Containment Domains: a Full System Approach to Computational Resiliency

Containment Domains Semantics

Revision 0.1

Michael Sullivan Ikhwan Lee Jinsuk Chung
Song Zhang Seong-Lyong Gong Derong Liu
Michael LeBeane Mattan Erez
`cds@lph.ece.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-2013-001
October 2013

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.

Abstract

Future large-scale computer systems will require a comprehensive and flexible resilience solution to maintain correctness while satisfying strict energy constraints. Toward this end, this document presents the definition of the semantics of scalable and efficient *containment domains* (CDs) resilience framework. Containment domains are a programming construct that enables applications to express, tune, and specialize error detection, state preservation and restoration, and recovery to satisfy application-specific resilience needs. Containment domains have weak transactional semantics and are nested to take advantage of deep machine and application hierarchies. The flexibility and hierarchy of containment domains enables the programmer and software system to match state preservation, error detection, and recovery costs with the rate and severity of errors. Containment domains also empower the programmer to reason about resiliency and to utilize domain knowledge to improve efficiency beyond what compiler analysis can achieve without such information. This report describes CD semantics and identifies the system features and mechanisms required for an efficient containment domains implementation.

Revision History

Version 0.1	October 2013	Initial draft for public comments.

Contents

1

Introduction

Reliability and resilience are major obstacles on the road to exascale computing. The massive number of components required for exascale systems and the decreasing inherent reliability of components in future fabrication technologies may result in error and fault rates that are orders of magnitude higher than those of petascale systems today. Currently, the prevailing system design philosophy is to compartmentalize error-tolerance mechanisms and 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. Such reliability and resilience techniques simply cannot cope with the simultaneous increase in scale and fault rates expected at the Exascale while maintaining high efficiency.

Containment domains (CDs) are a new approach that achieves low-overhead resilient and scalable execution [1]. CDs abandon the prevailing one-size-fits-all approach to resilience and instead embrace the diversity of application needs, resilience mechanisms, and the deep machine and application hierarchies expected in exascale systems. Containment domains give software a means to express resilience concerns intuitively and concisely. With CDs, software can preserve and restore state in an optimal way within the storage hierarchy and can efficiently support uncoordinated recovery. In addition, CDs allow software to tailor error detection, elision (ignoring some errors), and recovery mechanisms to algorithmic and system needs. To achieve these goals, CDs rely on five key insights: (1) different errors and faults can be tolerated most efficiently in different ways; (2) machines are becoming increasingly hierarchical and this hierarchy can be exploited to reduce resilience overhead; (3) scalable execution requires uncoordinated local recovery for common-case errors; (4) it is often more efficient to trade off lower error-free execution overheads for higher recovery overheads; and (5) carefully designed and analyzed algorithms can ignore, or efficiently compensate for, some errors without resorting to rollback and re-execution.

Containment domains incorporate these insights within a programming model and system through specialized programming constructs. Containment domain constructs have weak transactional semantics and are designed to be nested to form a CD hierarchy. The core semantics of a containment domain are that all data generated within the CD must be checked for correctness before being communicated outside of the domain and that each CD provides some means of error recovery. Failures in an inner domain within the CD hierarchy are encapsulated and recovered by the domain in which they occur—therefore, they are *contained* without global coordination. That being said, a specific error may be too rare, costly, or unimportant to handle at a fine granularity. For this reason, an inner (nested) CD can *escalate* certain types of errors to its parent. This flexibility of expressing how and where to preserve and restore data, as well as how to recover from errors, sets CDs apart from prior system-level reliability schemes.

2

CD Semantics and Usage

As stated above, the main goal of containment domains is to enable a truly scalable model and framework for resilience that is used consistently across all system levels and all failure and error types and rates. Thus, the semantics of containment domains are designed to be hierarchical and distributed and enable and promote proportional and tunable, yet portable, resilience schemes. We discuss how this is achieved below, starting with a general description of CD components, followed by a discussion of state preservation, error/failure detection, recovery, and implications and tradeoffs associated with the CD hierarchy and communicating tasks in both a message-passing and global-address space context.

2.1 Containment Domain Components and Semantics

Each containment domain has four explicit components, which enable the application to express resilience tradeoffs. The *preserve* component locally and selectively preserves state for recovery. This preservation need not be complete – as explained later, unpreserved state, if needed, may be recovered from elsewhere in the system or rematerialized. The *body* contains all non-CD aspects of the computation. All computation passes through a *detect* routine to identify potential errors. Detection is always performed before the outputs of a CD are committed and advertised to other CDs. Detection can be as simple as utilizing underlying hardware and system mechanisms or may be an efficient algorithm-specific acceptance test. Specialized detection can also be designed to ignore errors that are known not to impact the running computation. If a fault is detected, the *recover* routine is initiated. Recovery may restore the necessary preserved state and re-execute the CD, or it can escalate the error to the parent CD. Note that logically the body is preceded by preservation and followed by detection, but in practice preservation and detection may be intertwined with body computation.

CDs are designed to be hierarchically nested; failures in an inner domain are encapsulated and recovered by that inner domain whenever efficiently possible. Erroneous data is conceptually never communicated outside of a CD, and there is no risk of an error escaping containment. Because of constraints on available storage, bandwidth, and the need for inter-CD communication, some errors and faults are too rare or costly to recover at a fine granularity. If such an error occurs, an inner CD escalates the error to its parent, which in turn may escalate it further up the hierarchy until some CD can recover the error. Potentially, an error may be escalated to the root of the CD tree, which is functionally equivalent to recovery through global checkpoint-restart (*g-CPR*).

Figure 2.1 gives the organization of a hierarchy of nested CDs, with their four components shown. As we discuss later, the CD tree is formed in two major steps: (1) specification of *algorithmic CDs* that define all possible legal CD trees by specifying CDs and nesting from the perspective of the application; and (2) instantiation of *mapped CDs*, which are the actual CDs required to achieve desired resilience on a specific machine and which include system-level components and mechanisms.

Using the hierarchy we restate the fundamental notion of containment domains — that each CD does not propagate errors and is recoverable — as the refined semantics below:

1. Any error occurring while a CD is running must either:
   a) be detected within the CD (is *detectable by the CD*),
   b) be inconsequential to the application, or
   c) have a specific detector at an ancestor of the CD.
2. Any detected error triggers recovery by either:
   a) using the recovery routine of the CD that detected the error (typically, but not necessarily, by: (1) killing all active descendants, (2) restoring the preserved state of the recovering CD, and (3) re-executing the CD), or
   b) escalating to the parent CD for recursive recovery.
3. Communication between tasks must obey the following rules:
   a) Tasks belonging to the *same* CD (not just contained by a single CD) communicate freely. A CD that does not communicate with any other CD is a *strict CD*.
   b) Tasks not in the same CD may only perform error-free communication, with respect to detectable errors by the CD. A CD that does communicate with other CDs is a *relaxed CD*.

4. A CD completes, or *commits*, once all potential detectable errors and failures within it have been detected; the implication is that a CD may only complete once all of its children have completed. Upon completion, the following actions take place:

    a) Any state preserved by the CD that is not preserved at the parent is added to the parent's preservation. This is necessary to enable the child CD to correctly re-execute if the parent has to re-execute and enables a program where the parent does not a priori know all the data required by its descendants.

    b) In some cases, relaxed CDs require dynamic logging of some communication and state for correct recovery (see Section 2.6). If the completing CD is a relaxed CD, any data preserved via logging is also added to the parent CD's preservation *if* the parent is also a relaxed CD; note that the implementation must correctly identify communication with the correct tasks on re-execution.

    c) All preserved state is freed (once appropriate state added to the parent).

    d) All logs are freed (once appropriate entries added to parent). Note that these semantics free logged entries once the CD that contains all the CDs that may have communicated with one another completes; i.e., the least common strict ancestor of a group of communicating CDs completes.

As a result of these semantics, containment domains offer hierarchical and distributed preservation and recovery. We discuss these aspects for detection, preservation, communication, and recovery below.

2.2 State Preservation

The first logical component of every CD is a state-preservation routine, which preserves the state necessary for CD recovery. This explicit integrated preservation can be tailored to exploit natural redundancy within the machine. A CD does not need to fully preserve its inputs at the domain boundary; partial preservation may be utilized to increase efficiency if an input naturally resides in multiple locations and when preservation is more costly than escalation or data rematerialization. Examples for optimizing preserve/restore/recover routines include restoring data from sibling CDs or other nodes which already have a copy of the data for algorithmic reasons.

Figure 2.2 shows three different examples of possible optimized preserve/restore/recover routines, given three different failures. The flexibility of CDs allows the correct balance to be struck between the overhead of preserving state at each hierarchy level and

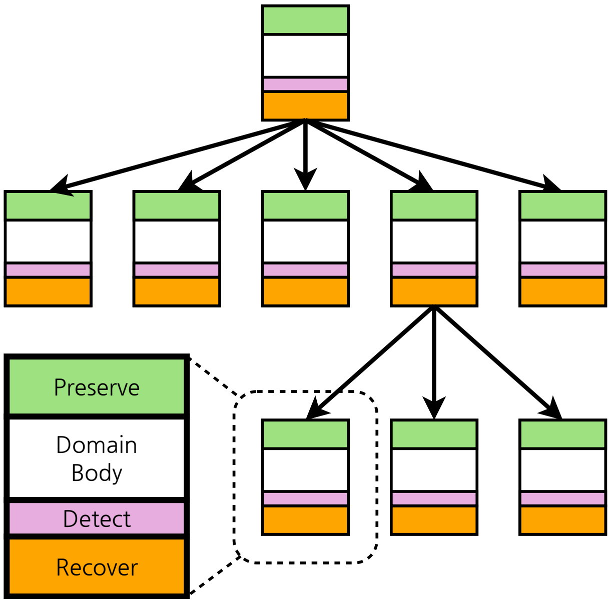
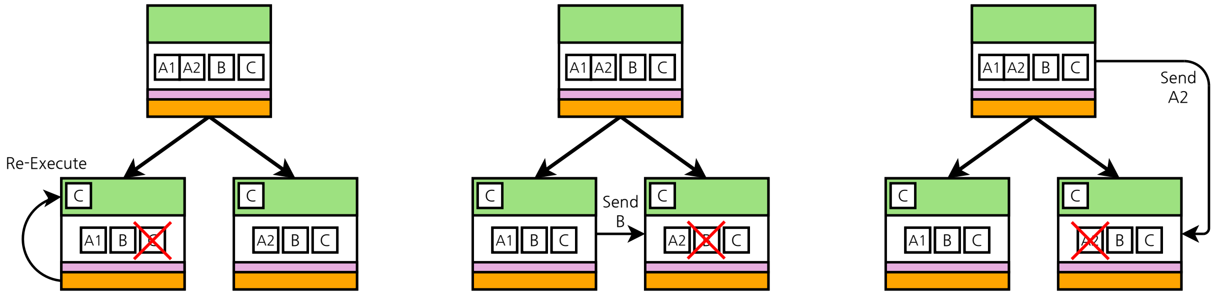


