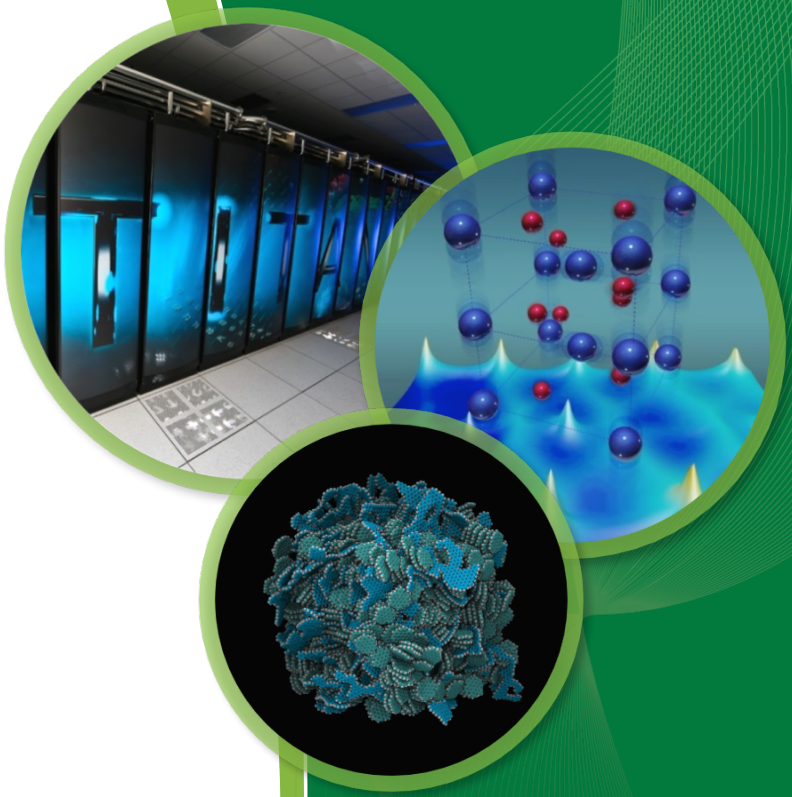
Language Support for Reliable Memory Regions



Saurabh Hukerikar,
Christian Engelmann

Computer Science & Mathematics Division
Oak Ridge National Laboratory

ORNL is managed by UT-Battelle
for the US Department of Energy

OAK RIDGE National Laboratory

Outline

- Motivation
  - Trends in HPC architectures
  - Challenges for future extreme-scale systems
- Concept of Havens
  - Memory-management with emphasis on reliability
  - Models of reliability for HPC applications through use of havens
  - Abstract Interface
- Language Support for Havens
  - Limitations of library interfaces & desirable features
  - Syntax & Semantics
- Experimental Evaluation: Fault injection and performance studies

ORNL's Next Machine: Summit

ATTRIBUTE	TITAN	SUMMIT
Compute Nodes	18,688	~3,400
Processor	(1) 16-core AMD Opteron per node	(Multiple) IBM POWER 9s per node
Accelerator	(1) NVIDIA Kepler K20x per node	(Multiple) NVIDIA Volta GPUs per node
Memory per node	32GB (DDR3)	>512GB (HBM+DDR4)
CPU-GPU Interconnect	PCI Gen2	NVLINK (5-12x PCIe3)
System Interconnect	Gemini	Dual Rail EDR-IB (23 GB/s)
Peak Power Consumption	9 MW	10 MW

Each Summit node will contain 512GB HBM + DDR4, and an additional 800GB NVRAM

DOE ASCR Computing: Upcoming Systems

System attributes	NERSC Now	OLCF Now	ALCF Now	NERSC Upgrade	OLCF Upgrade	ALCF Upgrade	
Planned Installation	Edison	TITAN	MIRA	Cori 2016	Summit 2017-2018	Theta 2016	Aurora 2018-2019
System peak (PF)	2.4	27	10	>30	>150	>8.5	>150
Peak Power (MW)	3	8.2	4.8	< 3.7	10	1.7	13
System memory per node	64 GB	38 GB	16 GB	~1 PB DDR4 + High Bandwidth Memory (HBM)+1.5PB persistent memory	> 1.74 PB DDR4 + HBM + 2.8 PB persistent memory	>480 TB DDR4 + High Bandwidth Memory (HBM)	> 7 PB High Bandwidth On-Package Memory Local Memory and Persistent Memory
Node performance (TF)	0.460	1.452	0.204	> 3	> 40	> 3	> 17x Mira
	Intel Ivy Bridge	AMD Opteron NVidia Kepler 64-bit	PowerPC A2	Intel Knights Landing many core CPUs Intel Haswell CPU in data partition	Node processors Multiple IBM Power9 CPUs & multiple NVidia Voltas GPUS	Intel Knights Landing Xeon Phi many core CPUs	Knights Hill Xeon Phi many core CPUs
System size (nodes)	5,600 nodes	18,688 nodes	49,152	9,300 nodes, 1,900 nodes in data partition	~3,500 nodes	>2,500 nodes	>50,000 nodes
System Interconnect	Aries	Gemini	5D Torus	Aries	Dual Rail EDR-IB	Aries 2nd Generation	Intel Omni-Path Architecture

