Language Support for Reliable Memory Regions

Saurabh Hukerikar, Christian Engelmann

Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {hukerikarsr, engelmann}@ornl.gov

Abstract. The path to exascale computational capabilities in high-performance computing (HPC) systems is challenged by the evolution of the architectures of supercomputing systems. The constraints of power have driven designs to include increasingly heterogeneous architectures and more complex memory hierarchies. These systems are also expected to experience in an increased rate of errors, such that the applications will no longer be able to assume correct behavior of the underlying machine. To enable the scientific community to succeed in scaling their applications and harness the capabilities of exascale systems, we need software strategies that provide mechanisms for explicit management of resilience to errors in the system, in addition to locality of reference.

In prior work, we introduced the concept of explicitly reliable memory regions, called *havens*. Memory management using havens supports selective reliability through a region-based approach to memory allocation. Havens enable the creation of software-enabled robust memory regions for which resilient behavior is guaranteed. In this paper, we propose language support for havens through type annotations that make the structure of a program's havens more explicit and convenient for HPC programmers to use. We describe how the extended haven-based memory management model is implemented, and demonstrate the use of the static type annotations to affect the resiliency of a conjugate gradient application.

This work was sponsored by the U.S. Department of Energy's Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

1 Introduction

The high-performance computing (HPC) community has their sights set on exascale-class computers, but there remain several challenges in designing these systems and preparing application software to harness the extreme-scale parallelism. Due to constraints of power, emerging HPC system architectures will employ radically different node and system architectures. Future architectures will emphasize increasing on-chip and node-level parallelism, in addition to scaling the number of nodes in the system, in order to drive performance while meeting the constraints of power [1]. The technology trends suggest that memory architectures are expected to have lower memory capacity and bandwidth per flop of compute performance. Therefore, memory architectures will be more complex, with denser memory hierarchies and utilize more diverse memory technologies [2]. The management of resilience to the occurrence of frequent faults and errors in the system has also been identified as a critical challenge [3]. HPC applications and their algorithms will need to adapt to these evolving architectures that will also be increasingly unreliable. These challenges have led to suggestions that our existing approaches to programming models must change to complement existing system-level approaches [4]. The demands for massive concurrency and the emergence of high fault rates require that programming model features also support the management of resilience and data locality in order to achieve high performance.

Recent efforts in the HPC community have focused on improvements in the scalability of numerical libraries and implementations of the MPI libraries for these to be useful on future extreme-scale machines. However, there is also a need to develop new abstractions and methods to support fault resilience. In prior work, we proposed a resilience-driven approach to memory management using havens [5]. Havens offer an explicit method for affecting resilience in the context of memory management. In haven-based memory management, each allocated object is placed in a program-specified haven. The havens guarantee a specified level of robustness for all the program objects contained in a memory region. However, the objects contained in havens may not be freed individually; instead the entire haven is deallocated, leading to the deletion of all the contained objects. Each haven is protected by a detection/correction mechanism, and different havens in a program may be protected using different resilience schemes. HPC applications may write their own disciplines to manipulate the resilience features of arbitrary types of memory. The use of havens provides structure to resiliency management of the program memory by grouping related objects based on the objects' individual need for robustness and the performance overhead of the resilience mechanism. While traditional region-based systems [6] statically assign program objects to regions based on compiler analysis to eliminate the need for runtime garbage collection, havens provide a scheme for creating regions within heap-allocated memory with distinct robustness features. In our initial design, we defined interfaces for the creation and use of havens that were implemented by a library interface. In this paper, we develop language support in

order to make havens in HPC application programs clearer and more convenient to use, and to support as many C/C++ language constructs as possible.

This paper makes the following contributions:

- We make a realistic proposal for adding language support for havens to mainstream HPC languages.
- We develop type annotations, which enable static encoding of the program object's allocation and deallocation into the robust regions, and optimize the trade-off between the robustness and performance overhead.
- We investigate how affecting the resilience of the individual program objects using these static annotations affects their fault coverage and performance during application execution.

2 Havens: Reliable Memory Regions

The concept of havens [5] supports resilience-driven memory management. The runtime memory is partitioned into *robust* regions, called havens, into which program objects are allocated. Each memory region is protected by a predefined error detection and/or correction scheme. The robustness scheme is intended to be agnostic to the algorithm features, or to the structure of the data objects placed in havens. There is a clear separation between the memory allocation semantics of havens and their implementation of a robustness scheme. Each haven may be protected using various alternative detection/correction schemes, such as software-based parity, hashing, replication, etc. Since each of these robustness schemes carry a different level of performance overhead the program memory is logically partitioned into regions that each possess a specific level of error resilience and performance overhead.