Figure 2.1: The organization of hierarchical CDs. Each domain has four components, shown in color. The relative time spent in each component is not to scale.

recomputing or refetching data which was not locally preserved. We show code examples of this natural redundancy in Section 4. The flexibility of CDs allows the application writer, compiler, or runtime system to balance the overhead of preserving state at each level of hierarchy at the cost of recomputing or refetching data which was not locally preserved. Optimizing this tradeoff is critical in order to avoid excessive preserved state which can waste valuable storage, bandwidth, and energy. These crucial tradeoffs are amenable to automatic optimization. Note that it is not necessary to preserve all data for a particular CD at the same location, level of hierarchy, or preservation method.



(a) Local Recovery

(b) Transfer from a Neighbor

(c) Transfer from a Parent

Figure 2.2: 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.

2.3 Error Detection

The first step toward resilience is to detect errors and failures. Ideally, the hardware and runtime system support the detection of errors and faults in memory and compute modules, which we discuss more in Section 3. In cases where silent data corruption is intolerable and the system does not provide complete error detection, the CD methodology enables application-based detectors and seamlessly integrates them with system detection techniques. The CD framework provides assertion-like abstractions for expressing application-level detectors, where a positive detection triggers CD recovery in the same way a system-level detector does. By uniformly treating all detection techniques, whether they be based on hardware, system software, generic duplication, or algorithm techniques, CDs can adapt and tune detection by mixing and matching techniques such as to simultaneous maximize detection effectiveness and efficiency on an application and system basis. This tuning, or mapping process, heavily relies on automatic tuning techniques, which we discuss further in Section 5.

Application-level detectors may be automatically inserted by the compiler or explicitly introduced by the programmer. In either case, the detection technique may be generic or algorithmic in nature. For generic techniques, the CD tree directly encapsulates the ideas of instruction or task duplication and can integrate them with the CD preservation and recovery mechanisms. Any CD can be duplicated for detection (and possible recovery) and this manipulation of the tree can be done automatically. Duplication can also be done by the compiler during code generation and integrated with CD preservation and recovery as a lower-level automatic and generic technique.

It is, however, desirable not to depend entirely on the generic techniques and to use algorithm and data structure knowledge as much as possible for detection. By utilizing such knowledge, it is possible to detect errors not covered by the system and for which generic duplication is costly. Algorithmic techniques can offer tremendous reductions in the overhead of error detection and correction and are thus invaluable when tuning protection. When the algorithm can inherently, or with CD recovery, tolerate errors, generic software and even some hardware mechanisms may be turned off to improve efficiency and sometimes also performance. Note that algorithmic knowledge may be captured in domain-specific languages and libraries and may, in some cases, be utilized even without intervention of the application programmer.

Some algorithmic detection and error-tolerance schemes require certain correctness and reliability guarantees for some portions of the code, while being able to relax typical requirements elsewhere. An example of this is fault-tolerant iterative methods [2], where the iteration control must be reliable while the underlying iteration itself may be (rarely) erroneous. With CDs, it is possible to tune the level of redundancy and reliability and to trade it off with resilience inherent to the algorithm. A simple example is to utilize duplication to selectively increase the reliability of susceptible CDs, while saving this extra overhead from inherently resilient CDs. To achieve this, the CD framework provides an interface for expressing flexible reliability guarantees. At a very coarse granularity, some CDs may be marked as *reliable* while others marked as *tolerant*. However, finer-grained annotations are necessary for effective tuning because the definitions of *reliable* and *tolerant* depend on the error rates and potentially types of allowed errors. Thus, at the finest granularity, *for each detector (each possible error/failure type) there is a rate below which the CD is tolerant to the errors/failures associated with that detector*.

Note that error detection is also required for communication and similar tradeoffs between precision/accuracy and overhead are possible. For example, the CD framework may log a checksum of all data sent/stored outside a CD and verify the correctness of values and destinations on re-execution, the application may provide an algorithmic test, or an application may choose to risk very rare errors in communication with reduced execution overhead.

2.4 Recovery

Recovery from soft errors requires re-executing a CD, which may include re-establishing its context after certain control failures. To recover from hard failures that disable a node (or group of nodes), the application must interact with the system to either request that new resources be mapped in (e.g., spare nodes) or to re-map its tasks onto a reduced-capability machine. The CD framework enables this application-level remapping. We anticipate, given the scale of future machines and the reliability trends of individual devices, that degraded operating will become necessary. While CDs allow remapping for graceful degradation without a change in paradigm, we have not yet explored the required interfaces in detail yet.

CD re-execution begins with restoring preserved state by reading it from the appropriate locations stated by the preservation routines (in a CD-local store or from a parent or sibling CD) or, instead, rematerializing needed data. While using the CD tree and preservation routines fully utilize the memory hierarchy and minimizes the average restoration latency, rematerialization 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 trade off the advantages of restoring the inputs of a faulty structure versus rematerializing those inputs from a reduced amount of state.

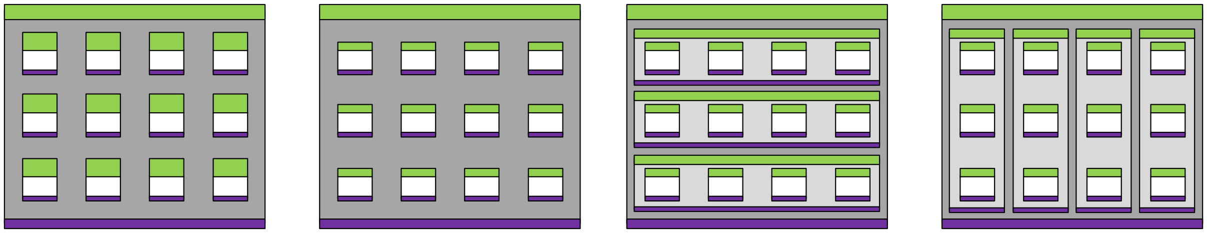
An important property of CDs is their weakly transactional nature, which enables CD recovery to be uncoordinated. Instead, recovery is initiated locally by a faulting CD and propagates up the hierarchy to the closest CD that can contain and recover the failure or error. This has several important advantages. First, no global barrier is imposed by the resilience scheme. Second, the recovery of multiple independent errors in different parts of the system can be overlapped, further improving scalability. These transactional semantics, however, disallow communication between concurrent CDs. This restriction may constrain the mapping of an application into the CD framework, and may reduce the advantages of CDs accordingly.

2.5 Communication

As mentioned earlier, CDs come in two flavors: strict and relaxed. Strict CDs do not communicate with any other CDs and all communication between tasks occurs within the same strict-CD context. This semantic restriction of strict CDs makes for a very scalable and implementation-friendly application characteristics. However, placing a restriction on communication introduce significant overhead in some cases. The overhead is a result of inherent communication within the application that places constraints on the CD tree. These constraints can be eliminated with relaxed CDs, offering more flexibility in optimizing resilience.

The two main tradeoffs that must be balanced, and for which flexibility is beneficial are the cost of preserving data vs. the cost of recovery from error or failure. Preservation costs generally increase when finer-grained CDs are tuned to tolerate many error/failure types while recovery costs increase when escalation is more frequent. As shown in Figure 2.3, when communication is frequent, there are three options for structuring the CD tree with strict CDs only and a fourth option with relaxed CDs. Each option offers a different tradeoff choice between preservation overhead, recovery cost, and imposed synchronization. Figure 2.4 illustrates recovery examples corresponding to the CD tree configurations of Figure 2.3.

Figure 2.3a shows the first option where the inner CDs can recover from many errors resulting in low overhead after errors. Achieving this, however, requires significant overhead for preservation. Figure 2.3b shows the second tradeoff option in which preservation for the inner CDs is minimal. Because the inner CD does little preservation, it is likely that more errors are escalated to the parent CD, resulting in high recovery overhead as all CDs



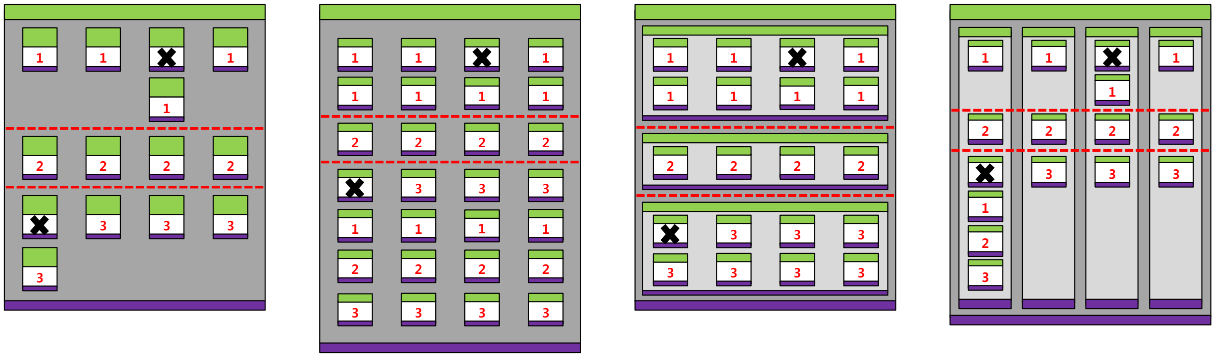
(a) Strict with inner recovery (b) Strict with inner escalation (c) Strict with intermediate level (d) With relaxed CDs

Figure 2.3: Example of CD configurations with communicating tasks. Visualization is of nested CDs with horizontal alignment indicating parallel tasks and vertically aligned tasks assumed to be serial. For simplicity, assume bulk synchronous parallel execution with all-to-all communication and barriers between each two groups of 3 horizontal CDs.

at the inner level must recover and reexecute on any escalation.

