Scientific Programming 21 (2013) 197–212
DOI 10.3233/SPR-130374
IOS Press

Containment domains: A scalable, efficient and flexible resilience scheme for exascale systems [1]

Jinsuk Chung [a,*], Ikhwan Lee [a], Michael Sullivan [a], Jee Ho Ryoo [a], Dong Wan Kim [a], Doe Hyun Yoon [b], Larry Kaplan [c] and Mattan Erez [a]

[a] *The University of Texas at Austin, Austin, TX, USA*
[b] *IBM, Yorktown Heights, NY, USA*
[c] *Cray Inc., Seattle, WA, USA*

Abstract. This paper describes and evaluates a scalable and efficient resilience scheme based on the concept of *containment domains*. Containment domains are a programming construct that enable applications to express resilience needs and to interact with the system to tune and specialize error detection, state preservation and restoration, and recovery schemes. Containment domains have weak transactional semantics and are nested to take advantage of the machine and application hierarchies and to enable hierarchical state preservation, restoration and recovery. We evaluate the scalability and efficiency of containment domains using generalized trace-driven simulation and analytical analysis and show that containment domains are superior to both checkpoint restart and redundant execution approaches.

Keywords: Exascale, resilience, flexible reliability, fault-tolerance

1. Introduction

Reliability and resilience are major obstacles on the road to exascale computing. The growing 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. Current reliability and resilience techniques simply cannot cope with the simultaneous increase in scale and fault rates while maintaining high efficiency.

We present *containment domains* (CDs), a new approach for achieving low-overhead resilient and scalable execution. CDs abandon the prevailing one-size-fits-all approach to resilience and instead embrace the diversity of application needs, resilience mechanisms, and the deep hierarchies expected in exascale hardware and software. CDs 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 are programming constructs for incorporating these insights within a programming model and system. CDs have weak transactional semantics and are designed to be nested to form a CD hierarchy. The core semantic of a CD is that all data generated within the CD must be checked for correctness before being communicated outside of the domain

[1] This paper received a nomination for the Best Paper Award at the SC2012 conference and is published here with permission from IEEE.

* Corresponding author. E-mail: chungdna@gmail.com.

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. A specific error may be too rare, costly, or unimportant to handle at a fine granularity. For this reason, an inner 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.

CDs enable a new range of resilience tradeoffs and can significantly improve application efficiency and performance relative to prior reliability approaches. We demonstrate the potential benefits of CDs by simulating synthetic application models with error injection. The application models are based on a set of mini-applications that we map to CDs and span a range of characteristics that impact CD effectiveness. We compare CDs to previously proposed techniques across a range of machine scales and error models. We show how CDs flexibly utilize the five insights described above to improve resilience overheads and provide superior scalability. To summarize the key aspects of the CD approach are:

(1) Hierarchical preservation and restoration enables localized handling; errors that can be recovered locally are handled by the innermost CD in a cost-effective manner.
(2) Specialized and hierarchical state preservation effectively utilizes the storage hierarchy and reduces preservation volume, overcoming the low bandwidth and high error rates associated with system scaling.
(3) The transactional semantics of CDs improve scalability, and uncoordinated recovery enables applications to tolerate high error rates on very large systems. While prior approaches drop below 50% efficiency for large systems with high error rates, CDs remain efficient.
(4) The structure imposed by the CD abstraction expresses resilience schemes in a way that is both general and amenable to automatic analysis and tuning.

The rest of the paper is organized as follows: we compare and contrast CDs with prior work in Section 2, describe the CD framework and its use in Section 3, develop a flexible analytical model for CDs and uncoordinated recovery in Section 4, summarize our evaluation methodology in Section 5, present the evaluation results and discuss their implications in Section 6 and conclude the paper in Section 7.

2. Background and related work

System-level reliability has been addressed in a variety of ways through prior work spanning many years. Containment domains are necessitated by the increasing need for an efficient, comprehensive, and flexible mechanism to handle all errors in the most appropriate manner, and draw concepts and inspiration from this large body of work. Related work includes the distributed, hierarchical checkpointing used in large-scale systems as well as programming languages which use hierarchical transactional semantics to interface with checkpointed state.

Checkpointing is a generic state preservation and restoration mechanism which is widely used by large-scale compute clusters to tolerate failures. Many current systems take a *global checkpoint and restart* (g-CPR) approach, and establish a synchronized program state of every node in a centralized location. Global checkpointing, however, is not feasible in future systems – application working set sizes are increasing, while I/O bandwidth for transferring data to a centralized non-volatile location scales poorly.

Researchers have proposed coordinated *local checkpoint and restart* (l-CPR) to distribute state preservation and to overcome some of the inefficiency associated with centralized global checkpoints. Local checkpoints store the state of each node in a distributed fashion and are faster and more scalable than global checkpoints. While the checkpoints are stored locally, all checkpoint and recovery actions are globally coordinated and may perform poorly when failure rates are very high. A naive implementation of local checkpointing, however, cannot recover from permanent node failures. Accordingly, infrequent global checkpointing is often combined with local checkpointing to tolerate such failures [28,36]. Local checkpointing schemes may utilize DRAM [5,29], non-volatile memory [9] or other methods to gain efficiency. However, their motivation and operation remains fundamentally the same. Moody et al. further generalize local/global checkpointing into *scalable multi-level checkpoint and restart* (SCR) [24]. Hierarchical checkpointing (h-CPR) employs multiple checkpoints with differing costs and levels of error coverage. This multi-level approach can improve efficiency by speeding up the recovery of common-case errors, while using slower and more powerful checkpoints to preserve functionality.

In addition to coordinated l-CPR, researchers have suggested recovering checkpoints and re-executing

without coordination between nodes. Uncoordinated distributed checkpointing, however, may result in the *domino effect* [31] – given an arbitrary set of interacting processes with distributed checkpoints, a single error local to one process can cause all processes to completely roll back. Common approaches to eliminate the domino effect include communication logging [1,11, 13] and coordinated checkpointing based on communication history [19,20]. Such mechanisms, however, often sacrifice a certain degree of process autonomy and incur extra (often unpredictable) run-time and messaging overheads [37].

Unlike checkpointing schemes, other existing reliability approaches allow for fine-grained and flexible programmer control. Randell's *recovery blocks* (RBs) [31] hierarchically decompose a program into nested elements. Each Recovery Block is implemented in two independent-yet-equivalent routines and a functionality check. If a recovery block executes incorrectly, the secondary (fault-tolerant) routine is executed. Properly implemented recovery blocks may tolerate hard, soft, or software errors, and the programmer can nest blocks such that an parent block can resolve an error from an inner child.