Challenges for Next Generation HPC

Old Optimization Targets

Peak: clock frequency as primary limiter for performance improvement

Cost: FLOPs are biggest cost for system: optimize for compute

Concurrency: Modest growth of parallelism by adding nodes

Memory scaling: maintain byte per flop capacity and bandwidth

Locality: MPI+X model (uniform costs within node & between nodes)

Uniformity: Assume uniform system performance

Reliability: It's the hardware's problem

New Optimization Targets

Power: is first-order design constraint for future HPC system design

Cost: Data movement dominates: optimize to minimize data movement

Concurrency: Exponential growth of parallelism within chips

Memory Scaling: Compute growing 2x faster than capacity or bandwidth

Locality: must reason about data locality and possibly topology

Heterogeneity: Architectural and performance non-uniformity increase

Reliability: Cannot count on hardware protection alone

Need for software-based techniques to navigate these emerging challenges

Havens: Reliable Memory Regions

- Memory-Management Style
- Havens represent areas of memory
- Program objects are allocated inside the havens
- Reliability of the haven implicitly guaranteed through software-based schemes
- Makes no assumptions about hardware-based protection schemes
- All-at-once deallocation; other deallocation policies possible

```
//Create a new haven
Haven h = new_haven();

//Allocate objects ”inside” the haven
double* x = haven_alloc(h, N * N * sizeof(double));

//Operate on objects inside haven
work(x);

//Delete haven – deletes all objects at once
delete_haven();
```

Havens: Abstract Interface

haven_create:
- request for the creation of a haven by an application
- returns a handle to the memory region, but no memory is allocated.
- error protection for haven scheme is specified

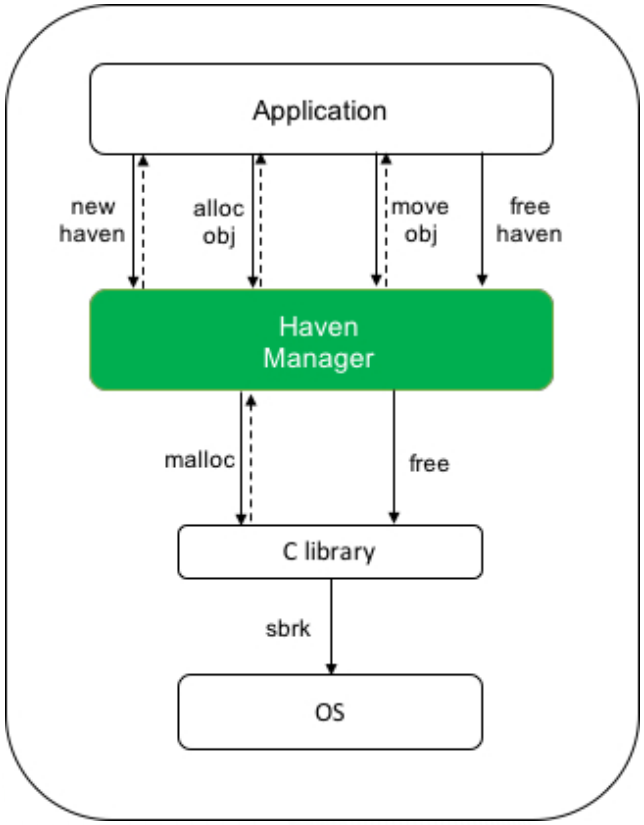
haven_alloc:
- request for a specified block of memory “within” a haven
- results in the allocation of the memory
- initialization of state related to the protection scheme

haven_delete:
- indicates intent to delete an object within a haven
- memory is not released until the haven is destroyed

haven_read, haven_write:
- read/update the program objects contained in the haven
- state of haven’s resilience mechanism updated

haven_destroy:
- results in all memory blocks allocated in the region to be deallocated; memory is available for reuse.
- no further operation on the haven are permitted

haven_relax, haven_robust:
- enable the error protection scheme applied to a haven to be turned on and off



Initial prototype implementation based on library-based implementation*

* Havens: Explicit Reliable Memory Regions for HPC Applications, IEEE High Performance Extreme Computing Conference 2016 (HPEC ‘16)