Figure 2.3c illustrates the third possible strict-CD configuration where an intermediate level is introduced. This intermediate CD level provides a new tradeoff where the inner-most CDs preserve little and escalate to the intermediate level. The intermediate level preserves more than the inner level and prevents escalation from the inner-most level from requiring all inner-CDs to recover. There are three problems with this third solution. First, all data transferred from one intermediate CD to the next must be preserved to prevent escalation. This potentially amounts to a large amount of data, which includes "communication" between serially-dependent tasks within a node. Second, all CDs within the intermediate level must re-execute, which typically would involve many nodes. Third, the CD completion semantics impose a barrier between each two intermediate CDs. While in many applications, this barrier exists because of the algorithmic communication, it may not be necessary in all cases.

Relaxed CDs offer a fourth possible configuration, shown in Figure 2.3d. With relaxed CDs, communication between tasks across CDs is possible. This allows a "vertical" intermediate CD level that encapsulates, for example, serial execution within a node. In this configuration, when an inner CD escalates an error, only the inner CDs within the vertical slice re-execute. This relaxed configuration strikes an intermediate tradeoff point. For the intermediate CD to correctly re-execute, the communication to tasks within it from tasks in other CDs must be preserved, or logged. Thus, a relaxed CD implementation must support the preservation of communication. By definition of CD communication semantics, all communicated data *is correct* with respect to detectable errors within the sending CD, enabling the uncoordinated recovery.



(a) Strict with inner recovery

(b) Strict with inner escalation

(c) Strict with intermediate level

(d) Relaxed CDs

Figure 2.4: Re-execution overhead examples: (a) inner CDs preserve significant data to enable just a single inner CD to re-execute after most errors; (b) with minimized preservation overhead, errors are frequently escalated to the outer CD, and hence, all inner CDs re-execute as the outer CD re-executes; (c) an intermediate CD level offers an intermediate preservation/recovery tradeoff compared to the first two examples; (d) each relaxed CD may re-execute by itself offering a different tradeoff between preservation and recovery where preservation where preservation is minimized and re-execution is confined to a single relaxed CD – a single re-execution takes similar extra time to example (b) but with much less re-execution and proportionally lower likelihood of additional errors.

2.6 Recovery Consistency Semantics

The containment domains model is amenable to, and intended for uncoordinated recovery between CDs. The most effective recovery mechanism in most cases is restoring of preserved state (by reading it from a CD's explicitly stored state, restoring from a parent or sibling CD or regenerating data) followed by CD reexecution or *replay*. In general, when a CD re-executes it may or may not execute in an identical fashion to its previous execution. What is necessary is that any application state observed by other CDs is consistent between executions of a CD. In other words, any application state that is a communication between CDs and in which a CD may affect the behavior of other CDs must be consistent regardless of whether a CD is executing for the first time or is reexecuting during recovery. For this discussion, synchronization operations and collective/atomics are treated similarly to data exchange.

Recall that by definition, strict CDs do not communicate with other CDs. Therefore a CD tree that is fully strict results in consistent and correct recovery even if execution is not deterministic and varies between the first execution of a CD and its reexecution for recovery. Recovery consistency behavior, however, is more complex with relaxed CDs and also differs somewhat between explicit two-sided message-passing communication and communication via a global address space.

Recovery Consistency with Relaxed CDs and Message Passing

With explicit message passing, communication between CDs is well defined[1][2]. Therefore, a *reexecuting CD is consistent when*:

1. All messages a CD attempts to send during reexecution are to consistent destinations and contain consistent data with its first execution; and

2. All messages the CD attempts to receive are from sources consistent with its first execution.

We define *consistent sources and destinations* as other sending and receiving tasks that are legal with respect to the application and reexecution. Strictly speaking, the most intuitive and simple interpretation of consistent sources and destinations are those that are identical to those in the first execution of a CD. However, many algorithms do not require a strict ordering of communication (at least between receives, see for example a discussion of *send-deterministic semantics* [3, 4, 5]). Moreover, some algorithms may still result in consistent execution and solution even if, during re-execution, a different source or destination is chosen.

Similarly, we define a *consistent communicated value* to be a value that is consistent with respect to the application. The common case is for the value to be identical, to within a small tolerance, to the value originally communicated, but some algorithms can tolerate greater imprecision than other algorithms.

A CD implementation provides an interface for the application to express how strict or relaxed its specification of source, destination, and value consistency is (see also Section 2.3). As an example, consider a strict CD implementation that logs all communication between CDs, including a checksum of all sent data. Every source, destination, and value is checked vs. the log and any mismatch results in escalation; escalation will continue until a CD is found that is guaranteed to result in consistent re-execution, i.e., the first strict CD that contains all CDs communicating with the CD that detected the potential error (see illustration in Figure 2.5). We define this group of CDs below the "least" common strict

[1] We do not consider the impact one CD may have on another via interactions through system control as communication because such effects are, by definition, consistent with respect to the application.

[2] If two relaxed CDs communicate via a shared file system, the communication is performed through a shared address space; it is treated similarly to communication through a global address space as described in Section 2.6.

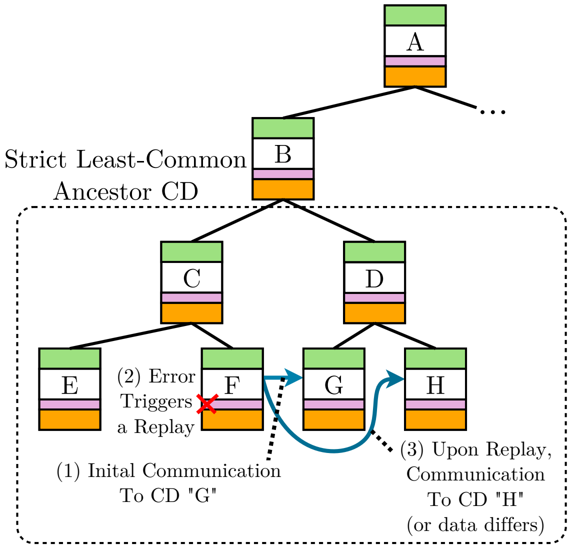

A
B
Strict Least-Common Ancestor CD
C
D
E
F
G
H
(2) Error Triggers a Replay
(1) Inital Communication To CD "G"
(3) Upon Replay, Communication To CD "H" (or data differs)

Figure 2.5: Illustration of potential reexecution inconsistency with relaxed CDs.

ancestor that may communicate with one another to be a *relaxed group* (CDs within the dashed box in Figure 2.5).

An implantation such as the one described above is consistent, but may escalate recovery frequently, if the checksum is computed on floating point values that may differ slightly from one execution to the next because of dynamic scheduling and finite floating-point precision.

Reordering and escalation can be reduced by logging all runtime events (e.g., logging and then replaying all application interactions with the software system) and computing checksums in a manner that accounts for bounded imprecision. The reason some imprecision is possible, even when logging and replaying system events, is that some dynamic decisions are taken without direct interaction with the application (e.g., out-of-order scheduling within the hardware). Note that any preserved communication and synchronization events may be discarded once execution again reaches an ancestor CD that contains all CDs with which communication exists in the log.

Recovery Consistency with Relaxed CDs in a Global Address Space

Recovery consistency is more complex when communication is unstructured and occurs through a global address space (GAS). This is the case when relaxed CDs communicate across PGAS contexts, in a global view language, via a hardware-implemented shared memory, or any other common namespace implementation. The reason is that with a GAS model, any read or write is potentially a communication because it may have been written or read, respectively, by a different CD. Preserving all loads and stores is prohibitively expensive, requiring somewhat more refined semantics and definitions as explained below.

We define the following terms:

Private

A read or write operation is *private* if it accesses data that will not be read or written by any other CD. Note that private addresses may not be used by any other CD until the entire relaxed group to which the CD belongs completes, or, may be reused provided any private address is written before being read again.

Temporarily privatized

A read or write operation is *temporarily privatized* if it accesses data that is: (1) private during the CD execution, (2) was not written by an earlier sibling CD within the relaxed group, and (3) which may be used by other CDs in the same relaxed group after the CD completes.

Shared

A read operation is *read shared* if it accesses data that is written by another CD within the relaxed group. A write operation is *write shared* if it writes data that is read by another CD within the relaxed group *and* the write is not temporarily privatized.

Potentially shared

A read or write operation is potentially shared if it cannot be classified as one of the other categories.

It is important to note that the classification above is logical with respect to the application. It does not matter wether a read or write is to a local or to a remote address because the semantics are defined by use rather than by mechanism. Using these definitions we can now better describe relaxed CD reexecution semantics in a global address space. Private reads and writes may be reexecuted without concerns for replay inconsistency; if private addresses were reused by other CDs, a new "shadow space" must be provided for re-execution. Temporarily privatized operations reexecute within a shadow space. This is necessary because other CDs within the relaxed group may or may not have already consumed the unprivatized values. Therefore, temporary values may be consumed by other CDs, resulting in inconsistent application state. Note that the last write by a CD may be squashed if it is not followed by another read in the CD.

Shared and potentially shared reads and writes are treated as communications and must be preserved in some way. There are two general options for preserving this communication. The first is to log (potentially) shared reads and replay the values on reexecution, while (potentially) shared writes are squashed. Writes may simply be squashed because their value has already been (potentially) communicated to other CDs and because reads from within the CD have been logged. The second general option is to provide some form of versioned memory for those addresses that are (potentially) shared.

Just as with explicit message passing, communicated values must be correct with respect to detectable errors. Also similar to message passing, it is possible to dynamically verify that re-executed writes are performed in a consistent manner, rather than simply squashing them, where the definition of consistent is as stated earlier.

2.7 CD Interface

This document focuses on the semantics of containment domains, required system support, and general operation, rather than on a specific implementation. However, to give a flavor of what interface a CD implementation provides, Table 2.1 lists the main abstractions as defined in the semantics above.