Transactional programming is primarily used for controlling concurrency in shared-resource systems [22], but it can also be used to enforce reliability. Transactions, if so defined, can be nested to any depth. The nested transaction structure is naturally used to localize the effects of failures within the closest possible level of nesting in the transaction tree. This style of programming, including [7,21,26,32], is a generalization of the recovery block to the domain of concurrent programming. A significant challenge and source of inefficiency with such transactions is that each transaction must be made durable to guarantee recovery, resulting in storage and contention issues similar to those of checkpointing schemes.

Recent research on transactional memory (TM) [14, 16,25] mainly aims to enable efficient concurrent and lock-free programming. Transactional memory inherently provides state preservation and restoration at transaction boundaries. There has been prior work that extends transactional memory concepts to reliability, including Relax [8] and FaulTM [38]. Both use TM-like semantics for hardware/software collaborative reliability; both provide state preservation and restoration at a transaction boundary. These techniques, however, can only tolerate soft errors in computation and memory that can be detected while a transaction is still running.

2.1. Advantages of containment domains

Containment domains share some features and strengths with all the above approaches, but offer substantive usability and efficiency improvements (see Section 3). The hierarchical state preservation and restoration capabilities of CDs are similar in concept to multi-level distributed checkpoints. However, the flexibility of CDs allows the programmer and software system to tune the location and method of preservation and recovery to meet a desired level of reliability while maximizing 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, thus making these checkpoints inefficient and inflexible to application needs. As described in this paper, CDs are not susceptible to the domino effect because of their transactional qualities.

Conceptually, CDs are similar to recovery blocks – each constrains the detection and correction of errors to a local boundary, and each is able to pass uncorrectable errors upwards to be handled by the parent domain or block. CDs are, themselves, a limited form of hierarchical transactional programming. Unlike other transactional programming models, CDs allow more aggressive optimization by fully exploiting the storage hierarchy and by allowing the partial preservation and selective rematerialization of data. By flexibly using a range of error detection and correction mechanisms, CDs also allow for varying levels of reliability and performance depending on application needs. Using a range of error protection mechanisms also allows CDs to avoid the shortfalls of TM – CDs are able to provide high levels of reliability against a variety of errors, including permanent machine faults.

3. Containment domains

Each containment domain has four explicit components, which enable the application to express resilience tradeoffs and take advantage of the key insights described in Section 1. The *preserve* component locally and selectively preserves state for recovery. This preservation need not be complete – unpreserved state, if needed, may be recovered from elsewhere in the system or rematerialized. The *body* is then executed, followed by 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.

As explained in Section 1, 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 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 g-CPR. Figure 1 gives the organization of a hierarchy of nested CDs, with their four components shown.

3.1. Illustrative example

To make the explanations below more concrete we show a simple and illustrative example through iterative *sparse matrix–vector multiplication* (SpMV). This 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

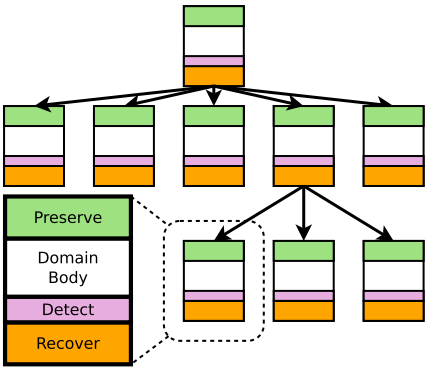

Fig. 1. The organization of hierarchical CDs. Each domain has four components, shown in color. The relative time spent in each component is not to scale. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

```
void task<inner> SpMV( in M, in V_i, out R_i) {
 cd = create_CD(parentCD);
 add_to_CD_via_copy(cd, M, …);
 forall(…) reduce(…)
   SpMV(M[…],V_i[…],R_i[…]);
 commit_CD(cd);
}

void task<leaf> SpMV(…) {
 cd = create_CD(parentCD);
 add_to_CD_via_copy(cd, M, …);
 add_to_CD_via_parent(cd, V_i, …);
 for r=0..N
  for c=rowS[r]..rowS[r+1] {
    R_i[r]+=M[c]*vec_i[cIdx[c]];
    check {fault<fail>(c > prevC);}
    prevC=c;
  }
 commit_CD(cd);
}
```

Fig. 2. Sequoia-style pseudocode for SpMV with CD API calls. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

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. We evaluate these features in depth using multiple mini-applications later in the paper.

Figure 2 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 [6] and target the hierarchical machine shown in Fig. 3. 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 Fig. 2 is for illustrative purposes and is based on an initial research prototype of CDs that is currently under development.

Table 1 summarizes how the SpMV program is partitioned into CDs and then mapped to the machine. We focus on representative error types and how they are protected with CDs. We also summarize what data is preserved at each level and how it is preserved – whether data is recovered locally (L), through a parent (P), or through a sibling (S). Note that we form our example with a deep CD hierarchy to illustrate the potential uses of containment domains. Later, when evaluat-

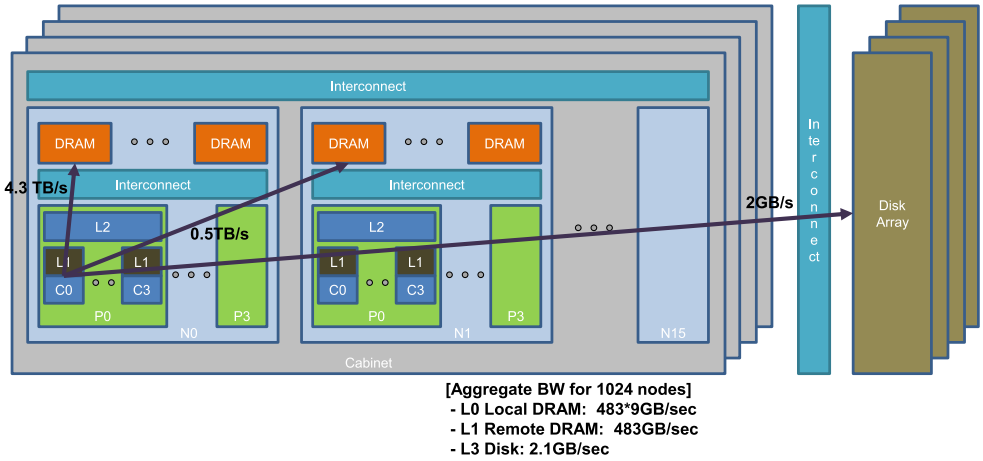

Fig. 3. An example system with 6 potential levels of CDs. Based on the configuration of [24]. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

Table 1
Protecting SpMV with CDs

	Error component type	CD level	Preservation method		
			M_{in}	V_{in}	V_{out}
Datapath	Soft	0	P	P	L
L1 cache	Soft	1	P	S	L
L2 cache	Soft	1	P	S	L
Core	Soft	1	L	S	L
DRAM	Soft	2	L	S	L
Processor	Hard	3	L	S	L
Node	Hard	4	L	S	L
Cabinet	Hard	5	L	S	L

ing the CD approach with a reasonable error model, the optimal mapping does not utilize CDs at a level finer than the processor–core granularity.