Havens: Expressing Locality & Resilience

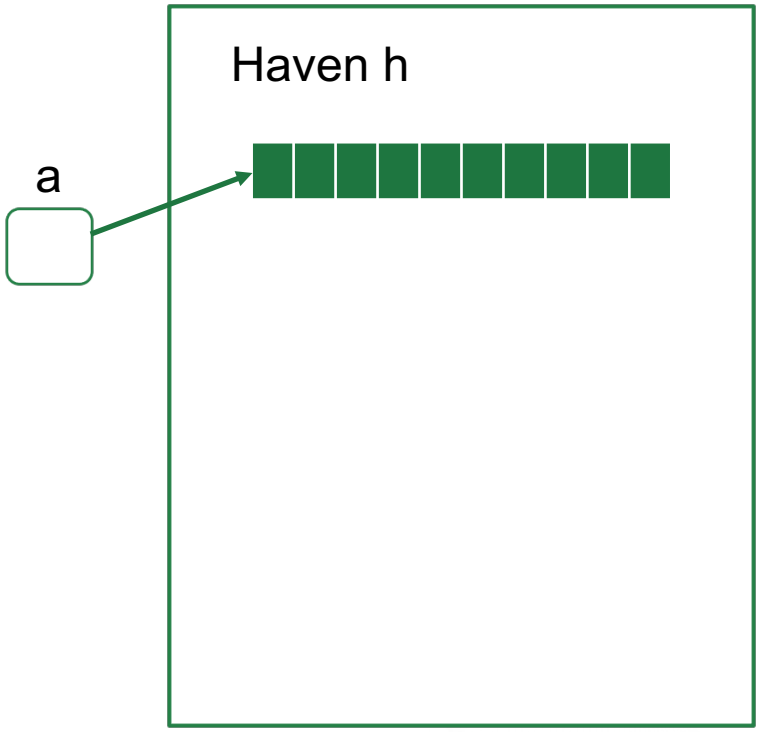
- Havens may be used to express locality:
  - Allocations within a haven may be optimized for cache/memory usage
  - Aggregation of several small objects into a single haven
- Resilience Scheme for Havens
  - *Default:* Parity-based XOR signatures (based on erasure codes)
  - Each haven may implement a separate protection mechanism
  - Agnostic to data structure
  - Aggregation of program objects based on their resilience needs

Using Havens: Vector Addition Example

```
Haven h = new_haven();
```

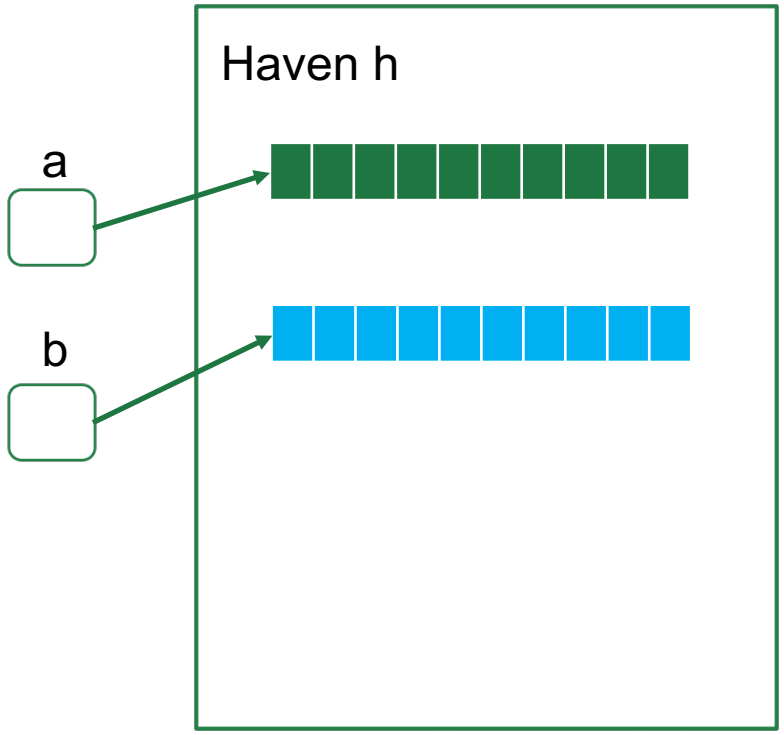
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h,
            N * sizeof(double));
```



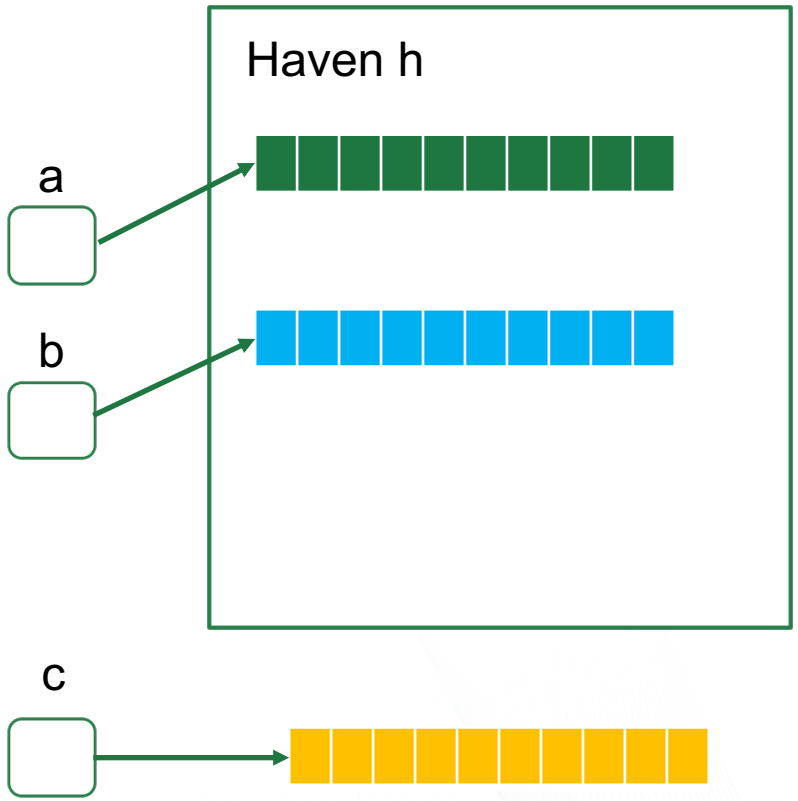
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h,
              N * sizeof(double));
double* b = haven_alloc(h,
              N * sizeof(double));
```