Table 2.1: Main CD Abstractions in pseudo-code API form.

`cd_begin([relaxed])`	Marks the starting point (and creates) a CD
`cd_complete()`	Completion point of a CD, where CD scope returns to its parent
`cd_preserve(data, [copy,parent,regen])`	Adds an item to be preserved in the CD store and directory, specifies all legal preservation methods
`cd_register([handler])`	Register a CD recovery handler routine for a specific system-reported error
`cd_assert(cond, [handler])`	Invokes CD recovery with generic or specific handler after application-level detection
`cd_handler()`	Recovery handler component of a CD (default is escalate or reexecution depending on contents of store and type of error
`cd_config(addr_check=0,value_check=1,...)`	Configure interaction with CD runtime, e.g., with respect to detection and reliability guarantees

3

System Support

Although we envision that the concept of containment domains can apply to any system, we describe our projection of a resilient large scale system for the future and discuss required system support to efficiently enable adoption of containment domains.

There is a general consensus that future large scale systems are expected to have deep hierarchies in every aspect of the system design. This includes the storage hierachy, hierarchical communication support, and hierarchical execution model. Containment domains are designed to exploit these hierarchies inherent to the system. For example, the amount of state preservation and restoration can be tuned based on the type and rate of errors that each level of storage hierarchy can handle. Also, hierarchical execution model and communication pattern help limit the amount of work that needs to rollback on an error.

To achieve the highest efficiency under given system properties, certain system support is required for each containment domain component. In this section, we discuss required system software and hardware support for preservation, detection, and recovery. Note that CDs are designed to make the application resilient and rely on the underlying system to provide some minimal reliability and resilience features for efficiency. These may not be strictly essential, but are necessary for any realistic implementation. We distinguish between essential and useful features in the text below.

3.1 System Support for Preservation

Storage Management

Managing storage for preservation requires similar system support to managing application storage. The CD runtime must be able to allocate and deallocate storage in memory and in the filesystem for preservation. It is also highly useful for a CD executing on one hardware resource to allocate and directly access storage on another resource (e.g., a memory or storage device on a remote node) in a one-sided manner (e.g., via RDMA operations). While not strictly necessary for implementing the CD model, such remote access is necessary for an effective CD implementation and in maintaining the "local action" principle of CD execution, where a CD is responsible for itself without coordination with independent CDs. Similarly, it is desirable to have an optimized path to manipulating local storage because the CD runtime is more aware of data movement patterns, liveness, and synchronization than the underlying node OS.

Another optional system feature that may help reduce the overhead of preservation is support for fine-grain memory protection to enable overlapped and asynchronous preservation and computation. Such fine-grain protection provides a mechanism by which the CD runtime can start preserving data while computation continues by marking data to be preserved as read-only until after it has been preserved.

Preserving Communication for Relaxed CDs

While strict CDs do not require communication preservation (Section 2.6), relaxed CDs require communication and synchronization preservation. Because CDs operate at the application level, all application communication can be preserved using CD-specific instrumentation without system support. However, the overhead of such an approach may be very high, in particular when communication is automatically handled by the system in a global address space.

Specifically, given the fine-grained and potentially frequent nature of communication events in a global address space (e.g., hardware-supported global address space and/or hardware cache coherence), hardware support for logging (potentially) shared reads and writes. The storage space into which the logs are stored must be, at least partially, managed by the CD runtime to balance the various trade offs. Hence, a particularly useful feature is for hardware to also log remotely by utilizing the existing communication mechanisms or by augmenting its network interface. Finally, if communication checksums are to be maintained and compared, it is useful for hardware to also provide support to reduce overhead.

During re-execution, some writes are squashed (and potentially checksummed), others are directed to a shadow space, and those that are private are simply re-executed. Similarly, some reads are played back from the log, others are directed to the shadow state, and the rest are re-execute from private addresses. It is possible for CD-enabled software to implement this behavior with no additional system support. However, a shadow space may be implemented with greater efficiency with system support for managing mappings without requiring software instrumentation or multiple software implementation variants.

Preserving System and Runtime State

The CD runtime is responsible for managing all application state and requires system support to establish and manage system context. System context includes access to storage resources (as with preservation), compute resources (i.e., processors and processes), network and communication-layer state, and other OS and system software components. The guiding principle is that any state, context, and resources that is not directly managed by the application and its runtime should either be stateless and made available as needed, or a means for preserving the component's state must be provided. An example is OS-level process state, which can be preserved by the CD runtime with appropriate support (as with BLCR [6]).

With respect to application state, the CD framework interacts with, or augments, other application runtime components to preserve necessary state for recovery. For example, to re-execute CDs in a deterministic fashion with high probability, the CD runtime may log the state of other stateful runtime components (e.g., the memory manager or random number generator) and library calls that return system-dependent values (e.g., calls that return the time of day). Of course, those values that are semantically stateless with respect to the application require no preservation (e.g., the runtime scheduler).

3.2 Integrated Error Detection and Error Reporting

One of the key features of containment domains is that they offer a single, consistent abstraction to express resilience concerns and to dictate hierarchical preservation and recovery. The CD framework does not expressely depend on any hardware support, but rather can exploit any system mechanisms that are available in order to raise the error coverage and lower the overheads of reliable execution. The ability of containment domains to evolutionarily add system error detection and error reporting mechanisms with a consistent abstraction is important for easing the adoption of CDs into a system context.

While containment domains do not rely on hardware support for functionality, system hardware is necessary to provide high error coverage with low overheads against the multiplicity of possible faults. Useful system features for CDs include mechanisms for reporting error events and faults to software with sufficient detail that software can efficiently map an error back to a particular CD. A supervisory framework is necessary to handle failures that prevent any application component from running, including the CD runtime itself.

In order to demonstrate the efficiency advantages of a system with full hardware support for CDs, a system error detection and reporting architecture is described below. These system reliability mechanisms should enable CDs to operate with optimal energy efficiency under a wide range of environmental conditions and operating scenarios.

Full hardware support for CDs relies on hardware-registered software-callback routines to enable low-overhead error exception handling. Each containment domain may specialize its error handling strategy by registering its own callback routines. Because software is encouraged to use hierarchical containment domains down to a fine granularity, each core must support a large number of concurrent CDs. The system, however, may not necessarily support a large number of registered callbacks. Hardware may strike a balance between efficiency and expressiveness by distinguishing error events that may be optimized by software from those that will almost certainly require significant recovery routines, and by distinguishing between errors in different hardware mechanisms.

Fast error recovery routines are used when hardware is able to precisely isolate the error to a single instruction (or fine-grained CD) and notify software before erroneous values are propagated. There could potentially be multiple fast routines registered to enable software to handle different error types appropriately. Slow routines are used for errors that can only be reported asynchronously or that cannot be isolated at a fine granularity. We describe the proposed error detection and reporting mechanisms categorized by error type below.

Note again that the descriptions below are of useful support for CDs, and that the CD concept may be implemented with minimal system support at greater execution overhead. This minimal support amounts to a supervisory framework for establishing catastrophic failures that prevent any CD runtime component from continuing to run. Also note that it is assumed that the CD infrastructure will log and propagate error and failure information for system analysis and optimization; the level of logging may be configured to tradeoff quantity and quality of information with logging overhead.

Memory and Data Transfer Errors

Full hardware support for low-cost and high-coverage reliability through containment domains relies on hardware error checking of all memories and communication channels. Note that it is possible to add software routines to check for such errors within the CD implementation or application, but we expect those to not be efficient in most cases; the exception is highly error-tolerant algorithms that require very few reliable data transfer and memory operations. Errors can occur on any data transfer or memory access, whether it is performed by an explicit load, store, or atomic-memory operation (AMO) instruction, an asynchronous direct memory access (DMA) operation, in sending an (active) message, or from an autonomous unit such as the memory scrubber.

Load Instructions. Unmasked errors that are associated with load instructions are isolated so that erroneous data is never propagated to a consuming instruction. An error may be detected at any level of the storage hierarchy, including the on-chip malleable memory and off-chip storage. Instead of attempting to immediately provide notification of the error when it occurs, the proposed hardware scheme for CDs returns *poisoned* data along the already-existing load path. Poisoned data indicates that there was an erroneous response somewhere along the load path, and can be economically implemented by purposefully encoding mismatched ECC and data information as data propagates up the hierarchy.

A load instruction returns poisoned data on any of the following conditions: an unmasked ECC error at any level of hierarchy, an unmasked error at any communication channel, and an interconnection network failure that results in a timeout. If a failure is detected, the thread that issued the load is notified when hardware attempts to write the poisoned load data back into a register. The load instruction writes the result back, immediately stalls the thread, and triggers the current registered CD callback. In this way, software can decide on the best way to handle the load error, which is typically one of the following: ignore the error, log the error and postpone recovery action, re-execute the load instruction (e.g., when the error was due to communication), engage a second-level ECC mechanism, or initiate full CD recovery. Data poisoning represents a straightforward error reporting mechanism that covers a large range of failures, while remaining exceptionally amenable to use by software.

Data poisoning is preferred over immediate notification of the error when it is detected because the latter introduces several problems and inefficiencies. First, the error notification signal must be sent to any load instruction that might access the same cache line, requiring notifications to be broadcast and received or buffered and sent multiple times. Second, notifications must utilize dedicated communication channels because this error reporting is out-of-band with respect to memory operations. Finally, additional synchronization is required to ensure that error notification is properly registered before any erroneous data reaches the pipeline. The need for this synchronization has been ignored by many processors until very recently; as a result, recovering from load errors on most architectures requires a reboot of the failing node.

When reporting an error, the CD handler may require the load address, information about the error, and potentially the erroneous loaded data. Because the report of failed loads is precise, all of the information required to reconstruct the load address is available within architectural state. Information about the error may include the level at which the error originated and, if known, the cause or type of the error. This information can potentially be communicated within the data being transferred, because the data is erroneous. However, this prevents software from accessing the loaded data, which may be useful for some error tolerant applications or those employing software (or second level) ECC. If software requires the erroneous data, it can issue a second request that indicates that data, rather than error status, should be returned. This enables the handler to first ascertain error status information and then to re-execute the load when appropriate to acquire erroneous data.