Figure 4 gives another example of the CD partitioning and mapping process. The application shown is a Monte Carlo neutron transport problem, and is described later in Section 5.3. Computation occurs in leaf CDs; leaves are grouped together under parent CDs, which are in turn nested. This nested CD structure is amenable to automatic analysis and optimization and we describe a tool that can be used to explore and autotune the design space in Section 4.

3.2. Hierarchical and partial preservation

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 Fig. 2 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 Fig. 2 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 5 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 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 Fig. 2). Each node has its own unique portion of the matrix, and thus the matrix can only be

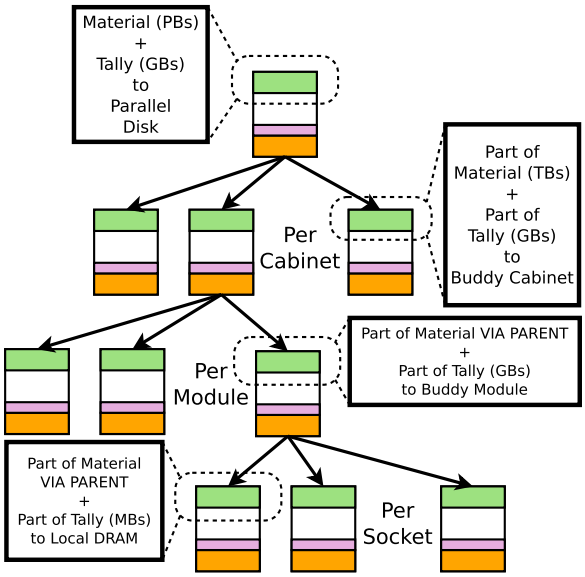

Fig. 4. Mapping example of neutron transport. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

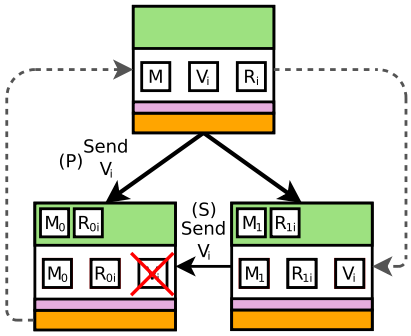

Fig. 5. Examples of recovery using the natural redundancy of SpMV. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

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 (Fig. 3), 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 [4,27]. CDs enable such automation without changing the underlying resilience paradigm while still providing the programmer with explicit control when needed.

The flexibility of CDs allows the application writer 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. We show how these crucial tradeoffs are amenable to automatic optimization in Section 4. Later, in Section 6, we demonstrate how these tradeoffs impact performance and efficiency for a range of machine and application characteristics.

3.3. Flexible error detection and recovery

We now describe how CDs enable flexible detection and recovery in combination with the state preservation and restoration schemes described above. The first step towards resilience is to detect errors and faults. Fault detection is not the focus of this paper and we assume that the hardware and runtime system support the detection of errors and faults in memory and compute modules. It is, however, rare for current systems to support error detection for arithmetic operations. Fortunately, the CD framework does *not* depend on full error detection support, and can evolutionarily exploit improved error detection capabilities as they become available. In cases where silent data corruption is in-

tolerable and the system does not provide complete error detection, the CD methodology can encapsulate the ideas of instruction or task duplication and can integrate them with the CD preservation and recovery mechanisms. In some systems, duplication-in-space is preferable and may improve performance.

Another interesting tradeoff that is natural within the containment domains framework is the inclusion of selective algorithmic-based fault tolerance. Algorithmic techniques can offer tremendous reductions in the overhead of error detection and correction, but may require certain guarantees for some portions of the code, as with fault-tolerant iterative methods [18]. With CDs, it is easy 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.

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 remap 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 do not discuss this aspect of recovery further in this paper.

Finally, in addition to the ability of CDs to fully utilize the memory hierarchy and to minimize the average restoration latency, the CD recovery routine 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. While we believe this ability may be used to improve the performance and scalability of CDs, a detailed analysis of rematerialization is left for future work.

An important property of CDs is their weakly transactional nature. The recovery procedure described above is not coordinated between the different CDs. Instead, it is initiated locally by a faulting CD and propagates up the hierarchy to the closest CD that can contain and recover the fault. This has several important advantages. First, no global barrier is imposed by the resilience scheme, improving scalability. 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.

3.4. Relaxed CDs

As described above, the *strict CD* hierarchy ensures that uncoordinated recovery is possible, but prohibits communication between concurrent CDs. If two parallel tasks need to communicate, they must both be within the same CD context. This restriction may introduce significant overheads in some cases. An illustration of this overhead is shown on the left hand side of Fig. 6. In this example, each inner CD requires significant preservation to ensure that common errors can be recovered entirely by re-executing the inner CD. Local recovery is essential because the communication patterns limit the depth of the CD hierarchy, increasing the cost of escalation to encompass all 12 inner CDs of this example. As an alternative to strict CDs in such a case, we offer the *relaxed CD* variant.

Relaxed CDs permit communication between concurrent domains. This requires the ability to log all communication into a CD. In addition, as with strict CDs, all data must be verified for correctness *before* being communicated. Given that only correct data is communicated, a failing CD may restore and re-execute in an uncoordinated manner by replaying received communications using its local log. Outbound communication is squashed during replay because the receiver cannot respond to an unexpected communication. In this manner, relaxed CDs allow uncoordinated recovery in the presence of communication (at the cost of some increased book-keeping overheads). This enables a new set of preservation/overhead tradeoffs as shown on the right hand side of Fig. 6.

