Survival Strategies for Synthesized Hardware Systems

Martin Rinard
Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

***Abstract*—Survival is a key concern of many complex systems. A standard approach to maximizing the likelihood of survival is to attempt to produce a system that is as free of errors as possible. We instead propose a methodology that changes the semantics of the underlying development and execution environments to cleanly and simply obtain survival guarantees that are difficult if not impossible to obtain with standard techniques. Examples of survival properties include continued execution in the face of addressing errors and guaranteed bounds on the amount of memory required during any execution of the system (even in the face of dynamic memory allocation). We summarize results for software implementations of these techniques and discuss issues and advantages that arise in the context of hardware implementations.**

***Keywords*-Reliability, Testing, and Fault Tolerance; Automatic Synthesis; Verification; Test Generation; Validation; Reliability; Risk Management; Error Processing; Storage Management; High Availability; Semantics of Programming Languages**

I. INTRODUCTION

It is common knowledge that all large deployed systems contain many errors [1]. Despite these errors, most of these systems provide acceptable service to their users. This is true even though the underlying development and execution environments (specifically, the programming language implementation, library design, and advocated software engineering practice) are extremely brittle — at the first sign of an error (such as an exception, invalid library call, or assertion violation) mechanisms built into the environment typically terminate the execution. Note that, given current engineering realities, a substantial level of resilience in the face of errors is a necessary prerequisite for the economically feasible construction of large systems. Any realistic attempt to build completely error-free systems on a broad scale would immediately restrict the scope to include only systems with unacceptably limited functionality.

Interestingly enough, there is evidence that some simple changes to the development and execution environment can substantially improve system robustness in the face of otherwise fatal errors, both naturally occurring and artificially injected [2], [3], [4], [5], [6], [7], [8], [9], [10]. This is true despite the fact (or perhaps even because of the fact) that the original developers were unaware of these changes and, indeed, developed the system using the original brittle environments.

These empirical results suggest that deployed systems can tolerate significant changes to the underlying development and execution environments and still execute (in many cases even more) acceptably. The possibility of changing these environments broadens the system design space, which may, in turn, make otherwise very challenging system goals easily attainable.

II. SURVIVAL

For many complex systems survival (i.e., continuing to execute to provide as much service as possible) is a primary concern. In past research, we have analyzed the classes of fatal errors in software systems and, for each class, developed a dynamic technique that enables the system to survive errors in that class [2], [3], [7]. Each such technique is applied automatically and uniformly across the entire system at the level of the underlying development and execution systems. They require no effort on the part of the original developers — indeed, the developers typically remain completely oblivious to these techniques. The application of these techniques makes it possible to develop *immortal* systems that survive *any* error in the software that implements the system. We discuss three such techniques:

- **Safe Memory Block Accessing:** Many programs use computed expressions to access arrays and pointer arithmetic to compute addresses for accessed memory blocks. In both cases, it is possible for out of bounds accesses to cause the program to fail. We have implemented three ways to avoid this failure:
  - **Failure-Oblivious Computing:** In failure-oblivious computing [2], the program contains checks for out of bounds memory accesses. It discards out of bounds writes and manufactures values for out of bounds reads. The experimental results indicate that this technique enables servers to survive otherwise fatal memory errors to continue on to correctly process subsequent requests.
  - **Cyclic Memory Block Addressing:** A variant on failure-oblivious computing, cyclic memory block addressing, maps reads and writes cyclically onto the addressed memory block. So out of bounds reads are cyclically remapped (typically via a `mod`

operation) to a corresponding location within the accessed block of memory. Out of bounds writes can either be discarded or cyclically mapped to overwrite older values in the accessed memory block.

In contrast to standard failure-oblivious computing, cyclic memory block accessing has the advantage that groups of out of bounds reads tend to access groups of values that satisfy any underlying data structure consistency constraints. In certain circumstances this consistency preservation may enhance the acceptability of the results that the program produces when it encounters out of bounds memory accesses [4].

– **Boundless Memory Blocks:** Boundless memory block implementations store out of bounds writes in a hash table to be returned on subsequent reads of corresponding addresses [3]. This modification can be seen as converting the size parameter of memory allocation operations from potentially affecting the semantics of the system to only affecting its performance.