From the perspective of an HPC application program, havens enable applications to exert fine-grained control on the resilience properties of individual program objects. Since different havens may have varying guarantees of reliability, based on the strength of the protection mechanism and its performance overhead, object placement in havens may be driven by the trade-off between criticality of the object to program correctness and the associated overhead. This creates a logical grouping of objects that require similar resilience characteristics. The objects in any haven are all freed at once by deleting the entire pool of memory. Therefore, havens enable the association of lifetime to the reliable memory regions.

3 Using Havens for Resilience-driven Memory Management

3.1 Basic Operations

While developing the concept of havens, we defined an interface for HPC programs to effectively use the reliable memory regions in their application codes [5]. The abstract interface is based on the notion of a haven manager, which provides

a set of basic operations that must be implemented to fully support the use of havens. The operations are summarized below:

1. `__haven_create__`**:** The request for the creation of a haven by an application returns a handle to the memory region, but no memory is allocated. The specification of the error protection scheme is specified during the haven create operation.
2. `__haven_alloc__`**:** An application requests a specified block of memory within a haven using this interface. This operation results in the allocation of the memory and the initialization of state related to the protection scheme.
3. `__haven_delete__`**:** The interface indicates intent to delete an object within the haven, but the memory is not released until the haven is destroyed.
4. `__haven_read__` and `__haven_write__`**:** These interfaces read and update the program objects contained in the haven; the operations are performed through these interfaces, rather than directly on the objects, to enable the haven manager to maintain updated state about the robustness mechanism.
5. `__haven_destroy__`**:** The interface requests that the haven is destroyed, which results in all memory blocks allocated in the region to be deallocated and the memory is available for reuse. Upon completion of this operation, no further operation on the haven are permitted. The state related to the robustness scheme maintained by the haven manager is also destroyed.
6. `__haven_relax__` and `__haven_robust__`**:** These interfaces enable the error protection scheme applied to a haven to be turned on and off based on the needs of the application program.

3.2 Haven Library Interface

The implementation of the havens library is similar to the one implemented in [5], in which the heap is divided into fixed-size pages, and each new haven creation is aligned on a page-size boundary. The library maintains a linked list of these pages. We provide the library API functions for each of the primitives that enable basic haven operations: the `haven_malloc()` and `haven_new()` implement the abstraction `__haven_create__` for the allocation of objects into the associated region. The implementation of the haven system imposes no changes on the representation of pointers and permits the access to individual objects in the havens using pointer operations. However, we only support per-region allocation and deallocation, and therefore per-object deallocation is an illegal operation. The `haven_release()` enables expressing the end of object life. However, the `haven_destroy()` operation must be invoked to release the memory by concatenating the haven's page list to the global list of free pages.

3.3 Protection Schemes for Havens

In our initial implementation of havens, the memory regions are guaranteed highly-reliable behavior through comprehensive error protection based on a software-based lightweight parity protection scheme. The haven library maintains a pair

of correction signatures for each memory region, which are of word length and an additional word length detection signature per 64 words in the memory block in the region. The detection signature contains one parity bit per word in the memory region. As memory is allocated for the region and initialized, the correction signature S1 retains the XOR of all words that are written to the memory region. We apply an XOR operation on every word that is updated in the memory region and the correction signature S2. Silent data corruptions or multi-bit errors are detected based on the state of the detection signature containing a parity bit for each word contained in the memory region.

When the parity signature for a memory location in the haven detects a parity violation, the location of the corrupted memory word may be identified. The value at the corrupted memory location may be recovered using the XOR signature words. The XOR of the two signatures S1 and S2 equals the XOR of all the uncorrupted locations in the haven. The corrupted value in the memory region is recovered by performing an XOR operation on the remaining words in the haven with the XOR of the two signatures S1 and S2. The recovered value overwrites the corrupted value, and the detection signature is recomputed. Using this correction scheme, multibit corruptions may be recovered from unlike hardware-based ECC, which offers single bit error detection and double bit error correction. Each of these of detection/recovery operations are transparent to the application. This parity-based protection is an adaptation of an erasure code and it maintains very limited state for the detection and correction capabilities and therefore carries very little space overhead in comparison to other software-based schemes such as software-based ECC and checksums. The detection is a constant time operation while the recovery is a O(n) operation based on the size of the haven.