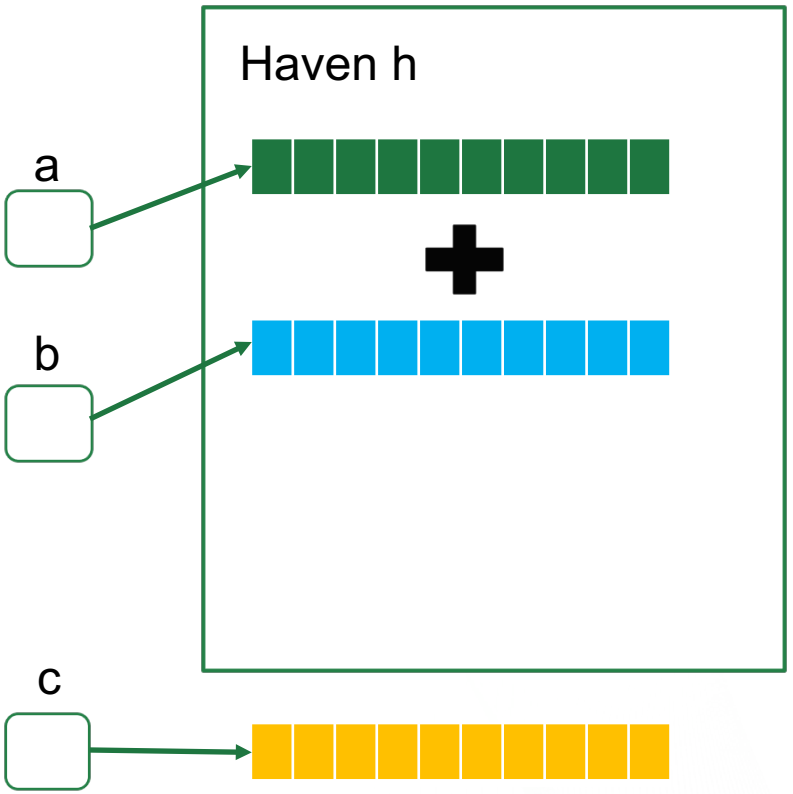
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h,
            N * sizeof(double));
double* b = haven_alloc(h,
            N * sizeof(double));
double* c = malloc(N *
            sizeof(double));
```



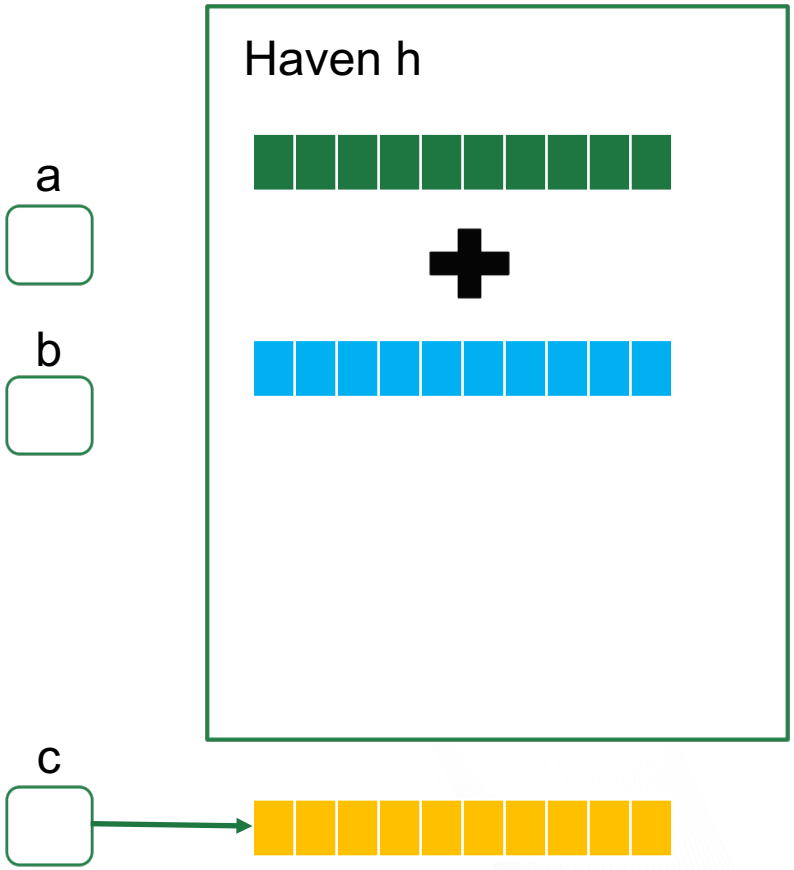
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h,
            N * sizeof(double));
double* b = haven_alloc(h,
            N * sizeof(double));
double* c = malloc(N *
             sizeof(double));

vector_addition(c, a, b);
```