In all cases, the programming language semantics determines the memory block granularity. Each array is typically a memory block, as is each (dynamically or statically allocated) object. An important aspect of these policies is the elimination of certain kinds of memory corruption via the confinement of changes to the memory block that the writes were intended to access. The change in memory accessing semantics can be profitably viewed as a change in the semantics of the programming language.

- **Cyclic Memory Allocation:** It is often desirable to obtain, before a system executes, a bound on the amount of memory that the system will attempt to consume. A standard approach to obtaining such a bound is to analyze the dynamic memory allocation behavior of the system to attempt to determine the maximum amount of memory that the system will attempt to allocate.

  Cyclic memory allocation applies a different strategy. Instead of attempting to provide a conceptually new block of memory for each memory allocation, the memory allocator instead provides, for each memory allocation site, a buffer containing a fixed number of object slots of the appropriate size. It then cyclically allocates objects out of the buffer [7]. Specifically, for the nth allocation from a buffer of size b, it allocates the new object in slot $n \texttt{ mod } b$. Note that with this memory allocation strategy, there is an immediate bound on the total amount of memory required to execute the system. Specifically, the total amount is simply the sum, over all allocation sites, of the size of the corresponding cyclic buffer for that site. One way to obtain appropriate buffer sizes is to profile the execution of the system on a representative set of inputs, then use the profiling results to compute the number of live objects in each buffer, then use a cyclic buffer large enough to contain the live objects.

  The potential drawback of this technique is that it has the potential to place multiple live objects in the same buffer slot, thereby overlaying live memory. Our results show that this overlaying can change the behavior of the program, but typically in a way that causes the program to degrade gracefully rather than failing outright. Cyclic memory allocation can therefore be seen as a mechanism for enabling survival with graceful degradation in the face of insufficient resources. Such a graceful degradation may often be preferable to the standard alternative of system failure.
- **Bounded Loops:** Infinite loops, like unbounded memory allocation, can threaten the survival of the system. Infinite loops can consume all of the time the system needs to produce acceptable results; unbounded memory allocation can consume all of the space. It is possible to eliminate infinite loops by simply choosing a maximum number of iterations that each loop is allowed to execute, then exiting the loop and proceeding on to the next activity if the loop attempts to exceed this number of iterations. One way to obtain these maximum numbers of iterations is to profile the execution of the system on appropriate inputs, then use the profiling results to choose appropriate bounds [7].

All of these techniques are simple to implement and provide survival guarantees that are difficult, if not impossible, to obtain using standard techniques, which are constrained by a perceived (and arguably unjustified) need to preserve the standard semantics of the underlying development and execution environments.

III. Potential Hardware Implementations

All of the above survival techniques have been implemented and evaluated in software systems. But for many systems the underlying implementation substrate (hardware or software) is becoming an increasingly fluid choice. There are several advantages to applying two of the three survival techniques listed above in hardware instead of software.

A. *Hardware Safe Memory Block Accessing*

Software implementations of safe memory block accessing in languages without array bounds checks or other out of bounds checks (such as C and C++) typically insert additional instructions into the execution stream to perform the bounds checks. In the absence of optimization, these instructions can degrade the performance of the system [2], [3] (although engineering effort can significantly reduce this overhead [11]). Of course, for languages (such as Java) that already have bounds checks, there is no additional overhead.

A standard way to synthesize hardware implementations of systems that use references to access memory blocks is to first perform an analysis pass that resolves each reference to (ideally) the block of memory that it accesses. If each block of memory is implemented as a distinct hardware resource, each reference is then routed to the corresponding hardware resource that implements the accessed memory block, with an offset into the hardware resource identifying the specific item to access. With a successful static disambiguation of the accessed hardware resource and all offset calculations performed modulo the size of the hardware resource, the synthesized hardware should be incapable of producing an access to any other block of memory. The only remaining issue is to flag offsets whose address calculations have wrapped around to enable the synthesized hardware to drop writes via these offsets.

If the static analysis is unable to disambiguate the reference to a single hardware resource, one way to synthesize the reference is to combine a resource identifier with an offset. In this case the resource identifier chooses the hardware resource to access, with the offset identifying the item within that hardware resource. Because address calculations affect only the offset and not the resource identifier, there is no possibility of the kind of out of bounds memory corruption characteristic of software implementations. Modulo addressing can be implemented in a combination of the offset calculation and (if the resources have different sizes) in an address rectification module placed before each resource that appropriately adjusts out of bounds offsets to fall within the hardware resource. Out of bounds address flags can enable the synthesized circuit to drop out of bounds writes.

