the Guardian from the partial preserved state. In the context of Containment Domains, Guardians are significant because they have a concept of mapping to the storage hierarchy, provide partial preservation and restoration of state, and associate specialized recovery routines with code which is executed following a catastrophic failure. All of this can be done with Containment Domains, as well as protection against soft errors in an application aware manner. Also, Containment Domains are conceptually simpler and more general than Argus, which should increase their programmability, adaptability, and usefulness.

Recent research on transactional memory (TM) [52, 53, 54] mainly aims to enable efficient concurrent and lock-free programming. Transactional memory inherently provides state preservation and restoration at transaction boundaries. Similar to nested transactions, TM also supports nested transactional memory models [55, 56]; the two prevalent nesting models for TM are *closed nesting* and *open nesting*. In closed nested TM, the effect of committed sub-transactions are available only to its direct parent transaction, and updates to the global memory space are deferred until the top-level transaction commits. Open nesting, on the other hand, allows sub-transactions to

commit to global memory space even before the parent commits.

The programming model for closed nesting transactional memory is similar to that of Containment Domains in many ways. Both have a hierarchical structure where a domain or a transaction localizes a failure and provides state preservation and restoration at the domain or transaction boundary. Containment Domains, however, allow more aggressive optimization than do nested transactional memory by exploiting the mapping of data to the storage hierarchy; state preservation is often free of charge, since multiple copies of any data exist in high-speed storage throughout the system. Furthermore, Containment Domains allow for selective re-materialization of data, in order to leverage compute as well as memory elements during system recovery.

There has been prior work that extends transactional memory concepts to reliability, including Relax [57] and FaulTM [58]. Both use TM-like semantics for efficient hardware/software collaborative reliability; both provide state preservation and restoration at a transaction boundary. However, neither exploits nested transactions for localizing failures to the closest domain, nor do they allow for further application or machine-specific optimizations which are enabled by Containment Domains.

Krujif et al.'s Relax [57] uses try/catch like semantics to provide reliability though a cooperative hardware-software approach. Relax relies on low-latency hardware error detection capabilities while software handles state preservation and restoration. The programmer uses the Relax framework to declare a block of instructions as "relaxed". It is the obligation of the compiler to ensure that a relaxed code block can be re-executed or discarded upon a failure. As a result, hardware can relax the safety margin (e.g., frequency or voltage) to improve performance or save energy, and the programmer can tune which block of codes are relaxed and how the recovery is done. Containment domains, unlike Relax, exploits the storage hierarchy that inherently provides isolated execution at each domain. As a result, it minimizes the cost of state preservation and restoration; often, comes at no cost due to inherent data replication through the storage hierarchy.

Yalcin et al.'s FaulTM [58] is another research project which uses transactional semantics for reliability. FaulTM uses hardware transactional memory with lazy conflict detection and lazy data versioning to provide hybrid hardware-software fault-tolerance. While the programmer declares a *vulnerable* block (similar to transactional memories and Relax), lazy transactional memory (in hardware) enables state preservation and restoration of a user-defined-block. FaulTM duplicates a vulnerable block across two different cores for reliable execution. The selective, node-level duplicate execution used by FaulTM can also be achieved using Containment Domains (Section 2.2). Containment Domains, however, give the programmer much greater flexibility when selecting the appropriate error detection, state preservation, and state restoration mechanisms for an application.

```
relax( rate ) { // hardware detects an error within a relax block
  do something important
} recover { retry; } // software defined recovery handler
```

(a) RELAX Syntax

```
vulnerable { // vulnerable section is duplicated on a different core
  do something important
} // relies on lazy hardware transactional memory
```

(b) FaulTM Syntax

Figure 5: The syntaxes that RELAX and FaulTM use to provide reliability.