Another possible use case of relaxed CDs is in situations where multiple tasks communicate in a chained fashion (task_{i+1} depends only on communication from task_i). In such a scenario, strict CDs effectively impose a non-programmatic barrier because all tasks must be within the same parent CD to communicate. This problem does not exist with relaxed CDs.

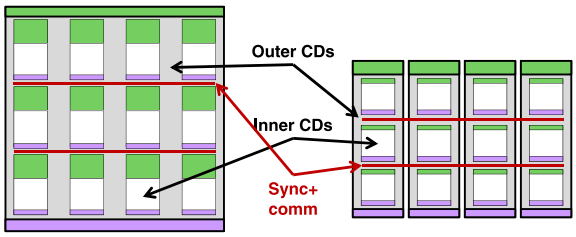

Fig. 6. Illustration of the tradeoffs between strict and relaxed CDs. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

4. CD mapping, tuning and modeling

Mapping an application to CDs involves two main parts. The first is identifying the application-specific CD properties (which are generally machine-agnostic); the second is mapping the application to a specific machine with information about the storage and bandwidth hierarchy, machine scale and expected error model. The CD properties are represented by the structure of the CD tree and the properties of each CD in the tree. The CD tree conveys information on parallelism, locality, communication and synchronization. Each CD within the tree specifies both the volume of data implicitly used by the CD from its parent and the volume of data communicated to it by its siblings. The CD also specifies which preservation methods are possible for each input. These machine-agnostic characteristics can ideally be determined by a compiler or profiler from the CD-annotated source code, but were extracted manually for this paper.

After the first stage of mapping, the CD tree is still abstract and has not yet targeted a specific machine. The exact instantiation of the tree and choice of preservation/restoration methods are determined during the second mapping phase. Finding the optimal mapping of the application onto a machine is challenging because of the numerous optimization options offered by CDs and the difficulty of estimating expected performance in large systems. Fortunately, the concise abstractions of CDs are amenable to automatic tuning and optimization using the CD characteristics. Towards this end, we developed a tool that takes a high-level description of an application structure and automatically sweeps the CD design space. The modeling tool can be used to determine tradeoffs such as: (1) what level of storage should be used for each preservation; (2) whether the CD hierarchy should be made deeper or more shallow to trade off localized recovery with preservation overheads (levels can be arbitrarily added or removed, so long as communication and synchronization semantics are preserved); (3) whether relaxed CDs provide compelling benefits for the application; and (4) whether data should be preserved locally, recovered from elsewhere in the machine, or rematerialized (trading off preservation and recovery overheads). We use the tool to map the applications and report the results, which are verified with a simulator, in Section 6.

4.1. Model assumptions

To simplify the analytical model and focus on the mapping and analysis of CDs, we make several assumptions. 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. Prior work has shown that this assumption is reasonable given typical fault rates and repair times [10,30]. For the error model we assume the presence of multiple independent fault/error processes that affect different aspects of the system. Each CD has an error rate associated with those errors it can locally recover (without escalation). We then associate these error processes with different levels of CD recovery. Finally, we assume that events within each error process are independently and identically distributed. Our error model is described in greater detail in Section 5. The implication of these assumptions is that we can use a binomial model for CD failure and re-execution: the probability that a CD fails, p, is directly proportional to its run time and the sum of all error rates that it contains.

At the execution model level, we assume that all errors associated with a CD are equivalent, regardless of their type and when they occur. Preservation and restoration may be asymmetric, but we do not account for errors triggering different forms of recovery in the same level of the CD tree. When a CD experiences an error, it restores its preserved state from the location at which it was preserved and then reexecutes the CD body. We assume zero overlap between execution and recovery; the extra time required for recovery is added to the time of a faulty CD. Recovery of different siblings, however, can proceed in an uncoordinated fashion in the absence of a synchronization or blocking communication.

4.2. Analytical model

Our analytical model estimates the execution time (performance) and average power consumption (or energy) of an application. We describe the execution-time

model first and later discuss power modeling. The performance model accepts a description of the CD tree, which conveys information on the properties of each CD, as well as the parallelism, synchronization and communication boundaries between CDs. Each CD includes information on the expected preservation and restoration times (preservation and restoration may be asymmetric) and leaf CDs also include their expected execution time, encompassing both the CD body and any detection overheads. Each CD may have multiple sequential CDs executing within it (which may, in turn, include nested CDs). Sequential child CDs allow us to express preservation, restoration, and execution at intermediate levels of the CD tree, because nesting degree may differ for each of the CDs in sequence. Note that the analysis tool derives preservation time and restoration time from the volume of preserved and communicated state and available machine bandwidths. For simplicity, we assume that the parameters of a CD are fixed for all of its dynamic instantiations.

Because of the CD hierarchy we need only consider the impact of resilience on two CD levels at any given time – a parent and its children. Analysis starts with the lowest two levels, where children are leaves, and derives the impact of resilience on the parent. At that point, the parent execution properties are modified based on the preservation and recovery overheads associated with it children and the entire two levels are encapsulated. This process continues until the root level is reached and the entire application properties are estimated. Due to this recursive process, we describe the model with respect to a parent with a set of child CDs. To explain the model, we present the performance model in four steps: (1) for a parent CD that has n identical children that all execute in parallel with no sequential loops; (2) we extend the model to include serial dependencies; (3) we then allow sequential groups of CDs to be heterogeneous; and (4) finally include asymmetric preservation and recovery times.

4.2.1. Parent with n identical parallel children

We start with a parent CD that has n identical parallel CDs. We first restrict ourselves to the case where the execution and recovery times for a particular child CD, T_c are uniform and do not account for execution variation. When a child fails, it is re-executed in full. During re-execution, the child CD may experience another error and may re-execute again. When n independent parallel CDs are grouped within the parent, the expected execution time of the parent is directly proportional to the expected maximum number of consecutive failures experienced by any one of the n parallel CDs.