Store Instructions. Unlike load instructions, stores do not have a single, well defined completion time. A store instruction first "completes" when it leaves the pipeline, again if executing a memory fence operation, and then a store further completes as it slowly progresses through the storage hierarchy and is evicted from one cache to another or explicitly transferred by a DMA operation. There is a fundamental difference between reporting an error in a store instruction at or before a memory fence and one that occurs after that point. Once a store passes a fence it is hard to associate it with a specific thread, and hence, a fine-grained CD. We address this difference below.

Pre-Fence Store Errors. Errors in a store instruction before a memory fence will only be reported while executing the fence. Hardware registers information of the first occurrence of a store error as well as a count of how many store errors have occurred (this information is reset after a fence operation). When a fence operation is executed, hardware triggers the CD callback associated with this type of error (current view is that this type of error triggers the "slow" callback routine that is used for general error handling by the CD). Note that the fence operation is a per-thread, per-CD operation. A store error will typically trigger full CD recovery, but may be ignored in some rare situations.

Post-Fence Store Errors. Once a store passes a fence, its data is committed to the memory hierarchy and is no longer associated with a specific thread or CD. In fact, it is quite possible that the CD that generates store data completes before a store error occurs. A store error may occur at any level as the data is written back from one level of hierarchy to another (or discovered during scrubbing or when servicing a remote read request). We are still investigating the best approach for handling this type of error using containment domains and will present our nominal mechanism below. Note that this error is a significant challenge for any resilience approach and today will result in application failure or a full rollback.

Our initial design uses a dual approach, using two different error types, to protect against post-fence store errors. The first store error is reported when hardware detects corrupted data. At this point, hardware poisons the data and propagates the poisoned information to the next level in the memory hierarchy. At the same time, software is notified via a CD callback to allow software to proactively handle the error. The second error is reported when software attempts to read data that was corrupted as explained above. This second error enables software to ignore the initial error and react only when an attempt is made to use the data. The rationale is that the memory hierarchy often contains dirty data that is semantically dead (i.e., the data was written at some point, but this value is no longer required by the application). Whether handled proactively or reactively, corrupted data is regenerated using the specific mechanisms encoded by software using CDs. This will most often require re-execution of some CDs. Thus, the main tradeoff between the proactive and reactive approach is between unnecessary re-execution (when errors are reported for dead data) and overly coarse-grained recovery and re-execution (when errors are reported much later than when they are first detected). Which CDs must re-execute is a CD implementation decision based on the CD tree structure and potentially available information on the address ranges generated by each CD.

Other Memory Operations

Atomic memory operations are treated as stores or load-store pairs, depending on whether the operation is a simple remote update or a fetch-and-op type operation, respectively. DMA write operations are essentially stores; DMA read operations are handled as loads and generate poisoned data if any was read. Autonomous memory operations such as cache evictions and scrubbing have been discussed in the context of store operations above. Active messages are, for the most part, treated as store operations. An active message may fail before it is registered at its destination, which is equivalent to a pre-fence store failure. Active messages are more complicated than stores because the computation carried out by the active message may also fail. We discuss this aspect in more depth below when discussing resource errors.

Datapath Errors

While not common to many processors today, full hardware support for containment domains includes low-latency mechanisms to detect compute errors, such as concurrent arithmetic error detectors or configurable duplication. When an error in a datapath computation is detected, it is reported synchronously with the erroneous instruction. Similarly to a load instruction, an instruction that encounters a datapath error writes back the erroneous data and initiates the appropriate CD callback routine to handle the error before the result is propagated. The CD may register a null callback allowing computation to proceed as if the error was never detected to enable higher-level error detection and recovery schemes.

Control Errors

Hardware which controls the execution of threads and tasks may experience errors. When a control error occurs, such as following the wrong control flow path, instructions that should not be executed are introduced into the pipeline. When the error is detected there is little chance that it can be recovered quickly and the slow CD callback is activated. If the control error can be localized to a single CD, then the behavior is similar to that of erroneous pre-fence stores. If the error can, instead, only be localized to an execution resource at some level of the control hierarchy, then the error is handled similarly to a resource error as described below.

Hard Failures and Resource Errors

Due to gradual wearout, environmental conditions, or resource constraints, a system may experience the temporary or permanent loss of some memory or compute resources. It is important that the overall system be resilient to such faults and that the application continue executing correctly in their presence. To achieve this goal we rely on the containment domains design. While detection can be incorporated into the CD framework, it is also necessary to have a system supervisory framework for reporting some of the hard failures as they may prevent any application execution, including the CD runtime.

Memory failures are initially reported through the mechanisms described above. Repeated memory errors trigger diagnostics to identify the location of the failure and initiate appropriate action using the containment domains framework. Recovery may be tailored to application and domain specific needs, similar to the handling of transient errors. The loss of compute capability requires a different approach than a memory failure because the CDs associated with a compute resource may not, themselves, be able to identify the failure. If the compute failure is due to repeated datapath errors, the CD utilizing the resource can initiate resilience action. However, CD-local recovery is not feasible if the failure prevents instruction execution. In such cases, a supervisory framework (implemented in software, hardware, or a combination of both) monitors compute resources to failures using a combination of watchdog timers and hardware failure detection mechanisms. When a failure is identified, all affected CDs must be notified. Alternatively, only the least common ancestor of all affected CDs in the CD hierarchy may be notified (using the relevant registered callback routine). Identifying which CDs to notify is done similar to the isolation of CDs associated with post-fence store errors.

Communication Errors

Communication errors (in networks) are either detected by the underlying memory operations (which include messaging) or watchdog timers. If a portion of the network fails and is no longer usable than it either manifests as resource errors on the now inaccessible nodes, or if the network is resilient, as reduced capability. Both error types are identified by the supervisory framework. Unreachable nodes are reported similarly to failed nodes.

3.3 System Support for Recovery

In this section, we discuss system and runtime support that are necessary for correct and efficient recovery.

Recovery from Control Flow Errors

A CD may not reach the detection routine if it encounters a control flow error. In this case, other detection schemes (e.g. watchdog timer) will be used to detect the error. Recovery from this kind of control flow error or fatal failure is implemented by registering a callback function which directs the control to the appropriate recovery routine. This may entail a supervisory framework to establish a new context for the CD recovery callback to operate in.

Task Management

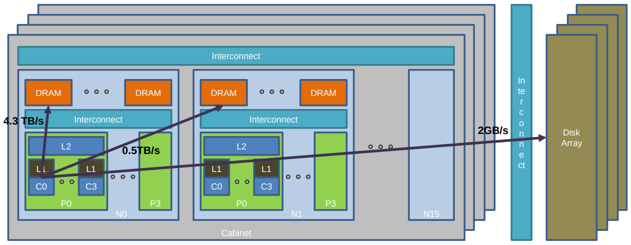
The CD runtime is responsible for managing all application state and requires system support to establish and manage system context. System context includes access to storage resources (as with preservation), compute resources (i.e., processors and processes), network and communication-layer consistent state, and other OS and system software components. These resources must be provided to the CD runtime for recovery by the supervisory framework or through the node OS. As mentioned in Section 3.1, the CD implementation interacts with the system for logging and recovering any application-visible state.

In addition to working with the system to reestablish all application context, the CD runtime also maintains the CD hierarchy so that it can easily bound the scope of recovery for a given error or failure event. When a child CD encounters an error that must be escalated to the parent, all the sibling CDs under the same parent may, most likely, also need to rollback. In this case, the CD runtime should instruct the system to interrupt all running siblings and restart execution from the parent. This can be implemented by allowing the CD runtime to kill any orphan thread and create new threads to rebuild the CD hierarchy. Note the CD correctness is maintained even without support for preemption, but efficiency may suffer as recovery overhead will increase.

4

Mapping Applications to Containment Domains

Understanding real world applications of CDs is important both for the enrichment of the model and to motivate its adoption. In this section, we explore several mini-applications that we manually map to CDs: a hierarchically-blocked iterative sparse matrix-vector multiplication, a Monte Carlo method neutron transport application, the HPCCG conjugate-gradient based linear system solver from the Mantevo mini-application suite [7], and the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) for hydrodynamic calculations [8]. While our analysis of these algorithms applicability to CDs is partially independent of the physical target, it is helpful to provide specific mapping examples in some of the more complicated applications. Therefore, our mappings specifically target the high-level architecture presented in Figure 4.1. This section will qualitatively discuss the mappings of these applications to our target system and provide intuition into some of the relevant CD design points. We will primarily use the SpMV application as our vehicle of exploration, due to its simplicity and heirachichal nature. Aftwards, we will explore some other applications and the challenges related to mapping them to CDs. A more-detailed quantitative analysis of HPCCG, SpMV, and Neutron Transport accross different system configurations are presented in our previous paper [1].



[Aggregate BW for 1024 nodes]
- L0 Local DRAM: 483*9GB/sec
- L1 Remote DRAM: 483GB/sec
- L3 Disk: 2.1GB/sec

Figure 4.1: An example system to ground the discussion of CD mappings. Based on the configuration of [9].

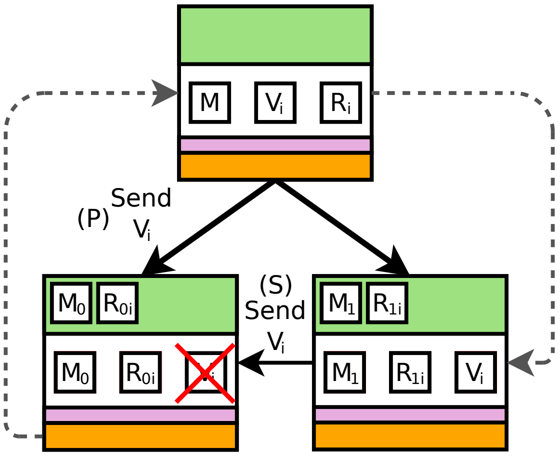


Figure 4.2: Examples of recovery using the natural redundancy of SpMV.