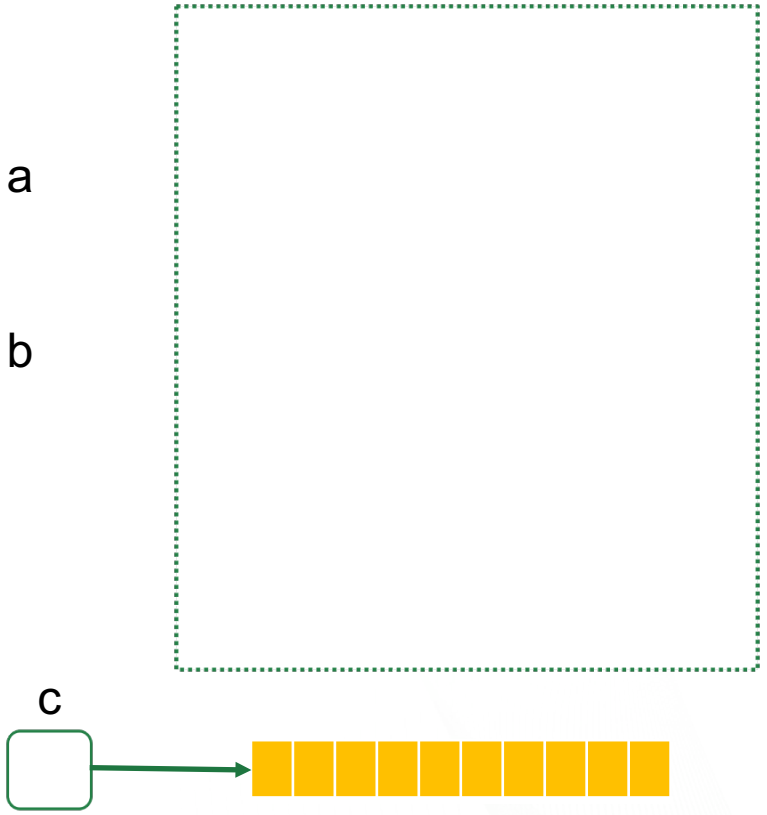
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h,
              N * sizeof(double));
double* b = haven_alloc(h,
              N * sizeof(double));
double* c = malloc(N *
             sizeof(double));

vector_addition(c, a, b);

a = null; b = null;
```



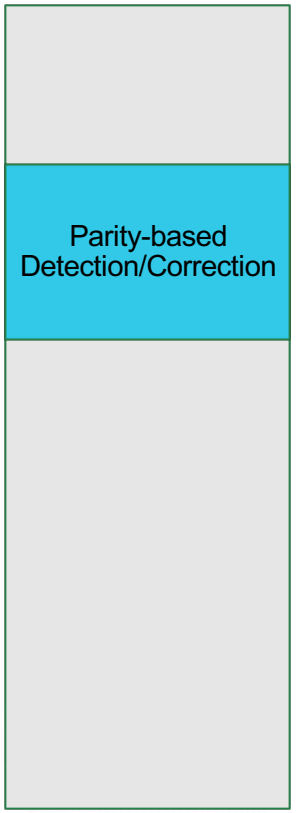
Using Havens: Vector Addition Example

```
Haven h = new_haven();

double* a = haven_alloc(h, N *
sizeof(double));
double* b = haven_alloc(h, N *
sizeof(double));
double* c = malloc(N *
sizeof(double));

vector_addition(c, a, b);

a = null; b = null;