Due to the model assumptions, the number of iterations of each CD follows a geometric random variable. While the statistics of geometric variables are well understood, we derive our model from first principles to allow us to incorporate the more complex behavior of asymmetric recovery and heterogeneous CDs. Let $q[x, n]$ be the probability that all child CDs experience at most x consecutive failures. We then derive $d[x, n]$, the probability that the child with the most consecutive failures experiences exactly x failures and then succeeds. We derive $d[x, n]$ iteratively by subtracting the probability that fewer than x failures occur from the probability that at most x failures occur (thus leaving only the probability of exactly x failures). We use $d[x, n]$ to compute the expected run time of the parent. In all equations, we use p_c to represent the probability that a child fails; we derive from the inherent error rate (p) associated with the child.

$$p_c = T_c p, \tag{1}$$

$$q[0, n] = (1 - p_c)^n, \tag{2}$$

$$q[x, n] = \left(\sum_{i=0}^{x-1} \left(p_c^i (1 - p_c) \right) \right)^n, \tag{3}$$

$$d[0, n] = q[0, n], \tag{4}$$

$$d[x, n] = q[x, n] - \sum_{i=0}^{x-1} d[i, n], \tag{5}$$

$$T_{parent}[x, n] = \sum_{i=0}^{\infty} (i + 1) T_c d[i, n]. \tag{6}$$

4.2.2. Serial dependencies between children

We now include the case where there are n parallel siblings, each responsible for executing m CDs sequentially. We follow the same derivation as above, but extend the definitions of the functions as follows. Let $q[x, m, n]$ be the probability that each of the siblings experiences at most x failures in the m serial CDs they contain. Similarly, $d[x, m, n]$ is the probability that the sibling with the most failures experiences exactly x failures before all of its m CDs succeed. This behavior is illustrated in Fig. 7 and the model is shown below.

$$q[0, m, n] = (1 - p_c)^{m \cdot n}, \tag{7}$$

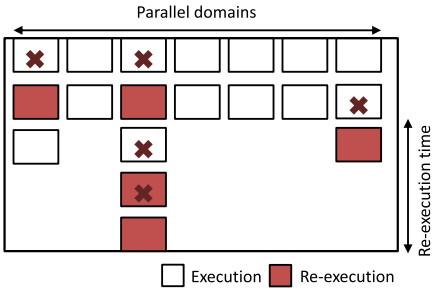

Fig. 7. Example of 7 nodes, each executing 2 sequential child CDs. Some CDs fail and require re-execution, which is overlapped between nodes with uncoordinated recovery. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

$$q[x,m,n] = \left(\sum_{i=0}^{x-1}\left(\binom{i+m-1}{i} \times p_c^i(1-p_c)^m\right)\right)^n, \quad (8)$$

$$d[0,m,n] = q[0,m,n], \quad (9)$$

$$d[x,m,n] = q[x,m,n] - \sum_{i=0}^{x-1} d[i,m,n], \quad (10)$$

$$T_{parent}[x,m,n] = \sum_{i=0}^{\infty}(i+m)T_c d[i,m,n]. \quad (11)$$

4.2.3. Heterogeneous CDs

We now extend the model to allow sequential CDs within a parent to differ. We rely on the assumption that all error processes are independent and derive the properties of an equivalent "average" child CD that can be directly substituted into the model for expected parent execution time. These properties are shown below assuming there are t CD types, each with its own error model (different failure probabilities $\{p_i\}_{i=1}^t$) and execution time $\{T_{c,i}\}_{i=1}^t$ (these CDs execute sequentially, which allows us to generalize the error model easily). Note that because the re-execution time now depends on which type of CD failed, we weight the average recovery time based on the execution time of each CD type.

$$p_{c,avg} = \sum_{i=1}^{t}\frac{T_{c,i}p_i}{t}, \quad (12)$$

$$T_{c,noerr} = \sum_{i=1}^{t}\frac{T_{c,i}}{t}, \quad (13)$$

$$T_{c,rexec} = \frac{\sum_{i=1}^t T_{c,i}^2}{\sum_{i=1}^t T_{c,i}}, \quad (14)$$

$$T_{parent}[x,m,n] = \sum_{i=0}^{\infty}(mT_{c,noerr} + iT_{c,rexec})d[i,m,n]. \quad (15)$$

4.2.4. Asymmetric preservation and restoration

Finally, to account for the asymmetric preservation and restoration times needed for certain preservation tradeoff optimizations, we modify the above model slightly and replace $T_{c,rexec}$ with Eq. (16), where $T_{c,i,exe}$ is the time to execute the body of CD type i, $T_{c,i,prsrv}$ is the time for preservation, and $T_{c,i,rstr}$ is the restoration time, which may be different than preservation time. This modification relies on the simplifying assumption that the error rates are low and that the asymmetry is relatively small (i.e. that compute time dominates preservation and restoration time)

$$T_{c,rexec} = \frac{\sum_{i=1}^t (T_{c,i,prsrv} + T_{c,i,exe})(T_{c,i,rstr} + T_{c,i,exe})}{\sum_{i=1}^t T_{c,i}}.$$

4.2.5. Power model

Because CDs enable localized recovery, execution resources associated with CDs that are not re-executing may remain idle. We account for this effect using a simple model and evaluate the relative power consumed by CDs. We assume that actively executing a CD has a relative power of 1, while a node that is idling consumes a relative power of α (α is a machine-dependent parameter, which we set to $\alpha = 0.25$ in our experiments). For a particular parent CD with n parallel children (or child sequences), we estimate the expected idle time as follows. Assuming that n is large, the overall expected number of re-executions is simply n times the average number of times a child re-executes. Thus, the idle time is the total execution slots occupied by the parent CD (n times the expected execution time of the parent) minus the expected re-executions.

5. Methodology

Evaluating the viability of resilience schemes at exascale is challenging. Several days of run time (and

prohibitively many CPU hours) are required to observe a meaningful number of errors. This necessitates coarse-grained modeling. We use a combination of a generalized and simplified trace-driven simulator that operates at the granularity of CDs and the analytical model described above. Both the simulator and analytical model reproduce the essential behavior of resilience schemes and follow the execution model of CDs and the CD mapping parameters as described in Section 4. We use the simulator on smaller configurations to verify our intuition and to validate the analytical model, and report results for full-scale systems using the analytical model only. As mentioned earlier, the CD framework is flexible enough to subsume and encompass many prior resilience approaches. We demonstrate this by using our tools to also evaluate coordinated global checkpointing (g-CPR) and multi-level hierarchical checkpoint-restart (h-CPR). In the subsections below we briefly describe the system parameters, error model, and applications that are used to evaluate the CD framework.

5.1. System parameters

Table 2 lists the seven system configurations we use for evaluation. We base the systems on reasonable assumptions concerning future technology trends towards exascale, targeting the 7 nm technology node. We assume that each core will have a peak performance of 10 GFLOPS and that system performance scales linearly with the number of cores. We model main memory based on the expected parameters of future memory packaging and advanced I/O technology that may reach 200 GB/s per socket.

