Samurai: Protecting Critical Data in Unsafe Languages

Karthik Pattabiraman*
University of Illinois (Urbana Champaign)
1308 W. Main St
Urbana, IL
pattabir@uiuc.edu

Vinod Grover†
NVIDIA Corporation
2701 San Tomas Expwy.
Santa Clara, CA 95050
vgrover@nvidia.com

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

ABSTRACT

Programs written in type-unsafe languages such as C and C++ incur costly memory errors that result in corrupted data structures, program crashes, and incorrect results. We present a data-centric solution to memory corruption called *critical memory*, a memory model that allows programmers to identify and protect data that is critical for correct program execution. Critical memory defines operations to consistently read and update critical data, and ensures that other non-critical updates in the program will not corrupt it. We also present Samurai, a runtime system that implements critical memory in software. Samurai uses replication and forward error correction to provide probabilistic guarantees of critical memory semantics. Because Samurai does not modify memory operations on non-critical data, the majority of memory operations in programs run at full speed, and Samurai is compatible with third party libraries. Using both applications, including a Web server, and libraries (an STL list class and a memory allocator), we evaluate the performance overhead and fault tolerance that Samurai provides. We find that Samurai is a useful and practical approach for the majority of the applications and libraries considered.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Fault-tolerance; D.3.3 [**Programming Languages**]: Dynamic Storage Management

General Terms

Experimentation, Languages, Reliability

Keywords

critical memory, memory safety, error recovery

*Most of this work was done when the author was an intern with Microsoft Research.

†This work was primarily done when the author was with Microsoft Corporation.

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

1. INTRODUCTION

Languages such as C and C++ do not provide intrinsic guarantees about memory safety that are present in type-safe languages such as Java. Many programs are still written using these languages for performance and compatibility reasons and, as a result, memory errors continue to be common causes of program failures and security vulnerabilities (e.g., CERT [30]). In this paper, we focus on an important subclass of memory errors, *memory corruptions*, although limited protection is offered from other subclasses of memory errors as well (e.g., uninitialized reads and invalid/double frees). Memory corruption occurs when a program breaks type safety and writes to an unintended location, potentially corrupting the data there. Common causes of memory corruptions include buffer overruns and dangling pointer errors.

Because all memory locations are equally accessible to all store instructions in a program, current approaches to providing safety from memory corruption in C and C++ require that every store in a program be either statically or dynamically checked for correctness (e.g., [8, 15, 16, 23]). As a result, there are significant challenges to existing approaches that limit their practical application. For dynamic approaches, the overhead of checking at every store is high (ranging from 2x to 30x overhead) [12, 16, 24]. For static approaches, it is difficult to reason about program components that are not available statically. For example, libraries can be loaded dynamically, and third-party libraries are often not available in source form [23]. A single unchecked pointer write in the program can void the guarantees that static analysis provides.

Figure 1 illustrates the challenges both static and dynamic techniques face in providing safety from corruptions. In the example, we focus on the value of the variable `balance`, which might represent a bank account balance. Preventing the value of `balance` from being illegally modified is difficult with existing methods. In this paper, we present an alternate approach to reducing the impact of memory corruptions on C and C++ programs. Our main contributions are as follows:

- *We define a new memory model,* **Critical Memory (CM)***, that prevents arbitrary stores from corrupting critical data.* Critical data is explicitly identified by the programmer as being vital for correct program execution. In our example, the variable `balance` would be identified as critical. *Distinguishing critical data enables local reasoning about safety from memory corruption*, so that the programmer writing `check_balance` does not need to know if the argument `i` is within the

```
int x, y, buffer[10];
// balance is critical
int balance = 100;

void check_balance(int i) {
    GUI_action(&x, &y);
    buffer[i] = 10000;
    if (balance < 0)
       check_credit();
}
```

Figure 1: Example unsafe C program

bounds of array `buffer`, or that the library function `GUI_action` does not corrupt memory. Critical memory allows the use of static analysis techniques on a subset of a larger unsafe program while ensuring that external libraries (or other functions) cannot corrupt local data. Critical memory is an abstract model that can be implemented in either hardware or software.

- *We describe Samurai, an object-based implementation of critical memory in software.* Samurai provides for critical data, probabilistic guarantees of safety from corruption at a cost proportional to the amount of critical data in use. Samurai uses replication and forward error correction (majority voting) to approximate the semantics of an ideal critical memory.