4.1 SPMV Detailed Overview

The SpMV computation consists of iteratively multiplying a constant matrix by an input vector. The resultant vector is then used as the input for the next iteration. We assume that the matrix and vector are block partitioned and assigned to multiple nodes and cores. This simple application demonstrates many of the features of CDs and how they can be used to express efficient resilience.

One of the advantages of containment domains is that preservation and recovery can be tailored to exploit natural redundancy within the machine. A CD does not need to fully preserve its inputs at the domain boundary; partial preservation may be utilized to increase efficiency if an input naturally resides in multiple locations. Examples for optimizing preserve/restore/recover routines include restoring data from sibling CDs or other nodes which already have a copy of the data for algorithmic reasons.

The SpMV program in Figure 4.3 exhibits natural redundancy which can be exploited through partial preservation and specialized recovery. The input vector is distributed in such a way that redundant copies of the vector are naturally distributed throughout the machine. This is because there are $N_0 \times N_0$ fine-grained sub-blocks of the matrix, but only N_0 sub-blocks in the vector. If a fault occurs that requires the recovery of an input vector for a CD, the vector is copied from another domain that holds it, as indicated in Figure 4.3 through the call to `add_to_CD_via_parent`. The CD which sends recovery data is determined by the parent of the faulting CD and is chosen to maximize locality, thus minimizing the bandwidth and power strain on limited resources. Figure 4.2 shows how the input vector of SpMV can be recovered from multiple sources (either directly from the parent or through a sibling CD), even if it was not preserved locally by the faulty leaf

```
void task<inner> SpMV(in M, in Vi, our Ri) {
  cd = cd_begin();
  cd_preserve(cd, M, [copy]);
  forall (...) reduce (...)
    SpMV(M[...], Vi[...], Ri[...]);
  cd_end(cd);
}

void task<leaf> SpMV(...) {
  cd = cd_begin();
  cd_preserve(cd, M, [copy, parent]);
  cd_preserve(cd, Vi, [copy, parent, sibling]);
  for r = 0..N
    for c = M.rowS[r]..M.rowS[r+1] {
      Ri[r] += M.col[c]*Vi[M.idx[c]];
      cd_assert(c > prev_c);
      prev_c = c;
  }
  cd_end(cd);
}
```

Figure 4.3: Sequoia-style pseudocode for SpMV with CD API calls.

CD. Such partial preservation tradeoffs cannot be easily expressed or exploited by prior resilience models.

The inner CDs in the hierarchy specify that the input matrix should be preserved explicitly, ideally in non-volatile memory (through the `add_to_CD_via_copy` call in Figure 4.3). Each node has its own unique portion of the matrix, and thus the matrix can only be restored from explicitly preserved state. If the matrix is *not* preserved, data corruption may be unrecoverable, except by rolling back to the root CD at great cost. When mapping to the example system (Figure 4.1), every CD responsible for protecting against failures in DRAM, nodes, and cabinets preserves the matrix "locally" to their domain (e.g., using a buddy approach), and can thus recover locally as well. CDs responsible for protection within a processor, on the other hand, may or may not preserve the matrix depending on the expected soft-error rate. If the matrix is not preserved, it will be restored by re-fetching the data from the parent. Such flexible state preservation optimizations have been proposed in the past and have also been expressed as automatic analysis algorithms [10, 11]. CDs enable such automation without changing the underlying resilience paradigm while still providing the programmer with explicit control when needed.

Figure 4.3 provides high-level pseudo-code for SpMV. The program structure and syntax are inspired by the Sequoia programming model [12] with explicit CD API calls [13] and target the hierarchical machine shown in Figure 4.1. The root-level task performs the iterative computation by hierarchically decomposing the matrix multiplication. Hierarchy is formed through a divide and conquer decomposition 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 encapsulated by a CD with its own preservation and recovery routines, which we will explain in the subsections below.

The hierarchy of the CD tree is created by nesting domains using the `parentCD` handle; the full management of these CD handles is not shown for brevity. The syntax given in Figure 4.3 is for illustrative purposes and is based on an initial research prototype of CDs that is currently under development.

4.2 Other Application Mappings

Neutron Transport (NT): The study of neutron movement and interaction is one of the earliest problems examined by computer systems, and remains an important high-performance application. A Monte Carlo approach to the simulation of neutron transport is desirable because the creation and simulation of every particle is independent, making the problem embarrassingly parallel. The major source of communication takes place in a parallel reduction to aggregate particle properties over the whole system.

CDs can take advantage of the stochastic nature of the Monte Carlo approach – the only state that needs to be preserved (and cannot be quickly rematerialized) is a global tally of particle densities and directions. Erroneous particles and failed CDs may be discarded without consequence, such that particle-local data does not need to be preserved or recovered. An overview of the CD mapping to neutron transport is shown in Figure 4.4.

HPCCG: HPCCG is a linear system solver that uses the conjugate gradient method and can scale to a large number of nodes [7]. This mini-application uses a simple and standard sparse implementation with MPI. The application evenly partitions rows across cores (leaf CDs) with each core responsible entirely for its partition.

This application is similar to SpMV, but our SpMV implementation is hierarchically nested and HPCCG distributes entire rows. In each iteration, each node computes a residual error value based on the previous value and the head node reduces these values into one residual value at the end of each iteration. If this value falls within an acceptable error range, then computation terminates. Otherwise, all nodes exchange local row data with their immediate neighbor and compute another set of residual values.

We scale the size of the input matrix linearly with the number of cores and assume weak scaling. Because each node communicates with its neighboring nodes at the end of each computation, every node is dependent on every other node and, in effect, a soft barrier is necessary before a new iteration can start. Thus, with strict CDs, data is preserved at the beginning of each iteration. Since it is communicating frequently, similar to SpMV, the duration of a leaf CD is limited by the soft barrier. HPCCG takes mostly read only data as input, thus, it is mapped to preserve those data to the parent level CD, and the leaves preserve the state that is essential for re-execution.

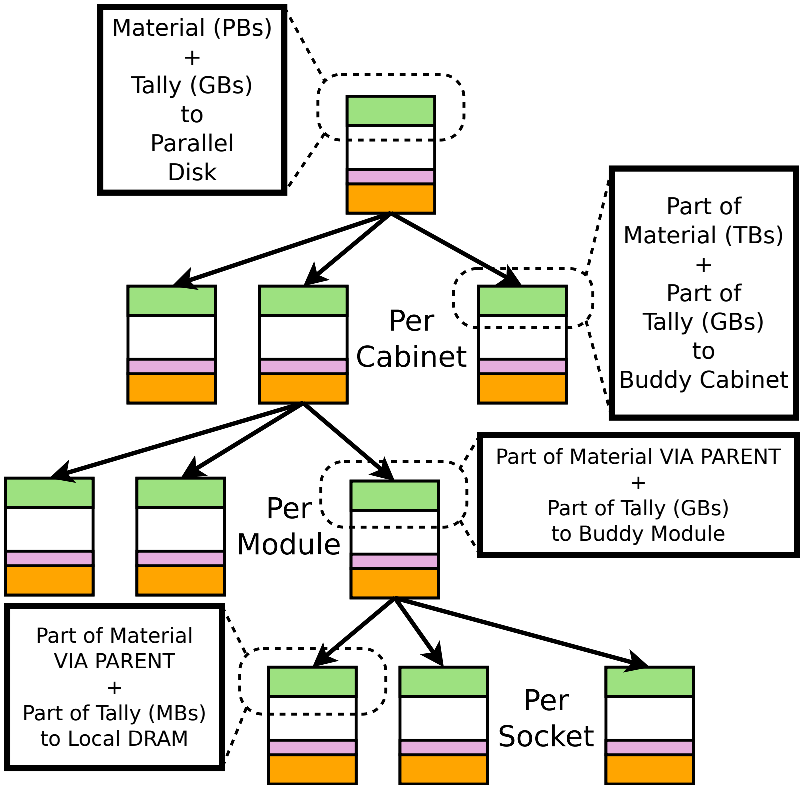


Figure 4.4: Example mapping of nuetron transport to CDs for our target architecture.

LULESH:

The Livermore Unstructured Legrange Explicit Shock Hyrdrodynamics proxy app was developed as one of the five challenge problems in the DARPA Ubiquitous High Performance Computing(UHPC) program. It is designed to mirror the full-scale unstructured mesh calculations that run on large supercomputing clusters. Since its creation, LULESH has become a popular research application due to its small size (5000 lines of code for the MPI + OpenMP variant) and implementation in a variety of languages.

Essentially, LULESH models a region of physical space using `domains`, consisting of thousands of `nodes` and `elements`, arranged in an irregular mesh, where each element is surrounded by a number of nodes. In each domain, the nodes represent the kinematics information of the region, while the elements represent the thermodynamics. At the beginning of program execution, LULESH sets the initial value for each domain, and assigns one domain to one rank. During the execution, each domain works in parallel, and also communicates with the neighboring domains through MPI. LULESH also injects a single point of initial energy into one of the corners of the mesh, setting up what is known as the Sedov blast wave problem.

For the whole simulation, the collection of domains run for thousands of time steps, in which a Lagrange leapfrog algorithm is followed by a global time step reduction. Essentially, the kinematics quantities (nodes) in each domain are updated using the information from the previous time quanta. The results are then shared to each of up to 8 neighbor domains that will require this updated information for their next time step. Similarly, the thermodynamics properties (elements) are updated and communicated to the appropriate neighbors. Finally, a new time step quanta is calculated by calculating local time step constraints in each region and performing a global reduction. These three computational and communication phases continue until the desired runtime or ending conditions are met.

LULESH is complicated to map to the CD framework mainly for its frequent communication with the neighboring domains. In the LULESH CD tree, each leaf CD executes one computation phase in the Lagrange Leapfrog function and a barrier exists between two phases for communication. The upper level CDs are mapped to our target architecture similarly to the previously discussed applications, where the higher level CD computes a larger number of domains. Notably, except for the leaf CDs, each upper level CD needs to communicate with its neighboring CDs. Therefore, for LULESH, there are some interesting observations that were not present in the previous applications:

1. **Relaxed CDs:**

   As we describe above, the current version of LULESH has a minimum of three distinct phases of communication for each time step. Therefore, it is preferable to consider using relaxed CDs to offer any explicit benefits. In LULESH, relaxed CDs allow for a more natural mapping of the application as well as a reduction in the storage overhead required by CDs closer to the leaves. The leaf CDs are strict CDs because they do not need to communicate with one another. The outer CDs are relaxed to interact with neighboring ones to update the information.