We fix memory capacity at 1 GB per core. We also assume that high-bandwidth non-volatile memory (NVM) is integrated within each module (or node). Integrated NVM may offer about an order of magnitude less bandwidth than DRAM and provide about an order of magnitude larger capacity. We expect that NVM will be used for preserving local and/or remote data (buddy storage), assuming that unused DRAM may be at a premium for many applications. We expect that intra-module interconnect will not limit bandwidth to NVM, but that bandwidth to remote NVM decreases by an order of magnitude per level because of the global network. Lastly, we set the per-core parallel file system bandwidth of the smallest configuration to 0.001 GB/s/core, based on the bandwidth reported for the 2.5 PFLOPS Jaguar system [2]; we assume that global file system bandwidth per core decreases by 10% each time the number of cores doubles.

While we believe our machine configurations are reasonable, different machines will have different parameters. We therefore run sensitivity experiments and vary the ratios of available bandwidths for preservation. We do not model application execution time directly and do not attempt to evaluate application scalability beyond the impact of resilience. We do this by ignoring any potential bottlenecks of communication and synchronization on the run time of the leaf CDs. The projected memory capacities and bandwidths are

Table 2
Machine parameters

Parameters	2.5	10	40	160	640	1280	2560
				Peak performance (PFLOPS)			
Number of cores per socket	64	64	64	128	128	256	256
Number of sockets per module	2	2	4	4	8	8	8
Number of modules per cabinet	32	32	64	64	128	128	128
Number of cabinets	64	256	256	512	512	512	1024
Total number of cores	262,144	1,048,576	4,194,304	16,777,216	67,108,864	134,217,728	268,435,456
Total memory capacity (TB)	256	1024	4096	16,384	65,536	131,072	262,144
				Failure in time (FIT)			
Processor	1.89E+07	7.55E+07	3.02E+08	1.21E+09	4.83E+09	9.66E+09	1.93E+10
Memory	3.60E+04	1.44E+05	5.76E+05	2.30E+06	9.21E+06	1.84E+07	3.68E+07
Software	3.41E+05	1.37E+06	2.73E+06	5.46E+06	1.09E+07	1.09E+07	2.18E+07
Power (Module)	4.10E+05	1.64E+06	3.28E+06	6.55E+06	1.31E+07	1.31E+07	2.62E+07
Power (Cabinet)	1.18E+05	4.71E+05	4.71E+05	9.42E+05	9.42E+05	9.42E+05	1.88E+06
Power (Multi-cabinet)	1.18E+04	4.71E+04	4.71E+04	9.42E+04	9.42E+04	9.42E+04	1.88E+05
Power (Global)	4.09E+03	4.09E+03	4.09E+03	4.09E+03	4.09E+03	4.09E+03	4.09E+03
Mean time to interrupt in hours	50.52	12.63	3.24	0.82	0.21	0.10	0.05

used solely for estimating the overheads of preservation and restoration.

5.2. Error/fault model

Projecting fault and error rates to the exascale regime is arguably more error-prone than the prediction of system parameters. Our assumptions on error scaling are summarized at the bottom of Table 2. We assume that the per-core soft-error susceptibility will remain roughly constant as technology scales. This accounts for increased transistor density and susceptibility coupled with improvements in error protection and detection techniques. Even under this assumption, the per-processor error rate we use is more than an order of magnitude greater than typical HPC processors today. For memory, we project that error rates will increase in proportion to memory capacities. We assume that socket failures, due to either system software or hard faults, will scale directly with the number of sockets, and similarly that power and network related failures will scale with the number of modules and cabinets. We expect multi-cabinet and global failures to be one and two orders of magnitude less frequent than single-cabinet failures, respectively. We base the initial values for fault and error rates on numbers reported in recent published work [15,33–35] and on private conversations with several hardware vendors. As with system configurations, we did our best to choose what we believe are reasonable numbers for future systems, and also evaluate sensitivity by sweeping error rates.

5.3. Applications

We evaluate three mini-applications that we map to CDs: a Monte Carlo method neutron transport application, hierarchically-blocked iterative sparse matrix–vector multiplication, and the HPCCG conjugate-gradient based linear system solver from the Mantevo mini-application suite [17]. We break each application into hierarchical CDs and then map it onto each of the system configurations. We tune CD parameters using the tool described in Section 4. We discuss the mapping options and parameters in the next section. We also obtain the optimal checkpoint intervals for h-CPR and g-CPR for each application, machine configuration and error rate.

5.3.1. Neutron Transport (*NT*)

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. For this application, we assume that the checkpointing schemes preserve a volume equivalent to 80% of each node's memory.

5.3.2. Sparse Matrix Vector Multiplication (*SpMV*)

We scale SpMV to use a fixed percentage of memory and to perform roughly the same amount of work per core of each machine. One iteration takes roughly 0.5 s to execute and consumes 50% of memory. At the end of each iteration there is a barrier, followed by communication. In the strict model, all leaf CDs must complete before any communication can occur. Thus, the run time of each leaf CD is limited by its communication interval. As explained in the detailed example in Section 3, leaf CDs utilize preservation tradeoffs: intermediate indices and sums are preserved locally in DRAM while the inputs are preserved via parent. SpMV can take advantage of relaxed CDs to increase the interval between full preservation, but our experiments did not indicate significant gain in performance efficiency. For this application, the checkpointing schemes preserve a volume equivalent to 50% of each node's memory.

5.3.3. HPCCG

HPCCG is a linear system solver that uses the conjugate gradient method and can scale to a large number of nodes [17]. 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. 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. HPCCG offers similar relaxed-CD tradeoffs as SpMV and similarly did not exhibit significant differences in performance efficiency. For this application, the checkpointing schemes preserve a volume equivalent to 10% of each node's memory.

5.4. Simulator and model validation

As mentioned previously, containment domain semantics can capture the behavior of other resilience schemes. To validate our simulator, we simulate the multi-level hierarchical checkpoint restart resilience scheme with the configuration described by Moody et al. [24]. We thus make a direct comparison, which demonstrates a good match between the two evaluation methodologies (Fig. 8). We use the *performance efficiency*, which is the fraction of machine resources (nodes and time) used to make forward progress, as our main evaluation metric. We also tested a range of configurations and verified that the analytical model and the CD-level simulator match within 3% in reported performance efficiency. Note that a full system implementation of CDs requires the management of substantial metadata, the cost of which is not captured by either the model or simulator. The application mappings in this paper do not use deep CD hierarchies or fine-grained domains, however, such that these metadata management overheads can be neglected. Initial experiments with a prototype that is still under development confirm this assumption.