- *We evaluate the use of Samurai in applications, including four SPECINT2000 benchmarks [13], a ray-shading application [21] (written in C) and a multi-threaded web server (written in C++) [2], and in libraries (an STL List class and a memory allocator).* Our results indicate that the performance overhead of Samurai varies based on the amount of critical data protected, and is often below 10%. We evaluate the error resilience of Samurai using fault-injection experiments in applications protected with Samurai to simulate memory errors. These experiments show that Samurai is able to tolerate corruptions in both critical and non-critical data, and recover the critical data successfully.

Aspects of our approach make it appealing to use in practice. First, programmers can deploy CM selectively and realize its benefits incrementally without major changes to the application and with low execution overhead. Second, CM protection works even in the presence of unsafe third-party libraries for which the source code is not available. CM makes no assumptions about and requires no modifications to third-party code, but still ensures critical data consistency. Finally, CM does not change the structure of the allocated objects or the format of pointers used to access them, making it compatible with library functions and system calls that know nothing about CM.

CM can also protect against security attacks caused due to memory corruption errors (e.g. buffer overflows). However Samurai (the software implementation of CM) does not explicitly address security attacks. While Samurai protection is adequate for a wide-range of existing memory corruption attacks, it is possible for a smart attacker to circumvent the protection provided by Samurai. This is addressed in Section 3.2 along with possible mitigation strategies.

In the remainder of this paper, we describe critical memory (Section 2), describe the design of Samurai (Section 3), and evaluate our implementation using both benchmark programs and modified libraries (Sections 4 and 5). We conclude by describing related work (Section 6) and summarizing (Section 7).

2. CRITICAL MEMORY

Critical Memory (CM) is a memory model that allows programmers to identify and protect data that is critical to the correct execution of their application. CM extends a traditional load/store instruction set architecture with additional instructions [1] that create and reference a new memory region that overlays normal memory. While traditional loads and stores read and write normal memory, critical loads and stores read and write both the normal and critical memory. To concisely summarize the semantics, critical loads return the value stored by the last critical store to a given address, despite any intervening normal stores.

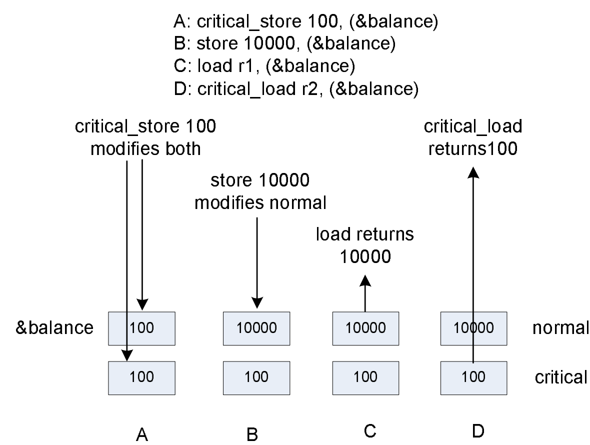

Figure 2: Critical memory operation

Figure 2 illustrates the effect of a sequence of four memory operations, `critical_store`, `store`, `load`, and `critical_load`. Critical memory exists in parallel with normal memory and critical store operations store values to both normal and critical memory, while normal store operations modify only normal memory. Load operations read normal memory, while critical load operations read from critical memory. Because this approach allows normal and critical memory contents to get out of sync, we allow critical load and store operations to detect a difference between the values in critical and normal memory and to trap if that behavior is desired (while debugging, for instance). In the figure, the critical load labelled D illustrates a mismatch that can either be tolerated or cause an exception.

Figure 3 provides an operational semantics of critical memory in a single-threaded computer[2]. Beside the critical load and store instructions, there are instructions to allocate and deallocate critical memory (`map_critical` and `unmap_critical`,