# 5 Conclusion

While Containment Domains are related to number of prior approaches, no existing approach offers the flexibility, potential for optimization, and ability to incorporate machine and domain-specific knowledge that Containment Domains do for the field of resilient computing. By bounding error propagation, operating within the storage hierarchy, incorporating programmer knowledge, and utilizing all available error detection mechanisms, Containment Domains are able to optimize the error detection and recovery process, while remaining usable.

Containment Domains are able to effectively incorporate domain knowledge, while being manageable for the programmer. A Containment Domain can specify that only some data be preserved whereas other data can be re-generated or re-read from elsewhere in the machine. This, combined with the flexibility of error detection and recovery routines, allows for application-specific optimization of the error recovery process. The burden placed on the programmer is minor; we have demonstrated that for all types of errors there are a reasonable set of error detection and recovery options, based on what the machine has available. It should be possible to set reasonable default behaviors for Containment Domains on a machine-specific basis, which only need to be overridden if a domain can benefit from domain-specific knowledge. Based on the analysis of the iterative sparse matrix-vector multiplication mini-benchmark, it appears that partitioning a program into Containment Domains should be achievable in few lines of code. In addition, many of the most demanding optimizations required for peak performance should be well suited for automatic compiler and runtime analysis and code generation.

The Containment Domain construct seems promising, and has shown some early success in efficiently protecting parallel code for a hierarchical machine with few additional lines of code. In the future, we will implement an evaluation framework to more fully analyze Containment Domains, as well as investigate their suitability for more relaxed programming models.

## Acknowledgements

This research was funded in part by DARPA contract HR0011-10-9-0008. The authors also wish to greatfully acknowledge the input of Mehmet Basoglu, Jinsuk Chung, Evgeni Krimer, Ikhwan Lee, Karthik Shankar, and Jongwook Sohn for helping formulate the ideas contained in this report.

## References

[1] H. Kuang-Hua and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, pp. 518–528, 1984.

[2] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, no. 10, pp. 1304–1308, 1990.

[3] D. M. Andrews, "Using executable assertions for testing and fault tolerance," in *9th Fault-Tolerance Computing Symposium*, Madison, Wisconsin, USA, June 1979.

[4] A. Mahmood, D. J. Lu, and E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions," in *1983 International Test Conference*, Philadelphia, Pennsylvania, USA, October 1983, pp. 622–628.

[5] M. Z. Rela, H. Madeira, and J. G. Silva, "Experimental evaluation of the fail-silent behaviour in programs with consistency checks," in *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, 1996, p. 394.

[6] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, pp. 226–244, April 1978.

[7] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors," 2002.

[8] X. Li and D. Yeung, "Exploiting application-level correctness for low-cost fault tolerance," 2008.

[9] R. A. Shafik, B. M. Al-Hashimi, S. Kundu, and A. Ejlali, "Soft error-aware voltage scaling technique for power minimization in application-specific MPSoC," *Journal of Low Power Electronics (JOLPE)*, vol. 5, no. 2, pp. 145–156, February 2009.

[10] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, 2008.

[11] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *J. Vacuum Science and Technology B*, vol. 28, no. 2, pp. 223–262, March/April 2010.

[12] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *Proc. the Int'l Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2009.

[13] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abail, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. the Int'l Symp. Microarchitecture (MICRO)*, December 2009.

[14] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative checkpointing theory," in *Proc. the Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[15] G. Bronevetsky, D. J. Marques, K. K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *Proc. the ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[16] P. M. Merlin and B. Randell, "State restoration in distributed systems," 1978.

[17] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ser. PLDI '92. New York, NY, USA: ACM, 1992, pp. 311–321.

[18] C. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 397 – 404, 2005.

[19] T.-T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *Reliability, IEEE Transactions on*, vol. 39, no. 4, pp. 419 –432, Oct. 1990.

[20] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 205 – 209.

[21] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-XII. New York, NY, USA: ACM, 2006, pp. 73–82.

[22] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128 – 143, 2004.

[23] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, 2002.

[24] G. Saggese, A. Vetteth, Z. Kalbarczyk, and I. Ravishankar, "Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005, pp. 760 – 769.