delete_haven();
```



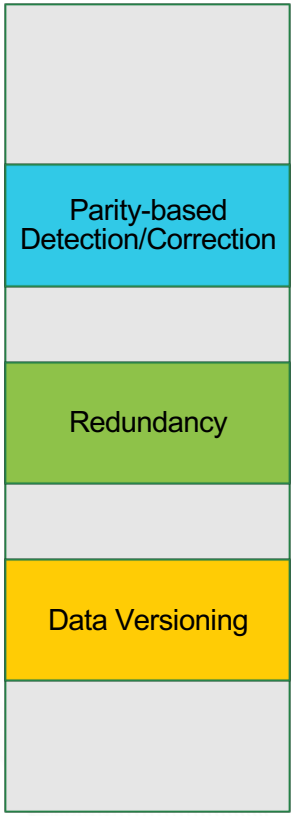
Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



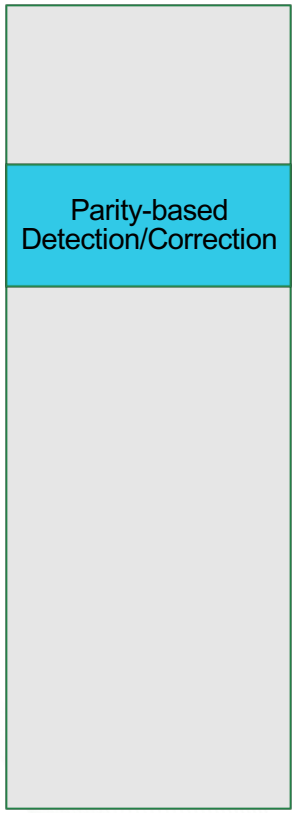
Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



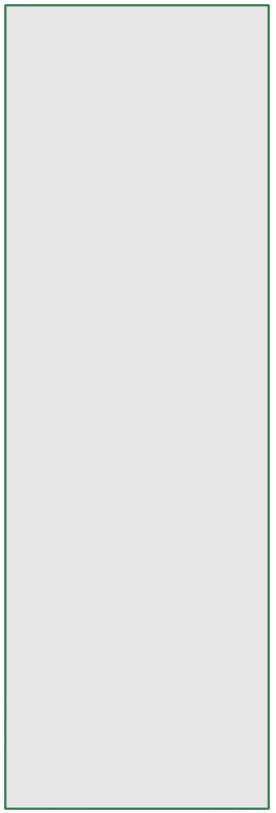
Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



Reliability Models for HPC Applications using Havens

- Selective Reliability
  - Havens provide specific regions of program memory with comprehensive error protection
- Specialized Reliability
  - Various havens in program memory may be protected using different protection scheme based on needs of application, hardware-supported capabilities.
- Reliability v Performance Trade-off
  - Reliability feature for a specific memory regions can be enabled/disabled.



Goals for Language-based Support

- **Explicit**
  - HPC programmers control where their program objects are allocated and their robustness characteristic and lifetime.
- **Convenient**
  - The need for a minimal set of explicit programmer annotations while supporting many C/C++ idioms.
- **Sound**
  - The language annotations must prevent dangling-pointer dereferences and space leaks.
- **Scalable**
  - The havens support various object types and performance overhead of the resilience scheme scales well.

Language Support: Features for Havens

- Expressiveness
  - Structured memory management
  - Limits number of code changes
- Safety
  - Dangling references to objects
- Optimizations for resilience schemes
  - Separation between resilience scheme and memory management

Haven: Static Type Annotations

- *Basic Idea:* specify haven names in types
- Haven-based type system:
  - τ => object type, haven identifier
- Type annotations make the structure of an application's memory management more explicit
- Haven pointers:
  - havenptr is a type of “smart” pointer object for a haven