[1] These operations could be implemented as extensions to the processor's instruction set or through API calls in software.
[2] The multi-threaded case is discussed in Section 3.3.

```
Word Mem[0..MAX_MEMORY];                    // Word-level representation of normal memory
Word CritMem[0..MAX_MEMORY];                // Word-level representation of critical memory
bool IsCrit[0..MAX_MEMORY];                 // Bitmap representing whether a memory location is critical
bool skipCheck, exceptionOnCriticalMismatch;

// map an address as critical
void map_critical(Address addr) {
  IsCrit[addr] = true;
  Mem[addr] = 0;
  CritMem[addr] = SENTINEL;
}

// map an address as non-critical
void unmap_critical(Address addr) {
  IsCrit[addr] = false;
}

// promote a value to critical
void promote(Address addr) {
  IsCrit[addr] = true;
  CritMem[addr] = Mem[addr];
}

// load a value from normal memory
Word load(Address addr) {
    return Mem[addr];
}


// store a value to normal memory
void store (Word rValue, Address addr) {
    Mem[addr] = rValue;
}
```

```
// load a value from critical memory
Word critical_load(Address addr) {
    if (!IsCrit[addr] || skipCheck)
        return load(addr);
    if (exceptionOnCriticalMismatch
        && (CritMem[addr] != Mem[addr])) {
        // Raise error on mismatch for debugging
        trap(CriticalMismatchError);
    } else {
      Mem[addr] = CritMem[addr];
    }
    return CritMem[addr];
}

// store a value to critical (& normal) memory
void critical_store (Word rValue, Address addr) {
    if (!IsCrit[addr]) {
        store(rValue, addr);
        return;
    }
    // raise error on mismatch
    if (exceptionOnCriticalMismatch
        && (CritMem[addr] != Mem[addr])
        && (CritMem[addr] != SENTINEL) ) {
        trap(CriticalMismatchError);
    }
    CritMem[addr] = rValue;
    Mem[addr] = rValue;
}
```

Figure 3: Operational semantics of critical memory

respectively). A final operation, `promote`, upgrades a normal memory address to critical status, allocating a critical-memory area to shadow the normal memory area for that data item, and copying its value from normal to critical memory.

The figure shows that the semantics of normal load and store instructions remain unchanged, which allows program components that use critical memory to interoperate with libraries that do not. Our semantics are motivated by the goal of allowing programmers to selectively apply critical memory to parts of their application. As such, we consider a "critical object" to be one that has a normal memory area as well as a shadowing area of critical memory. Normal store operations modify the normal memory of such a critical object, potentially creating a mismatch in its normal and critical memory areas. Critical load or store instructions can detect these anomalies; if the inconsistency is expected (for example, due to interoperation with modules that are not aware of critical memory) then the program resolves the inconsistency and continues. Otherwise it is assumed to be the result of an unintended write, and the program can trap or perform recovery before continuing. *In a sense, critical memory refers not just to the memory, but also to the sections of code that access it via critical memory instructions, creating a sort of lightweight checkpoint and recovery system.*

In the semantics we define, critical loads have the side-effect of updating normal memory to have the same value as the critical memory in cases where the values conflict. Our rationale is that the critical memory contains the preferred value, and if the programmer chooses not to trap on such mismatches, the critical value is the one that should be propagated forward from the critical load. Critical loads and stores performed on addresses not mapped as critical behave as normal loads and stores.

2.1 Checkpointing and Recovery

Critical memory can be used in conjunction with checkpointing systems to restart applications in the case of a program crash or error. Checkpointing ensures that application data is written to disk periodically or at specific program points [14], while critical memory ensures the consistency of the checkpointed data. With checkpointing, the onus is on the programmer to provide recovery routines that reconstruct the state of the application from the checkpointed data. However, critical memory simplifies the programmer's job as it relieves the programmer from checking whether the checkpointed data is correct before taking the checkpoint. Using critical memory in conjunction with checkpointing leads to the following strategies:

Error Detection Strategies: These define the behavior of critical loads in the program.

1. **Eager detection**: Every critical load to a critical memory location checks whether the critical value matches the value in normal memory.
2. **Lazy detection**: Critical loads do not check the consistency of the critical memory location. Consistency of critical memory is checked just before checkpointing.

Error Recovery Strategies: These define the behavior of the system when the value in critical memory does not match the value in normal memory.

1. **Forward recovery**: The normal memory location is updated with the critical value, thereby correcting the corruption.

2. **Backward recovery**: An exception is raised which triggers application recovery from the previous checkpoint.

3. **Trap, no recovery**: An exception is raised which the application handles without recovery.