4 A Haven Type System

In traditional region-based memory management systems, region inference is used to determine the creation and destruction of regions, as well as the allocation operations for each region using compiler analysis. With automatic region inference, the application programmer does not need to include explicit memory management idioms in the program code. However, from the perspective of supporting resilient behavior for memory regions, compiler-based analyses provide little insight into the error sensitivities of the program objects. Therefore, convenient interfaces are necessary for havens to be adopted in the development of HPC applications as well as their use in legacy codes. The prototype implementation contained library interfaces for each of the basic haven operations. In this paper, we focus on developing language-based annotations with emphasis on making the use of haven-based memory management convenient, enabling resilience and performance overhead to be tuned, and preventing dangling-pointer dereferences. Our design of the haven type system aims to address the following seemingly conflicting goals:

– 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.
– Soundness: 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.

We develop a typing system that enables HPC programmers to statically encode memory management decisions on the basis of their understanding of the resiliency requirements of the various program objects. By making the structure of the havens and their resilience schemes explicit, the overheads of the schemes are optimized.

4.1 Type Annotations for Havens

The current implementation of havens is based on a source-to-source compiler and a runtime library that together support the haven API. The type annotations that support havens are:

– `haven_ptr` is a type of smart pointer object for a haven. In addition to the pointer reference of the haven, the `haven_ptr` maintains bookkeeping information about the objects resident to the haven, including their sizes and a reference count. The size of individual objects within the haven enables the library to optimize the parity-based protection scheme that provides error detection and correction for the memory. Based on the size of the individual memory objects allocated, the library decides on the number of signatures per memory object.
– `type<haven_ptr>`: is a subtype for non-pointer variables that guarantees the allocation of the qualified object within a haven and its protection using a detection/correction scheme. The qualifier provides a method to qualify local variables and global variables in C/C++ programs. The qualifier creates a pointer to the object; the pointer is a class template that is declared on the stack, and is initialized with a reference that points to the object, which itself is allocated on the heap.
– `type*<haven_ptr>` subtype may be applied to pointer objects as well. Based on the included `haven_ptr`, this permits the application to imply locality for the object whose pointer it qualifies. This is particularly useful for the allocation of several small objects to a single haven, which helps amortize the overhead of employing parity signatures for single objects, by sharing the signatures among multiple separate objects that are resident to a haven.
– `haven h{s}` enables the creation of dynamic havens that are created with the construct `haven h{s}`, where h is a haven handle identifier, and s is a statement (that may be a compound statement). The haven's lifetime is the execution of s. In s, the h is bound to the haven handle, which allocation primitives may use to allocate objects into the associated reliable region.

– `deletehaven` operator: provides a static mechanism to deallocate the memory block corresponding to the haven, and which is pointed to by the `haven_ptr` type pointer object. If the `haven_ptr` is contains all null object pointers, the operation results in releasing the storage space for the haven, along with the program objects contained in the haven.

4.2 Syntax

The declaration of a `haven_ptr` typed pointer leads to the creation of a haven. The creation, however, does not allocate the vector in the example shown below. The `robust<haven_ptr>` type qualifier for the declaration of the vector object associates the object memory with the haven. When the `haven_alloc` allocation request is made, the runtime library initializes the parity signatures for the vector object itself.

```
haven_ptr h1;
double*<h1> vectorA;
vectorA = (double*)haven_malloc(nn*sizeof(double));

for(i=0; i<nn; ++i)
/* initialize vector */

/* vector operations */

haven_release(matrixA);
deletehaven h1;
```

The `deletehaven` operation deletes the haven completely and destroys the state maintained for the parity protection. The safety of this operation is guaranteed through the reference counting included in the `haven_ptr`.

4.3 Reference Counting

With the introduction of the `haven_ptr`, we also address one of the major shortcomings of our prior implementation: the lack of safety. In the initial havens proposal, it was possible to delete a haven without regard for the pointers to objects allocated inside the region. The deletion of a haven potentially left dangling pointers. With the use of `haven_ptr`, the pointer maintains a reference count of the number of external pointers to objects in each haven. The `deletehaven` operator placed in an application code is a safe operation that checks for the presence of valid pointers within the haven. When deleting a haven, there is runtime check for any active references to the memory objects that are contained in the haven. The delete operation fails if there are any active references.