  - Maintains bookkeeping information about the objects resident to the haven (sizes, reference count, etc.)
  - Enables support for dynamic data structures in havens
  - Dynamic optimization of error protection scheme

Haven Type System

- **type<havenptr>**
  - Subtype for non-pointer variables that guarantees the allocation of the qualified object within a haven
  - Default error detection/correction scheme of haven applied to object
  - Type provides a method to qualify local and global variables in C/C++ programs
- **type*<havenptr>**
  - Subtype may be applied to pointer objects
  - Permits the application to imply locality for objects
  - Allows aggregation of several small objects to a single haven

Dynamic Regions

- **haven h {s}**
  - *h* is a haven handle identifier
  - statement *s* (may be a compound statement)
  - {…} may contain code blocks
  - haven's lifetime is the execution of statement *s*
- Example Use Case: Selectively reliable computation e.g., FT-GMRES

Experimental Evaluation: Conjugate Gradient

Selective Reliability Model:
Key Data Structures considered for placement in havens:

- matrix A
- vector b
- vector x (solution, initialized to random values)
- matrix M (preconditioner)
- vector r (auxiliary vector)
- vector p (auxiliary vector)

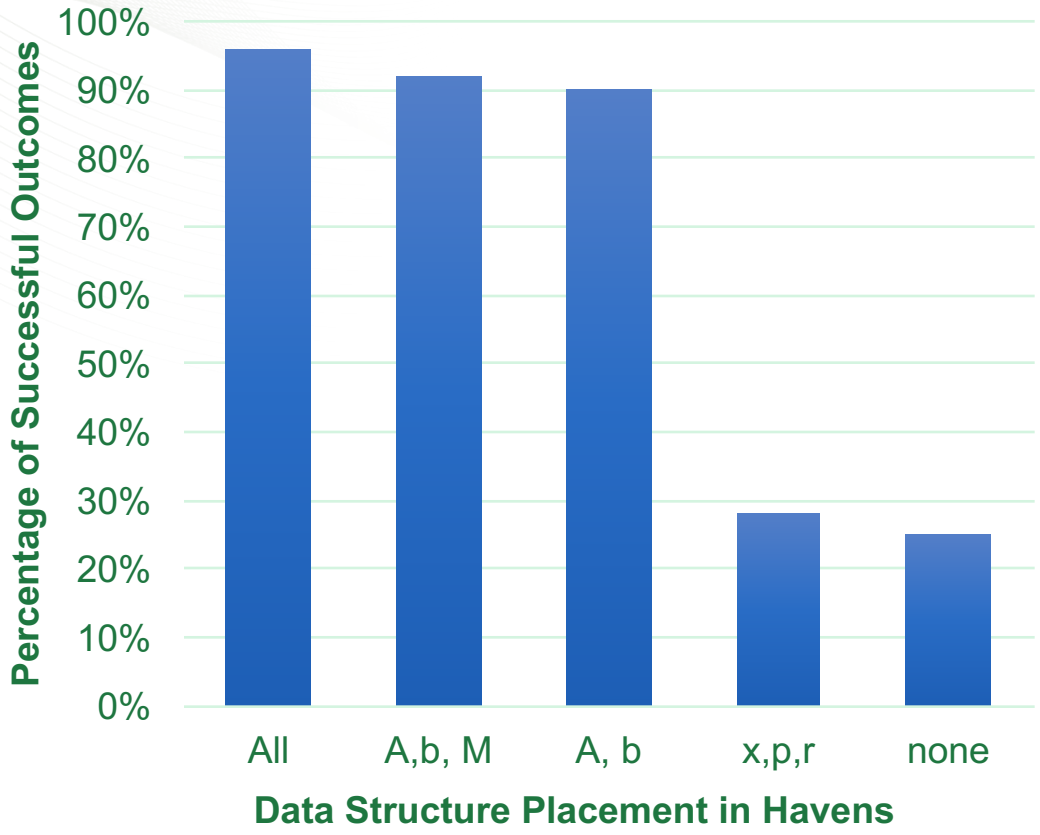
Fault Injection Experiments:

- Objects allocated selectively in havens
- Multiple fault injections per CG run
  - Monte Carlo fault injection

$r_0 = b - Ax_0$
$z_0 = M^{-1} r_0$
$p_0 = z_0$
$k = 0$
procedure REPEAT(A, M, r, z, p, x)
  $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  if r_{k+1} is sufficiently small **then**
    exit loop
  end if
  $z_{k+1} = M^{-1} z_{k+1}$
  $\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$
  $k = k + 1$
end procedure

Algorithm: Preconditioned Conjugate Gradient

Results: Selective Placement



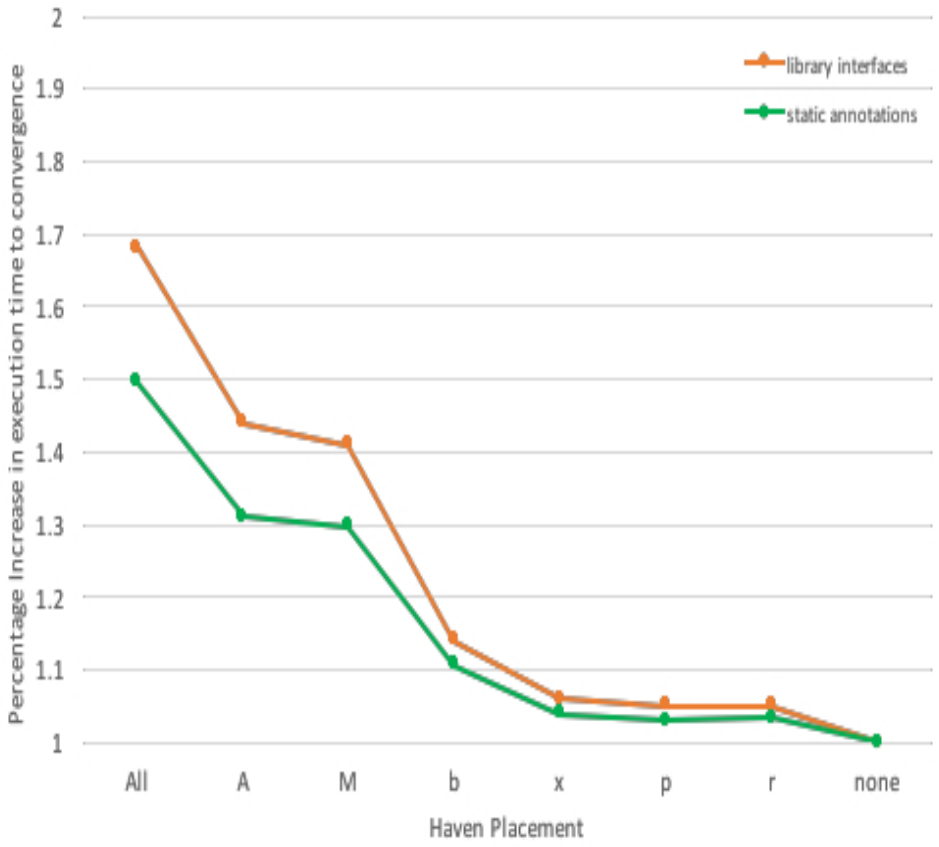
Higher Fault Coverage ≠ Higher Application Resilience