The error-detection strategies can be composed with the error-recovery strategies described above, leading to a total of six possible combinations of detection and recovery strategies. Each of the above strategies has specific tradeoffs in terms of performance and reliability. For example, the eager detection strategy has higher performance overheads as every critical load needs to be checked. However it also offers higher reliability than the lazy detection strategy as it leads to earlier detection of inconsistencies. Similarly, the backward recovery strategy that performs recovery from a checkpoint has higher reliability than the forward recovery strategy that corrects inconsistencies in critical data. This is because the latter can lead to propagation of erroneous values in non-critical data. However, backward recovery has higher performance overheads when errors are encountered as it involves rollback to the last checkpoint.

In this paper, we implement critical memory using an *eager detection, forward recovery* strategy. The eager detection strategy minimizes error propagation and the extra performance overhead it introduces can be alleviated using an efficient implementation (see Section 3). Further, the forward recovery strategy allows correction of inconsistencies without requiring explicit checkpointing support in the application.

2.2 What Should Be Critical?

A programmer using critical memory has to decide what data to make critical. Choosing how much data to make critical has to balance the performance impact with the reliability gains. For example, in our software-based critical memory implementation, making all data critical would have a significant performance impact, as shown by our results (Section 5.1).

Protecting critical data is tightly intertwined with providing crash recovery in applications (already discussed). Programmers writing applications such as word processing programs already identify data that must be preserved in case of a crash (e.g., the document). In general, the data required to reconstruct an application's state when it crashes is a good candidate for making critical.

Library writers also have obvious data that should be critical. For example, a memory manager implementing the malloc/free API would benefit from having its metadata be critical. We discuss our experience doing this in Section 5.5. More generally, any library collection implementation (hash table, tree, list) would be more robust if the "backbone" of the collection was made critical so that accidental overwrites of pointers do not render the entire structure unusable. If the performance is acceptable, the elements of the collection (e.g., list elements, tree contents) could also be critical. In Section 5.4 we describe our experience building a critical STL list class.

While critical memory protects against direct memory corruption, indirect errors can also corrupt critical data. For example, non-critical values that have been corrupted can be stored into critical data. Control flow errors can also cause a program to fail to perform a correct critical store to critical data or for an incorrect critical store to update a critical data location. In general, if non-critical data affects program control flow, then either that data should be made critical or other mechanisms to check the validity of control flow should be used [1]. In Section 5.3 we quantify the likelihood of indirect corruption in our benchmark applications.

2.3 Interoperability

Critical memory should allow programmers to reason locally about the critical data in their module and be confident that other modules will not corrupt it. To achieve this goal we need to (1) be able to define per-module critical data and (2) allow other modules to modify a module's critical data when necessary.

To address the first problem, we associate a module-specific key with each critical memory address when it is mapped. Critical load and store operations are required to hold the proper key when they access critical memory. Keys can be bound to critical load and store instructions by associating code address ranges with a specific module key. This binding can be performed by the linker by adding the key as an argument to critical load and store calls. Every `map_critical` operation performed by a module associates the key of the module mapping the memory with the critical address. Accidentally referencing another module's critical data with the wrong key could either default to a non-critical load or store, or raise an exception. Implementing this approach requires that the current module key be maintained across module transitions, and that the keys be checked on every critical load and store instruction. More sophisticated module/critical data bindings are also possible. We leave a more in-depth consideration of their design and implementation for future work.

Programmers will also want to allow potentially unsafe external libraries (possibly written without an awareness of critical memory) to modify their critical data. This can be accomplished without modifications to the library by allowing it to execute normally using non-critical stores to update critical memory. After the library returns, the module calling it must first check the validity of any updates made by the library to the normal memory area of critical objects. Once this memory has been vetted, the module can make the changes permanent using the `promote` operation. Any other changes to critical memory made by the external library can be transparently undone or detected when the calling module subsequently reads from the locations using critical reads or writes. Note that a crash of the library function does not result in critical data corruption, as the library does not perform critical loads and stores.