Finally, the hardware implementation may place multiple memory blocks in the same hardware resource, with all blocks in the resource accessed via a uniform address space. Like software implementations, the implementation must deal with the possibility that address calculations starting from an address that refers to one memory block may produce an address that refers to another memory block within the same hardware resource. So like software implementations of safe memory blocks, the synthesized hardware implementation must also typically perform bounds checks. In comparison with software implementations, however, hardware implementations have a wider range of implementation choices available to them. An example of a more efficient bounds check implementation implements addresses as combinations of memory block identifiers and offsets (potentially with a translation into the uniform address space of the underlying hardware resource performed just before the access). This alternative resembles standard base plus bounds translation and checking mechanisms present in segmented memory system implementations [12], with further optimizations possible that exploit statically determinable properties of the specific memory blocks in the system. For example, if the memory blocks are identically sized powers of two, it may be possible to eliminate the standard translation from the memory block identifier into the base of the corresponding region in the hardware resource.

B. Cyclic Memory Allocation

In many cases the hardware synthesis system statically preallocates all of the memory blocks, eliminating the need for any memory management as the system executes. In some cases, however, especially when reusing components originally designed for implementation in software, the synthesized hardware implementation may need to manage the allocation and deallocation of memory as the system executes. In this situation key issues include the allocation of one or more hardware resources to hold the allocated blocks of memory, adequately sizing these hardware resources, and devising an algorithm to manage the memory in each hardware resource. These issues are complicated by the fact that standard dynamic memory management approaches work with the abstraction of an unbounded amount of memory.

Cyclic memory allocation (and related techniques that allocate a conceptually unbounded number of memory blocks into a memory of fixed size) can provide a relatively simple solution to the memory allocation problem. The first step is to group the dynamically allocated objects together into classes that are placed in the same hardware resource. There are a variety of options — for example, one could place all objects allocated at the same allocation site together into the same hardware resource. The next step is to profile representative executions to obtain an estimate of the size of the hardware resource required to hold the allocated objects. The profiler can either track allocations and deallocations, or, for more precision, track reads and writes to dynamically compute the number of live objects. It is possible to feed the profiling results back into the assignment of objects to hardware resources by, for example, allocating groups of objects with disjoint lifetimes in the same hardware resource.

Cyclic memory allocation is simple and easy to implement, but works best when the lifetimes of the objects correlate with their allocation order. The broader implementation space available to hardware implementations makes it feasible to contemplate more sophisticated memory management strategies such as least recently used allocation. This strategy would combine hardware instrumentation to track accesses at the granularity of individual allocation units with the allocation of the least recently accessed object slot on each new allocation.

IV. Testing

The sometimes distressing tendency of systems to vary from their anticipated behavior makes testing an essential step in virtually every system development effort. But testing often has a broader purpose — instead of merely checking conformance (or the lack of conformance) with a specification, a primary benefit of many testing efforts is simply to make certain aspects of the system behavior apparent to the developers. The resulting enhanced understanding of the system behavior can then serve as a foundation for developing new functionality or altering the existing specification or implementation.

A. Standard Testing Approaches

Safe memory block access mechanisms and cyclic memory allocation (along with extensions that use more sophisticated allocation strategies within a buffer of fixed size) affect the execution of the system only when the execution goes outside the original anticipated execution envelope. Specifically, in the absence of out of bounds memory accesses, systems that use safe memory block access mechanisms have the same behavior as systems without these mechanisms. Similarly, when the system does not overlay live objects, systems with cyclic memory allocation and its extensions have the same semantics as systems with more standard memory management approaches. Thus, standard testing approaches, while essential for exploring the overall behavior of the system, may not exercise the survival mechanisms at all and may therefore provide little or no insight into the potential consequences of activating these mechanisms.