Placement guided by type of program state:

Static state, Dynamic state, Computational environment:

- matrix A, vector B higher error sensitivity:
  - errors in the operand matrix A or vector b fundamentally changes the linear system being solved
- Protecting the static state (operand matrix A, vector b, matrix M) provides higher application resilience
- Removing M from the protection domain of havens makes only a small difference to application resilience

Results: Performance Overhead

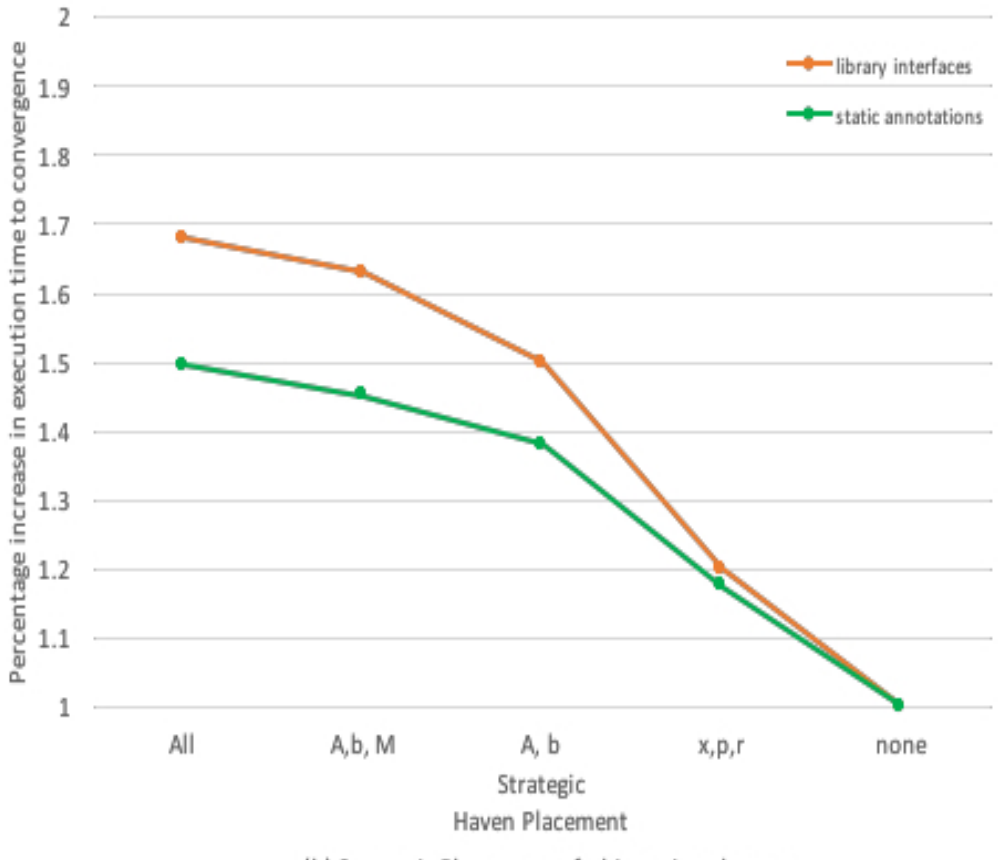


Individual Allocation of CG objects into Havens

Baseline: memory allocations using standard malloc/free interfaces

- The performance overhead generally tracks the memory footprint of the program objects
- matrix A occupies dominant part of the solver's memory
  - Over 50% of the active address space
- preconditioner matrix M, vectors x, p, r demonstrate lower sensitivity to errors
  - errors typically lead to performance degradation, but solver converges.

Results: Performance Overhead



Placement guided by type of variable state

- Naïve placement of **all** objects into havens proves very expensive given their impact on overall application resilience.
- Protecting the static state (operand matrix A, vector b, matrix M) provides higher application resilience
- Haven typing system enables optimizations of software-based resilience scheme.

Summary

- HPC node and system architectures are changing
- Need for software techniques to navigate heterogeneity & complexity
- Havens: Memory management with reliability as a prominent feature
  - Allocation of program objects based on their individual error sensitivity
  - Each haven may have separate protection scheme
  - Allows complementing hardware-level solutions
- Language Support:
  - Region expressiveness adds structure to memory management
  - Enables optimization of resilience techniques

Questions?