2.4 Programming Model

The most basic programmer interface to critical memory mirrors the instructions we have already presented, providing functions to allocate, deallocate, promote, and reference critical data. We have implemented this API, allowing the creation and use of critical objects on the heap (e.g., with critical_malloc, critical_free, etc.) and have used this API to modify several benchmark programs in the Olden suite [28] as well as gzip from the SPEC2000 suite [13].

```
// (A) Original C code
void add_to_list(Node* start, Patient*patient) {
    Node* list = (Node*) malloc(sizeof(Node));
    list->patient = patient;
    list->forward = NULL;
    start->forward = list;
    start = list;
}

// (B) Modified to make list objects critical
void add_to_list(Node* start, Patient* patient) {
    Node* list = (Node*)critical_malloc(sizeof(Node));
    critical_store(&list->patient,
                   sizeof(list->patient), &patient);
    list->patient = patient;
    Node* temp = NULL;
    critical_store(&list->forward,
                   sizeof(list->forward), &temp);
    list->forward = temp;
    critical_store(&start->forward,
                   sizeof(start->forward), &list);
    start->forward = list;
    start = list;
}

// (C) Modified using critical type specifier
void add_to_list(critical Node* start,
                 Patient* patient) {
    critical Node* list =
         (Node*) critical_malloc( sizeof(Node) );
    list->patient = patient;
    list->forward = NULL;
    start->forward = list;
    start = list;
}
```

Figure 4: Critical memory in the Health benchmark

Figure 4 presents two approaches to adding critical memory to the Olden program health, which performs a simulation of the Columbian health-care system [28]. In this application, patient data is stored in a linked list and we choose to make the nodes of the linked list critical. Version A in the figure is the original code, while Version B contains the same code modified to use a low-level interface to critical memory. The critical memory API described in Figure 3 has been modified to allow each operation to take a range of bytes instead of a single word, but is otherwise similar to the instruction-level API already presented [3].

Version B illustrates the challenges of using a low-level interface to critical memory. The programmer needs to ensure that all potential references to critical objects are done with critical loads and stores. While the transformation is clear in this example, in general, having the programmer identify which source statements may reference critical data is tedious and error prone.

Version C in Figure 4 illustrates a promising approach that requires additional compiler support. In this version, we have introduced a new type specifier, critical that indicates that the object pointed to is critical (much as the const specifier is used to identify constants). The compiler both checks to ensure that critical objects are not used where they are not expected and inserts the appropriate calls to critical_load and critical_store when critical objects may be referenced.

[3]The critical store implementation does not actually update the variable, so the original assignment remains in the code.

2.5 Fault Model

The focus of critical memory is to protect programs from memory corruption errors caused by pointers writing outside their intended objects and corrupting other objects in memory. In addition, critical memory also offers limited protection against the following other error categories:

1. **Soft errors in memory**: These are typically caused due to electical disturbances or cosmic ray radiation affecting hardware memory circuits, and may lead to data corruption in memory. Critical memory provides protection from soft errors that affect critical data.

2. **Uninitialized reads**: These occur when memory locations are read before being written to, which can result in non-deterministic values being returned by the read. Critical memory protects against uninitialized reads to the critical data, as the contents of the critical memory location and the normal memory location will be out of sync until a critical store or critical promote is performed on the location. This is because the critical memory contents are initialized to a special sentinel value during the *map_critical* operation as shown in Figue 3.

3. **Double/Invalid Frees**: The software implementation of critical memory, Samurai, is built on top of the DieHard allocator (see section 3), which offers probabilistic protection against double/invalid frees.

3. THE SAMURAI RUNTIME SYSTEM

Samurai is a runtime system for increasing program reliability that *probabilistically* implements critical memory using replication and forward error correction. Samurai maintains additional copies of every critical object that is allocated on the heap. The copies are called shadows and mirror the contents of the original object. When a critical store is performed, the shadows are updated with the data being stored. When a critical load is done on the object, the object is compared with its shadows to ensure that the data is consistent. If there is a mismatch, then the object and its shadows are brought in sync using simple majority voting on their contents. Because Samurai uses replication within the same address space, it cannot guarantee that multiple replicas will not be corrupted through a memory error (although the likelihood of such corruptions is minimized by the randomized allocation policy described below). As a result, it only implements critical memory semantics probabilistically, with the likelihood of corruption as a function of the amount of replication and the degree that the additional replicas are protected.

Underlying Allocator Samurai uses *DieHard* [5] as its underlying memory manager [4] because DieHard allows critical load and critical store operations to be implemented efficiently (as shown below). Even more importantly, DieHard

[4]We use DieHard in the standalone mode without process-level replication.

randomizes the location of objects in the heap, which minimizes the probability that a memory error corrupts both the object and one or more of its replicas. Thus, corruptions caused by memory errors can be corrected using simple majority voting on the object and its replicas.