2. **Halo/Ghost Regions:** Relaxed CDs are helpful because of the frequent neighbor communication in LULESH. This communication occurs between timesteps and therefore requires frequent synchronization and logging as well. Because of the nature of the CD tree, this synchronization is effectively a global barrier. To reduce the frequency of synchronization and logging, each domain can be extended with *halo* or *ghost* nodes and edges. These nodes and edges are not part of the responsibility of the domain, but are included to allow multiple timesteps to be computed before another communication with neighboring domains is required. The overall volume of communication (and logging) is reduced if the halo-to-domain ratio is small, but additional computation (on the halo cells) is necessary. Balancing the potential loss

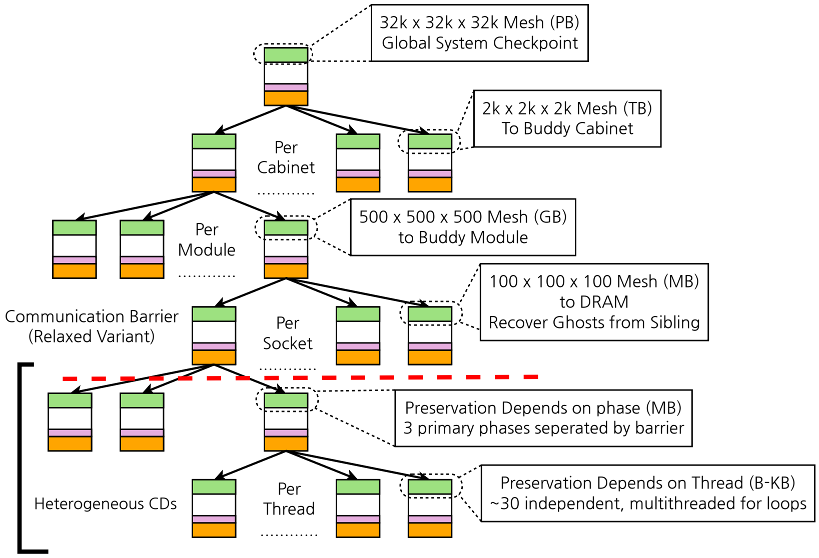

32k x 32k x 32k Mesh (PB)
Global System Checkpoint
2k x 2k x 2k Mesh (TB)
To Buddy Cabinet
Per
Cabinet
500 x 500 x 500 Mesh (GB)
to Buddy Module
Per
Module
100 x 100 x 100 Mesh (MB)
to DRAM
Recover Ghosts from Sibling
Communication Barrier
(Relaxed Variant)
Per
Socket
Preservation Depends on phase (MB)
3 primary phases seperated by barrier
Preservation Depends on Thread (B-KB)
~30 independent, multithreaded for loops
Heterogeneous CDs
Per
Thread

Figure 4.5: Example mapping of CDs for LULESH on our target architecture.

of performance and efficiency to extra communication with the benefits of reduced synchronization and communication requires careful tuning.

5

Analyzing and Auto-Tuning CDs

In this section, we present analytic models of CDs and show how to apply them to some cases with respect to the main tradeoffs. CDs are designed to be tuned and optimized automatically. As part of the tuning process, it is necessary to estimate the expected performance quickly enough to evaluate numerous auto-tuning points that represent the behavior of extreme-scale systems. Common auto-tuning techniques of repeatedly running the application with different configurations to collect information on execution characteristic do not work well for evaluating many aspects of CD optimization because of the large time scales required for resilience evaluation. Therefore, we use a set of models to evaluate different CD tree configurations and parameters.

The goal of the models is to not only enable an evaluation of a specific CD configuration, but also to allow the automatic tuning of the many parameters that can be specified within the CD tree. These parameters include the depth of the tree, the granularity of each CD, whether a particular level should be relaxed or strict, choosing preservation location and method, etc. A full discussion of the analytical formulas and the autotuning algorithms is beyond the scope of this document and we only provide the main assumptions and current limitations below.

5.1 Basic Assumptions

The CD analytic models require information about the application and system. We enumerate this basic information and other model assumptions below. As we explain later, the nested semantics of CDs enable a bottom-up modeling approach where time and resources are propagated up through the hierarchy.

1. Information about the properties of each CD, parallelism, synchronization, and communications between CDs is given by the description of the CD tree; communication and synchronization information may be simplified using nested bulk parallelism.

2. The time required for preservation and restoration of each CD as well as the time required for execution of leaf CDs (we assume any significant execution is modeled as a leaf in the tree) are inputs to the model. Preservation and restoration parameters may be derived through experiments and models relating volume of preservation to overhead, but are not a part of the analytic CD analysis described below.

3. Errors and failures follow a memory-less random process with a fixed rate per error/failure type; thus, the probability of a CD failing is proportional to its execution time. The error and failure processes are provided (e.g., soft errors are uniformly distributed in time with a fixed rate).

4. For simplicity, we treat all errors and failures in one of two ways: (1) the error/failure can be tolerated by the CD through re-execution recovery, or (2) the error/failure is escalated to the parent CD; the rates of errors/failures that can be elided by the CD are zeroed out for modeling.

5. At the machine level, we assume that there are sufficient spare nodes and cabinets to recover from errors and that those spares can be brought on line quickly without slowing down the recovery process.

6. We assume zero overlap between execution and recovery; the extra time required for recovery is added to the time of the faulting CD. Recovery of different siblings, however, can proceed in an uncoordinated fashion in the absence of a synchronization or blocking communication.

With these assumptions we have developed initial models for the execution time, energy, and storage overheads of a CD tree. However, these models are not fully generic. Specifically, we have so far developed a model for strict CDs and an initial model for relaxed CDs. Both models require the CD tree to be symmetric, that is, each CD of a particular level is equivalent to all its siblings. In addition, our model for relaxed CDs currently treats all communication conservatively as hierarchical bulk-synchronous; even though the application may have more sparse synchronization and communication in practice.

6 Related Work

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.

6.1 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. [14] 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 [15, 16].

Panda et al. describes a recovery scheme consisting of local and global mechanism [16]. Every *k*th checkpoint is saved on the global storage, while other points on its local disk. If a transient failure occurs, the processor will roll back to the most recent local or global checkpoint; if there is a permanent failure or the local disk has a failure, the processor must resort to the recent global checkpoint. This two-level checkpointing scheme recovers diverse failures with different overhead.

For coordinated checkpointing, there exist two concepts of checkpointing: blocking and non-blocking. Blocking represents stopping MPI computation time to save the global checkpoints, while non-blocking can store the state during the MPI execution time. It is obvious that blocking can offer a better control of the state preservation and communication process, since no message exchange occurs during this process. In contrast, non-blocking reduces the overall execution latency with a higher algorithmic complexity. According to their characteristics, Franck [17] provided the first comparison of such two global checkpointing on three various computing platforms: grids, clusters of workstations, and high speed networks. It is shown that the checkpoint frequency affect the performance of blocking mechanism in high speed networks, while non-blocking demands more overheads for its high synchronization cost for clusters and grids.

Containment Domain also needs to store the state information to provide the rollback recovery, and the preservation timing overhead occupies a large fraction of the total overheads. Therefore, to alleviate this cost, we will discuss the asynchronous preservation in the later section, which can save the data during computation, similar to the non-blocking mechanism. Nevertheless, because of the nested structure and weak communication of CDs, we dont need to pay so high control overheads for data synchronization. Additionally, the amount of data saved in each CD is not related with the frequency, just depending on the specific application. Therefore, CD can decrease the required preservation time with relatively fewer modifications.

Coordinated checkpointing based on disks causes IO bottleneck when writing the checkpoints to disks. To avoid this, diskless checkpointing is introduced. With the help of extra resources and encoding techniques, diskless mechanism fulfills the checkpointing.

Chiueh et al. propose an alternative diskless checkpointing mechanism [18] 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. Gomez et al. presented a new encoding algorithm, i.e. weighted checksum encoding, to tolerate the faults [19]. To decrease the spare resources, the computing processors are also used as the encoding processors. In addition, the checkpoints are distributed into different groups in order to enhance the robustness. Thus, if one processor fails, the checkpoints saved in other processors can help to recover the original data.

Dong et al. suggest phase-change memory (PCRAM) for fast local/global checkpointing [20]. 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 [9]. Multi-level checkpointing employs multiple types of checkpoints with different costs and different levels of resiliency in a single run. For large-scale systems, Moody et al. designed a Scalable Checkpoint/Restart (SCR) library [21]. Since a job only needs its recent checkpoint and a typical failure only affects a small portion of the system, the library stores the checkpoint local to the compute nodes. If a failure occurs, the system restarts the job from the local cache. If the failure cannot be recovered from the cache, the library fetches the checkpoint from the higher level system.

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* [22]—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 [23, 24] and coordinated checkpointing based on communication history [25, 26]. These mechanisms, however, often sacrifice a certain degree of process autonomy and incur run-time and extra message overheads [27].

In terms of the log-based methods, a transparent rollback-recovery protocol, named *Manetho*, is used to deal with distributed applications [23]. *Manetho* takes the advantage of the existing transparent methods, with limited rollback processes, asynchronous logging of recovery information, and less latency of committing outputs. However, this mechanism may cause a relatively complex recovery system for applications with high error-rates. In this mechanism, an Antecedence Graph, i.e. AG, is introduced to describe the state intervals and their communication relationships. Also, it is assumed that the application is composed of a set of recovery units which contain multiple threads. During the operation, the recovery units keep the AG of the current state and log the data and identifier of its sending message in volatile memory. The AG of this state is updated based on its previous AG and the AG of the received messages. Besides saving information in volatile storage, recovery units also take checkpoints periodically in stable storage and asynchronously with other units. To avoid the *domino effect* during the recovery process, only the recovering unit needs to be rolled back to its latest checkpoint while other units still keep their states. This limited rollback is similar with Containment Domains for the failures can be localized in a certain domain. However, *Manetho* does not have the escalation recovery strategy, so it can only deal with the applications with infrequent errors. Additionally, *Manetho* causes a relatively complicated recovery system to recover the failed process. As for Containment Domains, since every domain preserves the data at the beginning and has its corresponding recovery methods, it is easy to recover the failed domain from the preserved data with low overheads.