[25] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer, "An Experimental Study of Soft Errors in Microprocessors," *IEEE Micro*, vol. 25, no. 6, 2005.

[26] T. R. N. Rao, *Error Coding for Arithmetic Processors*. Orlando, FL, USA: Academic Press, Inc., 1974.

[27] J. Ohlsson and M. Rimen, "Implicit signature checking," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, Jun. 1995, pp. 218 –227.

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

[29] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, ser. DFT '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 581–.

[30] R. Venkatasubramanian, J. Hayes, and B. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, 2003, pp. 137 – 143.

[31] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 32. Washington, DC, USA: IEEE Computer Society, 1999, pp. 196–207.

[32] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.

[33] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, "Modeling the impact of checkpoints on next-generation systems," in *Proc. the IEEE Conf. Mass Storage Ssytems and Tech. (MSST)*, 2007.

[34] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *Proc. the Joint Int'l Conf. Measurement and Modeling of Computer Sys. (SIGMETIRCS)*, 1995.

[35] B. S. Panda and S. K. Das, "Performance evaluation of a two level error recovery scheme for distributed systems," in *Proc. the Int'l Workshop on Distributed Computing, Mobile and Wireless Computing (IWDC)*, 2002.

[36] T.-C. Chiueh and P. Deng, "Evaluation of checklpoint mechanisms for massively parallel machines," in *Proc. the Ann. Symp. Fault Tolerant Computing (FTCS)*, 1996.

[37] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Tr. Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, Oct. 1998.

[38] A. T. Moody, G. Bronevetsky, K. M. Mohror, and B. R. de Supinski, "Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system," Lawrence Livermore National Laboratory (LLNL), Tech. Rep. LLNL-TR-440491, July 2010. [Online]. Available: https://e-reports-ext.llnl.gov/pdf/391238.pdf

[39] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 437–449.

[40] E. Elnozahy and W. Zwaenepoel, "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit," *Computers, IEEE Transactions on*, vol. 41, no. 5, pp. 526 –531, May 1992.

[41] B. H. L. Alvisi and K. Marzullo, "Nonblocking and orpha-free message logging protocols," in *Proc. IEEE Fault Tolerant Computing Symp. (FTCS)*.

[42] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proc. IEEE Symp. Reliable Distr. Syst.*, 1991.

[43] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[44] Y.-M. Wang and W. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on*, Oct. 1993, pp. 78 –85.

[45] N. A. Lynch, "Concurrency control for resilient nested transactions," in *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, ser. PODS '83. New York, NY, USA: ACM, 1983, pp. 166–181.

[46] D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, MIT Laboratory for Computer Science, 1978.

[47] J. E. B. Moss, "Nested transactions: An approach to reliable distributed computing," Ph.D. dissertation, MIT Laboratory for Computer Science, 1981.

[48] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 381–404, July 1983.

[49] C. T. Davies, Jr., "Data processing spheres of control," *IBM Systems Journal*, vol. 17, no. 2, pp. 179 –198, 1978.

[50] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, 1st ed. Morgan Kaufmann, September 1992.

[51] B. Liskov, "Distributed programming in argus," *Commun. ACM*, vol. 31, pp. 300–312, March 1988.

[52] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[53] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2004, p. 102.

[54] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006, pp. 254–265.

[55] J. E. Moss and A. L. Hosking, "Nested transactional memory: Model and preliminary architecture sketches," in *Proc. the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[56] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," in *Proc. the Ann. Int'l Symp. Computer Architecture (ISCA)*, 2006.

[57] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proc. the Ann. Int'l Symp. Computer Architecture (ISCA)*, 2010.

[58] G. Yalcin, O. Unsal, I. Hur, A. Cristal, and M. Valero, "FaulTM: Fault-tolerant using hardware transactional memory," in *Proc. the Workshop on Parallel Execution of Sequential Programs on Multi-Core Architecture (PESPMA)*, 2010.