6. Evaluation

In this section, we discuss the results of exploring a realistic subset of the CD design space. We find that CDs offer superior flexibility and that fine-tuning resilience schemes improves performance and energy efficiency when compared to alternative approaches. The results also show that CDs scale well to very large system sizes and error rates, while even hierarchical checkpoint-restart loses significant efficiencies in such cases. Note that the point of the evaluation is to study the effectiveness of CDs and that the optimizations are enabled in a structured manner, using our modeling tool for tuning rather than relying ad-hoc approaches. We do not claim that previous methods are inadequate or that similar benefits cannot be attained with explicit tuning and code modifications.

Figure 9 shows the expected performance efficiency and energy overheads due to resilience for each application when using CDs, h-CPR and g-CPR. The solid bars indicate performance efficiency when compared to an implementation that assumes no failures or errors and performs no preservation or recovery. Similarly, the clear bars represent the overall relative energy overhead (average power multiplied by execution time relative to baseline, or more accurately, average power divided by performance efficiency). Because uncertainty exists in both machine parameters and error rates, we also evaluate the applications and systems with error rates that are 10 and 50 times higher than our baseline and with a preservation overhead that is 10 times higher than baseline (to model lower global bandwidth than we projected). We summarize these results in Fig. 10.

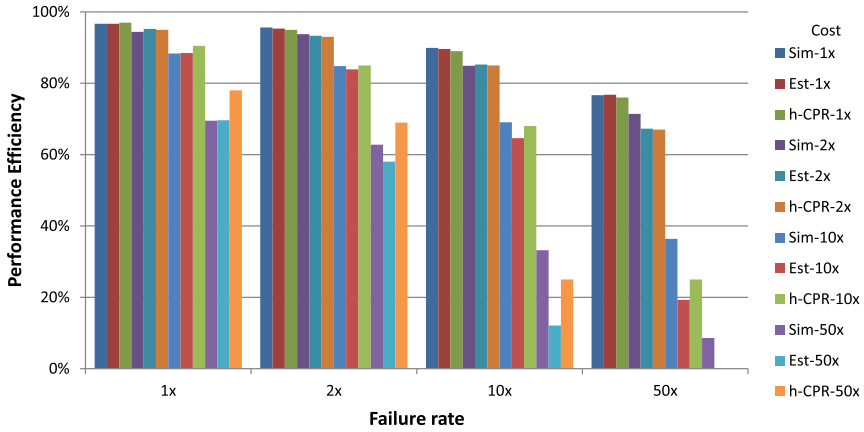

Fig. 8. Comparison of the CD simulator and an h-CPR analytical model [23]. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

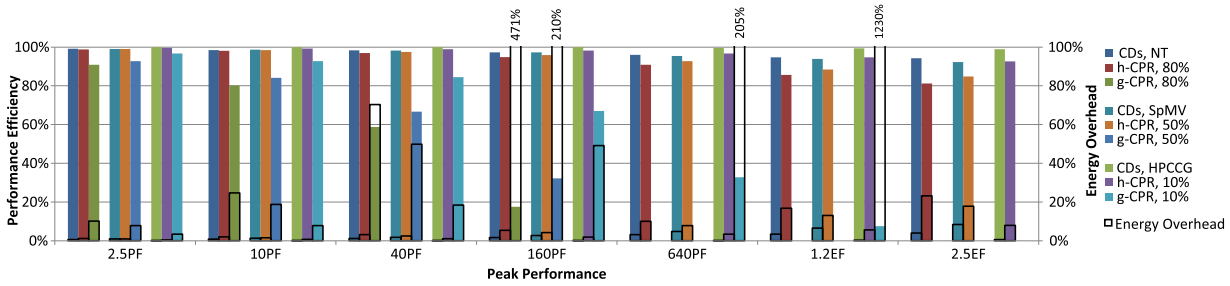

Fig. 9. Performance efficiency (solid bars) and energy overheads (clear bars) with CDs, h-CPR and g-CPR for the three applications across a range of system scales. For each application, we indicate the equivalent fraction of memory preserved by the checkpointing schemes. Note that g-CPR fails to complete execution with some applications and scales and often incurs very high energy overheads. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

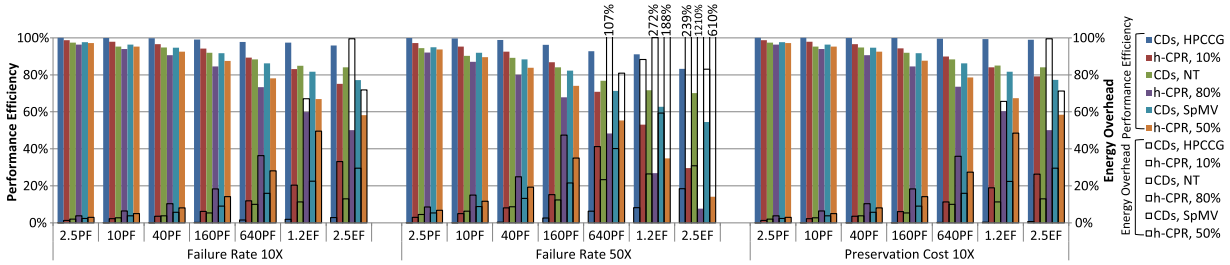

Fig. 10. Performance efficiency of the three applications using CDs and h-CPR with different relative error rates and preservation overheads. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-130374.)

The overall conclusion from our experiments is that CDs consistently outperform the alternatives. Both CDs and h-CPR utilize the storage hierarchy well, and (with our baseline machine parameters) both scale to the largest configuration. At that configuration, CDs exhibit a 5–15% performance efficiency and a 7–19% energy efficiency advantage over h-CPR. Note that h-CPR causes all nodes to re-execute on recovery, and thus energy overheads precisely track the inverse of performance efficiency. With the uncoordinated and overlapped recovery enabled by CDs, energy efficiency is improved when compared to overall execution time. This improvement is, however, small in the applications we evaluate because they have short-duration leaves and a small degree of nesting, limiting the potential benefits of hierarchical re-execution. Global checkpoint-restart, as expected, does not scale well in terms of either performance or energy overheads. In fact, in some experiments, g-CPR failed to make forward progress. When either error rate or preservation overhead is increased from the baseline, even h-CPR starts suffering from low efficiency at the target configurations. With CDs, however, high efficiency is achieved; the lowest efficiency with CDs is still better than a full replication approach [3], which suffers from a greater than 50% penalty.