5 Experimental Results

To apply the static annotations in an HPC application, we identify program objects that must be allocated in havens, and annotate their declarations with

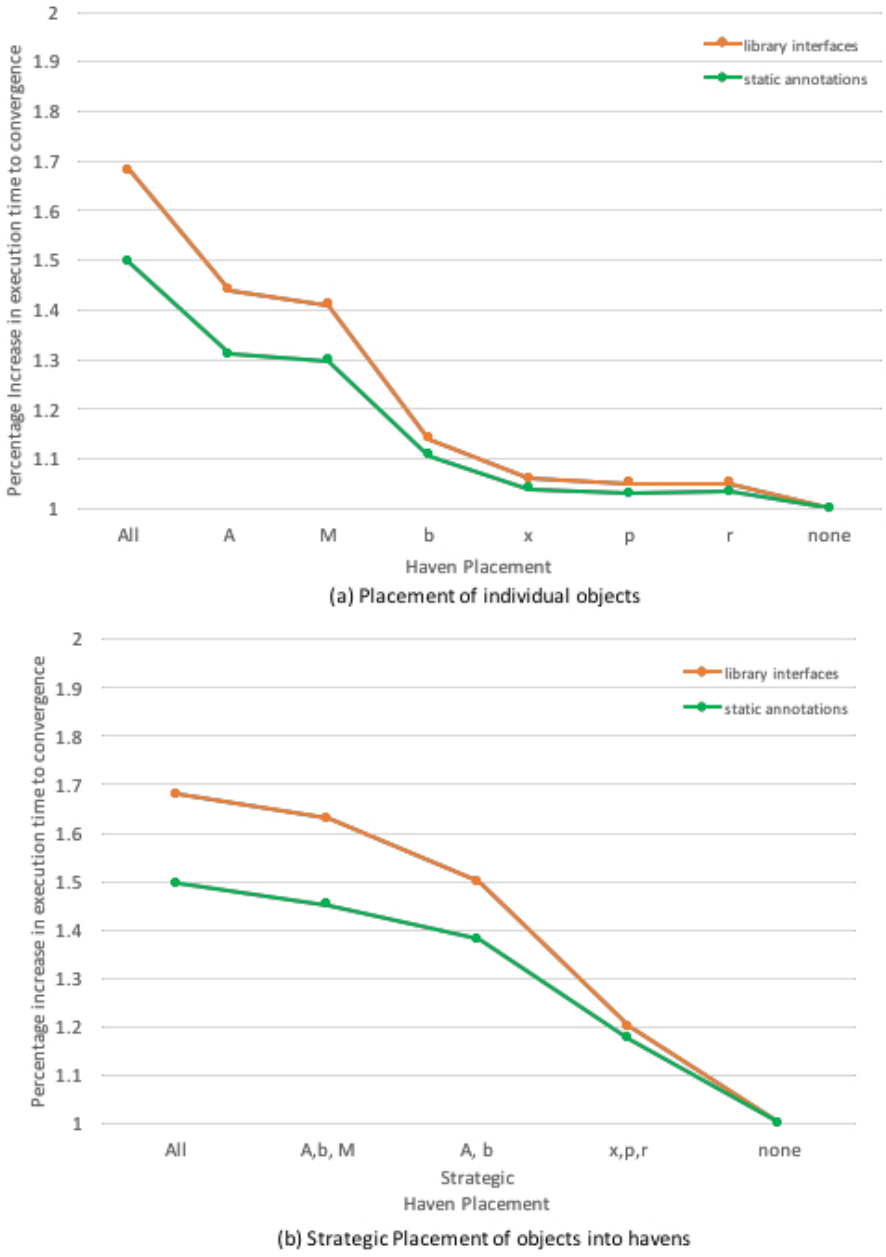

Fig. 1. Performance overheads of havens with static annotations

the type qualifiers. For these experiments that evaluate the use of haven-based memory management using the type qualifiers, we modify a conjugate gradient code to include the qualifiers on the various application objects. We use a pre-conditioned iterative CG algorithm and we validate the correctness of the outcome of the solver with a solution produced using a direct solver. We compare the evaluation with the results from our previous implementation that required insertion of raw library interfaces. One of the key advantages of using the static annotations is that the lines of code change is reduced significantly when compared to the changes required for insertion of library calls in the same application code.