B. Reduced Resource Testing

The success or failure of these survival techniques depends on the natural resilience of the specific system to which they are applied. Luckily, there is a simple mechanism, *reduced resource testing*, available to explore this natural resilience. Specifically, it is possible to drive the system into activating the anticipated survival mechanisms by artificially reducing the size of allocated memory blocks or cyclic buffers to provoke the system into dropping writes, wrapping out of bounds addresses around back into the appropriate memory block, overlaying live data, or terminating loops early. The resulting behavior of the system under these operating conditions can provide insight into its potential behavior when and if it encounters operating conditions during production that activate the survival mechanisms. We therefore advocate the inclusion of this kind of testing into the overall testing plan for the system.

V. Verification and Modularity

In general, reusable modules can reasonably tolerate higher reliability efforts than complete systems. Any verification costs can be amortized across all uses of the module, reusable modules are typically substantially smaller and easier to verify than complete systems, an explicit specification (which is typically required for any verification effort to succeed) is easier to justify since it can help clients understand the module interface, and (because reusable modules may be used in a different contexts with different reliability requirements) it is difficult to comfortably settle for anything less than total conformance to the specification.

Verified modules typically assume the standard execution semantics. They also typically use an assume/guarantee approach in which the correctness of the module depends on the client using the module correctly. Both of these assumptions undermine the ability of the module to behave acceptably in a broad range of contexts. Specifically, changes in the underlying execution semantics may invalidate the verification reasoning; clients may fail to satisfy the assumptions that the module relies on for its correct execution. We therefore advocate a broader verification approach which relaxes these assumptions. Specifically, we advocate an approach in which the verification produces a guarantee that key properties (such as the consistency of any internal state and an eventual return to correct behavior) are preserved even in the presence of nonstandard execution semantics (such as cyclic memory allocation or other strategies that may overlay live objects). We also advocate the use of *self-defending* modules that make no assumptions whatsoever about how the client uses the module. In effect, self-defending modules attempt to provide an acceptable outcome in the face of all possible client behaviors. In comparison with standard modules, self-defending modules in some cases contain additional error checking to detect and protect themselves against client behavior they are not prepared to tolerate; in other cases they generalize the standard semantics of the module to provide intuitive behavior in the face of possible client behavior that the standard interface assumes does not occur.

VI. Conclusion

Standard development and execution environments deploy unforgiving mechanisms in response to exceptional system behaviors such as out of bounds memory accesses, unbounded memory allocation, and infinite loops. This brittle approach can undermine the ability of the system to execute successfully in the face of unexpected events and errors.

We propose instead to expand the system design space to include changes to the underlying development and execution environments. Such changes alter the standard semantics to cleanly and simply provide guaranteed survival properties such as continued execution and the use of a statically bounded amount of memory. Our experience using these techniques to enable software systems to survive otherwise fatal errors demonstrates the potential benefits of this approach.

In comparison with software implementations of these survival mechanisms, there is a larger design space available to hardware implementations. Hardware synthesis systems can exploit this larger design space to provide hardware systems with the same survival benefits, but with less (and in some cases no) overhead. Reduced resource testing can provide insight into the consequences of deploying the survival mechanisms in the synthesized system.

REFERENCES

[1] C. Jones, *Estimating Software Costs Bringing Realism to Estimating*. McGraw-Hill, 2007.

[2] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *Proceeding of 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, 2004.

[3] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," in *Proceedings of the 2004 Annual Computer Security Applications Conference* , 2004.

[4] M. Rinard, C. Cadar, and H. H. Nguyen, "Exploring the acceptability envelope," in *2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '05 Companion) Onwards! Session*, Oct. 2005.

[5] S. Sidiroglou, O. Laadan, C.-R. Perez, N. Viennot, A. D. Keromytis, and J. Nieh, "ASSURE: Automatic Software Self-healing Using REscue points," in *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[6] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *Proceedings of the 8th Information Security Conference (ISC)*, 2005.

[7] H. H. Nguyen and M. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," in *Proceedings of the 2007 International Symposium on Memory Management*, 2007.

[8] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[9] ——, "Static specification analysis for termination of specification-based data structure repair," in *IEEE International Symposium on Software Reliability*, 2003.

[10] ——, "Data structure repair using goal-directed reasoning," in *Proceedings of the 2005 International Conference on Software Engineering*, 2005.

[11] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proc. 2006 International Conference on Software Engineering*, 2006.

[12] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.