Of all three applications, neutron transport (NT) is best at utilizing the hierarchy and tradeoffs enabled by CDs. Because the computation of NT has long periods of time between communications, each CD level can be tuned to the optimal preservation interval. In addition, by preserving most of its data via a parent, the error-free preservation overhead is much lower than with a straightforward implementation using h-CPR. Even though the memory footprint of NT is 80% of all available memory, leaf CDs only preserve a small fraction to DRAM. We assume a single checkpointing scheme with h-CPR, which requires preserving the entire dataset at each interval. Because a large fraction of memory is utilized h-CPR must forgo DRAM-level preservation entirely and rely on NVM. The efficient preservation of NT with CDs enables it to tolerate even a 10× increase in error rate or preservation overhead while achieving greater than 80% efficiency.

Like NT, SpMV and HPCCG also preserve some of the leaf data via a parent. Unlike NT, these two applications have a smaller memory footprint. SpMV utilizes 50% of memory, which reduces the preservation overhead of direct checkpointing, but still prevents h-CPR from preserving to DRAM. CDs thus offer a very substantial benefit, especially for larger configurations and higher error rates and preservation overheads. De-

spite this advantage, the overall performance of SpMV is worst among the three applications. The reason is that the tight communication requirements of the program, coupled with significant memory use, reduce the efficiency of preservation, even with CD optimizations. HPCCG only uses 10% of memory and is thus the most scalable when using either CDs or h-CPR. With CDs, even with the largest machine and an error rate $50\times$ higher than our baseline estimate, HPCCG with CDs still achieves over 80% efficiency, which is far better than the only 30% achievable with h-CPR or the $<50\%$ expected with full redundant MPI ranks. Because its memory footprint is small and the fact that preservation is optimized, HPCCG with CDs is insensitive to preservation overheads.

7. Conclusions

We present the concept of *containment domains*, a flexible abstraction which encapsulates and expresses resilience concerns. CDs rely on weak transactional semantics and nesting to enable hierarchical and distributed state preservation, state restoration, and recovery. In addition, CDs can tune error detection and recovery for different error types, allowing the resilience scheme to adapt to available error detection mechanisms and changing application needs. CDs are flexible enough to leverage a large variety of existing system resilience approaches without changing the programming model or reverting to ad-hoc implementations. Thus, the single CD abstraction can be used to tune across a wide range tradeoff options. We demonstrate through experimental evaluation that the tradeoffs introduced by CDs are amenable to automatic optimization with a CD-oriented analysis tool that we implemented. The results show that hierarchical state preservation and restoration with uncoordinated hierarchical recovery can be exploited by CDs to improve performance and energy efficiency even at very large machine scales in the presence of a high rate of errors and failures. Some of our planned future work includes the full-scale analysis of large applications mapped to the CD framework, a deeper integration with existing programming models, and the investigation of CD performance with application-specific fault tolerant algorithms.

Acknowledgements

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. Funding sources include DARPA contract HR0011-10-9-0008, the National Science Foundation Grant #0954107 and support from the Texas Advanced Computing Center.

References

[1] B.H.L. Alvisi and K. Marzullo, Nonblocking and orphan-free message logging protocols, in: *Proc. IEEE Fault Tolerant Computing Symp. (FTCS)*, 1993.

[2] A. Bland, R. Kendall, D. Kothe, J. Rogers and G. Shipman, Jaguar: The world's most powerful computer, in: *Proceedings of CUG*, 2009.

[3] R. Brightwell, K. Ferreira and R. Riesen, Transparent redundant computing with MPI, *Recent Advances in the Message Passing Interface*, 2010, pp. 208–218.

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

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

[6] Cray Inc., Containment domains API, April 2012, available at: lph.ece.utexas.edu/public/CDs.

[7] C.T. Davies, Jr., Data processing spheres of control, *IBM Systems Journal* **17**(2) (1978), 179–198.

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

[9] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann and Y. Xie, Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems, in: *Proc. of the International Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, November 2009.

[10] A. Ejlali, B. Al-Hashimi and P. Eles, A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems, in: *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, IEEE/ACM, 2009.

[11] E. Elnozahy and W. Zwaenepoel, Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit, *IEEE Transactions on Computers* **41**(5) (1992), 526–531.

[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: *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ACM, New York, NY, USA, 2006, p. 83.

[13] 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 2011 International Parallel Distributed Processing Symposium (IPDPS)*, May 2011, pp. 989–1000.

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

[15] T. Heijmen, Radiation-induced soft errors in digital circuits – a literature survey, Philips Electronics Nederland, Technical Report 2002/828, 2002.

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

[17] 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, Technical Report SAND2009-5574, Sandia National Laboratory, 2009.

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

[19] R. Koo and S. Toueg, Checkpointing and rollback-recovery for distributed systems, *IEEE Transactions on Software Engineering* **SE-13**(1) (1987), 23–31.

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

[21] B. Liskov and R. Scheifler, Guardians and actions: Linguistic support for robust, distributed programs, *ACM Trans. Program. Lang. Syst.* **5** (1983), 381–404.

[22] N.A. Lynch, Concurrency control for resilient nested transactions, in: *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS'83*, ACM, New York, NY, USA, 1983, pp. 166–181.

[23] 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 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society, 2010, pp. 1–11.

[24] 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), Technical Report LLNL-TR-440491, July 2010, available at: https://e-reports-ext.llnl.gov/pdf/391238.pdf.

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

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

[27] A. Oliner, L. Rudolph and R. Sahoo, Cooperative checkpointing theory, in: *Proc. of the International Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

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

[29] J.S. Plank, K. Li and M.A. Puening, Diskless checkpointing, *IEEE Trans. Parallel and Distributed Systems* **9**(10) (1998), 972–986.

[30] D. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, 1996.

[31] B. Randell, System structure for software fault tolerance, in: *Proceedings of the International Conference on Reliable Software*, ACM, New York, NY, USA, 1975, pp. 437–449.

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

[33] B. Schroeder and G. Gibson, A large-scale study of failures in high-performance computing systems, *IEEE Transactions on Dependable and Secure Computing* **7**(4) (2010), 337–351.

[34] B. Schroeder, E. Pinheiro and W. Weber, DRAM errors in the wild: a large-scale field study, in: *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ACM, 2009, pp. 193–204.

[35] C. Slayman, Impact and mitigation of DRAM and SRAM soft errors, IEEE SCV Reliability Seminar, May 2010, available at: http://www.ewh.ieee.org/r6/scv/rl/articles/Soft%20Error%20mitigation.pdf.

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

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

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