In the CG algorithm, which solves a system of linear equations A.x = b in which the algorithm allocates the matrix A, the vector b and the solution vector x. Additionally, the conjugate vectors p and the residual vector r are referenced during each iteration of the algorithm. We perform performance evaluation experiments in which the various object in the CG are allocated using havens. We perform two sets of experiments: (i) we allocate only one structure using the haven static annotations, while the remaining structures are allocated using the standard memory allocation interfaces; (ii) we strategically place the data structures of the CG by allocating structures to havens using this classification. We compare these strategies with allocations in which havens provide complete coverage and with experimental runs which do not allocate any structure using havens. We evaluate the following combinations: (i) allocation of only the static state, i.e., the matrix A and vector B, the preconditioner M into havens, while the dynamic state, i.e., all the solution vectors, are allocated using standard memory allocation functions; (ii) allocation of only matrix A and vector B into havens; (iii) only the dynamic state is provided fault coverage using havens.

The performance overhead of using havens to the convergence time of the CG code for the above selection of program objects for allocation into havens is shown in Figure 1. The annotation of all the program variables to be allocated into havens provides higher fault coverage, but it results in higher overhead to the time to solution for the CG application. When the variables are allocated using raw library interfaces, each program object is protected by a pair of signatures. When these objects are qualified with the static annotations in the application code, the compiler and library have a better understanding of the size and structure of the program objects. Therefore, the larger program objects, notably the operand matrix A and the preconditioner matrix M, are split and protected by multiple pairs of parity signatures. This split protection is transparent to the application programmer and the application still accesses the matrix elements as a single data structure. The use of multiple signatures improves the read/write overhead for the objects and the observed overhead with static annotations for all program objects is 11% lower than with the library-based allocation, which provides monolithic protection for the entire data structure. The operand matrix A occupies a dominant part of the solver's memory, occupying over 50% of the active address space, whereas the solution vector x, the conjugate vectors p and the residual vector r and the preconditioner matrix M account for the remaining

space. Therefore, the annotation of matrix A individually results in 9% lower overhead than with monolithic parity protection using library interfaces. The improvement in performance when smaller data objects are annotated is within 2% of the version using library interfaces.

The program objects in the CG application demonstrate different sensitivities to errors. Errors in the operand matrix A or vector b fundamentally changes the linear system being solved. For errors in these structures even if the CG solver converges to a solution, it may be significantly different from a correct solution. The preconditioner matrix M demonstrates lower sensitivity to the errors, as do the vectors x, p, r. These features of the CG algorithm form the basis for the strategic placement of the objects into havens, since the allocation of only sensitive data structures into havens provides a substantially higher resilient behavior in terms of completion rates of the CG algorithm for reasonable overheads to performance than a naive placement strategy.

6 Related Work

Much research has been devoted to studies of algorithms for memory management based on garbage collection or explicit deallocation. The concept of regions was implemented in a storage package, in which objects may be allocated in specific zones [7]. While each zone permitted a different allocation policy, the deallocation was performed on a per-object basis. The vmalloc library [8] provides programmers a method to manage memory allocation and impose different policies on each memory allocation. Region-based systems such as arenas [9] enable writing special purpose memory allocators that offer better performance heap memory allocation for specific applications. Several early implementations of region-based systems were unsafe; the deletion of regions often left dangling pointers that were subsequently accessed. Such safety concerns were addressed through reference counting schemes for the regions [10].

For dynamic heap memory management through static analysis, regions were proposed [6] as an alternative to garbage collection methods, in order to provide more predictable and lower memory space. In this proposal, the assignment of program objects to regions was statically directed by the compiler; this concept was refined [11] by relaxing the requirement for region lifetimes to be lexical. Support for regions was implemented in ML [12], Prolog [13]. Cyclone was a language designed to be syntactically very close to C, and which provided support for regions through an explicit typing system [14]. While most of these implementations of the concept were in the context of declarative programming languages.

Implementations such as vmalloc place the burden of determining policy of allocation of objects to regions on the programmer [8]. Other schemes have used profiling to identify allocations that are short-lived and place such allocations in fixed-size regions [15]. Our previous work on *havens* [5] provided a reliability-driven method for memory allocations. Recent efforts seek provide programming model support for reliability, such as containment domains [16], which offer pro-

gramming constructs that impose transactional semantics for specific computations. Rolex [17] offers language-based extensions that support various resilience semantics on application data and computations. Global View Resilience (GVR) supports reliability of application data by providing an interface for applications to maintain version-based snapshots of the application data [18]. In support of fault tolerance of dynamic memory allocation, the `malloc_failable` may be used by the application to allocate memory on the heap; callback functions are used to handle error recovery for the memory block [19].