Although message logging can solve the *domino effect*, it causes a large expense of logging data, finally consuming lots of storage space and communication bandwidths. To eliminate the *domino effect* with fewer costs, Franck[28] proposes a protocol targeted at MPI send-deterministic applications, which only logs a portion of the messages and reduces the number of processes to rollback. In a send-deterministic application, the sequence of sending messages always keeps the same, so each orphan message will be sent again during recovery for the delivery order is already ensured. However, when re-executing the failed process, the processes that send messages to it before the failure also need to rollback, thus causing another *domino effect*. To avoid this condition, the conception of *epoch* is given, whose value increases when taking each checkpoint. With *epoch*, only the messages sent from the last *epoch* and received by the next *epoch* are logged. Additionally, the conception of *phase* is also proposed to make sure the causal dependencies. The value of *phase* increments with the number of checkpoints, and is also determined by the *phase* of received messages. Therefore, with these two notions, the causal relationships between the messages are quite clear such that no messages will be neglected or re-sent with incorrect sequences. With the cost of logging a subset of messages, the *domino effect* can be avoided efficiently and only a few processes need re-execution.

The principle of this protocol is quite similar with Containment Domains, both of which save only a part of information and rollback a portion of processes in an uncoordinated manner. Although this protocol reduces the cost of logging data, it is required to pay attention to the causal dependencies of each message to make sure the correctness during replaying. For Containment Domains, since its overall structure is nested with weak communication semantics, it is easier to recover the failed domain with less control of replaying sending messages. In addition, if multiple independent errors occur in various parts of the application, Containment Domain can overlap the recovery process, while the above protocol needs more processes to rollback, thus leading to a larger recovery cost.

6.2 Software Interface to Preserved State

Randell proposed a software fault-tolerant programming interface, more related with sequential systems, *Recovery Blocks* [22]. 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 6.1 shows an example of a Recovery Block which might be used to protect a sort routine. Since the state is saved on entry to the Recovery Blocks, an outer (higher level) Recovery Block can resolve the error from an inner (lower level) Recovery Block. Thus, the programmer can design hierarchical Recovery Blocks with backward recovery. Nowadays, many applications has combined Recovery Blocks with other techniques, such as N-version programming, or modified recovery blocks as retry blocks, using data diversity [29].

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. However, Recovery Blocks mainly depends on AT to check the correctness, which must be designed very carefully. 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 6.1: 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 [30], 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 mechanism can be used for distributed data objects, where one object can cause the usage of other objects. Beeri et al. developed such a computation model for nested transactions [31]. In this work, transactions include sub-transactions, finally forming a nested tree structure, suitable for multi-level database systems. This style of programming, including [32, 33, 34], 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 [35]. Spheres of Control are nested and transactional in nature, and can rollback any errors which occur before a transaction has been committed. The commitment is controlled dynamically by constraining the potential erroneous process to avoid releasing the output until the repeated processing is unnecessary. 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 [36]. 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 [37, 34]. 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.

Xu et al. designed a software framework for fault tolerance in a concurrent object oriented environment [38]. This framework is implemented based on two complementary concepts: *conversions* and *transactions*. *Conversation*, containing communicating processes, supports the backward recovery surrounded by two lines. The recovery line is a set of coordinated states for backward recovery; additionally, the test line is served as a set of acceptance tests. *Transactions* mainly operate on shared data and objects to deal with hardware-related failures, which is a nested structure concerned with only one process. There exist some similarities between this framework and Containment Domains: this system also has weak transactions for *conversation* only allows the inner processes interacting with each other; all the processes are tested at the end of *conversation* to make sure the correctness; this system also supports backward error recovery when the error occurs, and can return back to the recovery line to restart. Unlike Containment Domains, this framework supports coordinated recovery for all the processes in a *conversation*. Even if one process fails, all the processes must return to recover from the recovery line. In addition, the recovery line does not exist necessarily for this system also supports forward recovery. In the forward recovery, actions are performed to correct the errors without returning backward. At that time, the recovery line can be removed. Nevertheless, in Containment Domains, we must preserve the useful data at the beginning of each domain because we only accept backward recovery. In addition, Containment Domains can provide local recovery: if one CD has a soft error, then only this CD need be re-executed without affecting other neighbouring CDs.

Wang et al. take advantage of the log-based rollback strategy to tolerate software errors [39]. When a transient error occurs, the system executes a local recovery and replays the message logs to recover the system state. However, if the above approach does not work, the system state is reconstructed and the messages are reordered to bypass the error, while increasing the non-deterministic degree. Additionally, the scope of rolling back can be adjusted to tolerate the software errors, but the larger scope causes higher non-determinism. Like Containment Domains, this mechanism also offers several recovery methods to deal with different errors to reduce the overhead. The concept of scope is similar with domains, which give the boundary of error recovery. However, Containment Domains are nested programming constructs that can be mapped to memory hierarchies, while this mechanism mainly relies on the timing boundary. Furthermore, the non-determinism in this mechanism increases algorithmic complexity.

For soft errors in iterative computation to solve linear equations, Hoemmen[40] provides a collaboration strategy with programmers to report faults and take selective checkpoints. In this method, the author assumes that the reliability in each part of the application is variable, where some computations are quite reliable while others are not. Since only a small portion of code is susceptible to faults, this method introduces fault-oriented introspection for this portion, to reduce the impact of soft faults. The method mainly focuses on preconditioned iterative approach, i.e. inner-outer iteration approach, where the inner iteration preconditions the outer iteration. In general, for a specific algorithm, instead of reporting errors and restarting, the modified fault-tolerant algorithm tries the listed recovery strategies, and the solution space grows with each outer iteration, offering better fault tolerance. Similar to this method, Containment Domain also has local recovery mechanism, and can also deal with various types of faults. Additionally, Containment Domains has the flexible recovery mechanism with a light burden of error detection and correction. In contrast to this method, Containment Domain can recover from hard failures by re-mapping tasks to new resources.

Recent research on transactional memory (TM) [41, 42, 43] 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 [44, 45]; 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 [46] and FaulTM [47]. 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 [46] uses try/catch like semantics to provide reliability through 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 [47] 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. 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 6.2: The syntaxes that RELAX and FaulTM use to provide reliability.

Bibliography

[1] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 1–11. 5, 30

[2] M. Hoemmen and M. Heroux, "Fault-tolerant iterative methods via selective reliability," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2011. 11

[3] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel hpc applications," in *Proceeding of International Conference on Computer Communications and Networks (ICCCN)*, 2010. 16

[4] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *Proceeding of IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2011, pp. 989–1000. 16

[5] A. Guermouche, T. Ropars, M. Snir, and F. Cappello, "Hydee: Failure containment without event logging for large scale send-deterministic mpi applications," in *Proceeding of IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2012, pp. 1216–1227. 16

[6] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-54941, 2002. 22

[7] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich, "Improving performance via mini-

applications," Sandia National Laboratory, Tech. Rep. SAND2009-5574, 2009. 30, 33

[8] "Hydrodynamics challenge problem lawrence livermore national laboratory," Tech. Rep. LLNL-TR-490254. 30

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. 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 30, 41

[10] A. Oliner, L. Rudolph, and R. Sahoo, "Cooperative checkpointing theory," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, 2006. 32

[11] G. Bronevetsky, D. J. Marques, K. K. Pingali, S. McKee, and R. Rugina, "CoMPIler-enhanced incremental checkpointing for OpenMP applications," in *Proceedings of the International Symposium on Principles and Practive of Parallel Programming (PPoPP)*, 2008. 32

[12] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. New York, NY, USA: ACM, 2006, p. 83. 32

[13] Cray Inc., "Containment domains API," lph.ece.utexas.edu/public/CDs, April 2012. 32

[14] 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. 39

[15] 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. 40

[16] 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. 40

[17] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault

tolerant MPI," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2006, p. 127. 40

[18] T.-C. Chiueh and P. Deng, "Evaluation of checkpoint mechanisms for massively parallel machines," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1996. 40

[19] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 63–72. 41

[20] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2009. 41

[21] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2010, pp. 1–11. 41

[22] 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. 42, 43

[23] E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 526 –531, May 1992. 42

[24] B. H. L. Alvisi and K. Marzullo, "Nonblocking and orpha-free message logging protocols," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1993. 42

[25] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, 1991. 42

[26] 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. 42

[27] Y.-M. Wang and W. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation," in *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, Oct. 1993, pp. 78 –85. 42

[28] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, may 2011, pp. 989 –1000. 42

[29] B. Randell and J. Xu, "The evolution of the recovery block concept," 1995. 44

[30] 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. 44

[31] C. Beeri, P. A. Bernstein, and N. Goodman, "A model for concurrency in nested transactions systems," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 230–269, 1989. 44

[32] D. P. Reed, "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, MIT Laboratory for Computer Science, 1978. 45

[33] J. E. B. Moss, "Nested transactions: An approach to reliable distributed computing," Ph.D. dissertation, MIT Laboratory for Computer Science, 1981. 45

[34] 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. 45

[35] C. T. Davies, Jr., "Data processing spheres of control," *IBM Systems Journal*, vol. 17, no. 2, pp. 179 –198, 1978. 45

[36] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*, 1st ed. Morgan Kaufmann, September 1992. 45

[37] B. Liskov, "Distributed programming in argus," *Commun. ACM*, vol. 31, pp. 300–312, March 1988. 45

[38] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, vol. 0, p. 0499, 1995. 45

[39] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Proceedings of the Symposium on Fault-Tolerant Computing (FTCS)*, 1993. 46

[40] M. Hoemmen and M. A. Heroux, "Fault-tolerant iterative methods via selective reliability," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011. 46

[41] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 289–300. 47

[42] 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 International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Jun 2004, p. 102. 47

[43] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2006, pp. 254–265. 47

[44] 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. 47

[45] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006. 47

[46] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010. 47

[47] 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. 47, 48