7 Conclusion

With the evolution of HPC architectures, and the emergence of reliability as a major concern, applications must contend with complex memory hierarchies, which may possess different levels of hardware-based reliability protection schemes. Havens provide a software-based approach for HPC applications to manage the reliability of their programs. Through explicit, robust region-based memory management, the HPC application programmers may control the level of robustness and resilience of individual memory regions in their applications. In this paper, we focused on providing static typing discipline for these robust regions. We demonstrated that the incorporation of some static information about an application program's region structure makes the structure of the program's memory and its robustness requirements more explicit. This enables the resilience of the associated memory regions to be managed at compile-time and runtime. These annotations to C/C++ HPC application codes are not required, but they permit encoding the resilience requirements in heap memory-management idioms. We also demonstrated how stack-based memory allocation discipline is affected by the use of type qualifiers.

References

1. Shalf, J., Dosanjh, S., Morrison, J.: Exascale computing technology challenges. In: Proceedings of the 9th International Conference on High Performance Computing for Computational Science. VECPAR'10, Springer-Verlag (2011) 1–25
2. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dallya, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R.S., Yelick, K.: Exascale Computing Study: Technology Challenges in Achieving Exascale systems. Technical report, DARPA (September 2008)
3. DeBardeleben, N., Laros, J., Daly, J., Scott, S., Engelmann, C., Harrod, B.: High-End Computing Resilience: Analysis of issues facing the hec community and path-forward for research and development. Whitepaper (December 2009)
4. Amarasinghe, S., Hall, M., Lethin, R., Pingali, K., Quinlan, D., Sarkar, V., Shalf, J., Lucas, R., Yelick, K., Balaji, P., Diniz, P.C., Koniges, A., Snir, M., Sachs, S.R., Yelick, K.: Exascale Programming Challenges: Report of the 2011 workshop on exascale programming challenges. Technical report, U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR) (July 2011)

5. Hukerikar, S., Engelmann, C.: Havens: Explicit reliable memory regions for hpc applications. In: IEEE High Performance Extreme Computing Conference (HPEC). (September 2016) 1–6
6. Tofte, M., Talpin, J.P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '94, New York, NY, USA, ACM (1994) 188–201
7. Ross, D.T.: The aed free storage package. Communications of ACM **10**(8) (August 1967) 481–492
8. Vo, K.P.: Vmalloc: A general and efficient memory allocator. Software: Practice and Experience **26**(3) (1996) 357–374
9. Hanson, D.R.: Fast allocation and deallocation of memory based on object lifetimes. Softwre Practices & Experience **20**(1) (January 1990) 5–12
10. Gay, D., Aiken, A.: Memory management with explicit regions. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. PLDI '98, New York, NY, USA, ACM (1998) 313–323
11. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: Improving region-based analysis of higher-order languages. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation. PLDI '95 (1995) 174–185
12. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P., Bertelsen, P.: Programming with regions in the ml kit, technical report (diku-tr-97/12). Technical report, University of Copenhagen, Denmark (April 1997)
13. Makholm, H.: A region-based memory manager for prolog. In: Proceedings of the 2nd International Symposium on Memory Management. ISMM '00, New York, NY, USA, ACM (2000) 25–34
14. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. PLDI '02, New York, NY, USA, ACM (2002) 282–293
15. Barrett, D.A., Zorn, B.G.: Using lifetime predictors to improve memory allocation performance. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. PLDI '93, New York, NY, USA (1993) 187–196
16. Chung, J., Lee, I., Sullivan, M., Ryoo, J.H., Kim, D.W., Yoon, D.H., Kaplan, L., Erez, M.: 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. (2012) 58:1–58:11
17. Hukerikar, S., Lucas, R.F.: Rolex: Resilience-oriented language extensions for extreme-scale systems. The Journal of Supercomputing (2016) 1–33
18. Chien, A., Balaji, P., Beckman, P., Dun, N., Fang, A., Fujita, H., Iskra, K., Rubenstein, Z., Zheng, Z., Schreiber, R., Hammond, J., Dinan, J., Laguna, I., Richards, D., Dubey, A., van Straalen, B., Hoemmen, M., Heroux, M., Teranishi, K., Siegel, A.: Versioned distributed arrays for resilience in scientific applications: Global view resilience. Procedia Computer Science **51** (2015) 29 – 38
19. Bridges, P.G., Hoemmen, M., Ferreira, K.B., Heroux, M.A., Soltero, P., Brightwell, R.: Cooperative application/os dram fault recovery. In: Proceedings of the 2011 International Conference on Parallel Processing - Volume 2. Euro-Par'11, Springer-Verlag (2011) 241–250