Object Metadata Samurai requires all critical data in the program to be allocated using the Samurai memory allocation routines. For every critical object allocated, the Samurai memory manager transparently allocates two shadow objects on the heap. This is because at least three objects (including the original) are needed to perform majority voting for correcting corruptions caused by memory errors [5].

The addresses of these shadow objects are chosen from the Samurai heap by the DieHard allocator. The addresses are then stored as part of the heap metadata of the original object so that a write (read) of the original object can follow the pointers to the shadows and update (compare) the shadows with the contents being written to (read from). Table 1 shows the fields of the metadata in a Samurai object.

Field	Size	Purpose
Valid tag	2 bytes	Special flag for valid heap objects
Shadow pointer 1	4 bytes	Address of first shadow copy
Shadow pointer 2	4 bytes	Address of second shadow copy
Object size	4 bytes	Exact size of object
Checksum	2 bytes	Checksum of shadow pointers and size

Table 1: Fields of object metadata

Heap Organization: In order to update or compare the shadow objects, the object metadata must be accessed on every reference to the object, so that the pointers to the shadows can be followed. In the Samurai implementation, the object metadata is stored at the start of the main object and may be retrieved from the base address of the object. Hence, given an internal pointer to an object on the heap, a fast mechanism to retrieve the base address of the object is required. This mechanism is provided by the organization of the DieHard/Samurai heap as a big bag of pages (BIBOP) [5, 17].

In the BIBOP heap, objects of the same size (typically rounded to the nearest power of two) are grouped together in a contiguous heap region. Since every object in a region is the same size and regions are aligned to powers-of-two sizes, a pointer into any object in the region can be efficiently masked to create a pointer to the base of the region. From this base pointer, the size of the objects in the region can be determined via a table lookup. Knowing the object size and the offset from the base of the region, the offset of an arbitrary internal pointer to the base of the object can then be determined. The function `getBase`, which takes a pointer and returns a pointer to the base of the object, implements this translation.

Additional Checking: In order to prevent access to invalid objects on the heap through critical loads, the valid field of the object metadata contains a specifically chosen unlikely bit pattern for valid objects. The critical load operation checks for this bit pattern in the valid field before performing the load, and aborts the operation if the pattern has been corrupted.

Samurai also checks that critical stores to the critical data do not exceed the bounds of the critical-object by storing the actual size of the allocated object (not the rounded-up size) as part of the object's metadata, and checking the access to ensure that it is within bounds. This prevents critical stores from writing outside the allocated object, and corrupting other objects on the heap (critical and non-critical).

The metadata for each heap object is itself protected with checksums to protect it from corruption. In addition, a redundant copy of the metadata is stored in a separate hash table in a virtual memory protected heap region. This can be used to restore the metadata in case it is corrupted.

Critical_Malloc and Critical_Free: The pseudo-code for `critical_malloc` and `critical_free` are shown in Figure 5. These operations make use of the underlying memory allocator's allocation and deallocation routines to allocate and deallocate the object and its shadows. They also maintain the mapping between the object and its shadows in the object's metadata and the hash table.

```
void* critical_malloc(size_t size) {
    // Allocate object and its shadows with DieHard
    ptr = default_malloc( size + metadataSize );
    shadow1 = default_malloc( size );
    shadow2 = default_malloc( size );
    // Initialize metadata of the object
    ((metadata*)ptr)->shadow1 = shadow1;
    ((metadata*)ptr)->shadow2 = shadow2;
    ((metadata*)ptr)->size = size;
    ((metadata*)ptr)->valid = validFlag;
    ((metadata*)ptr)->checksum= computeChecksum(ptr);
    addToHashTable(ptr, shadow1, shadow2, size);
    return ptr + metadataSize;
}
void critical_free( void* ptr ) {
    (shadow1, shadow2, size) =
          retrieveRemoveHashTable(ptr);
    // Reset metadata corresponding to the object
    ((metadata*))ptr->valid = invalidFlag;
    // Free the pointer and its shadows
    default_free( ptr );
    default_free(shadow1);
    default_free(shadow2);
}
```

Figure 5: Pseudo-code of critical malloc/free

Critical Load and Store: Functions `critical_load` and `critical_store` are responsible for comparing and updating the shadows of an object. In order to update/compare the shadows, the pointer to the shadows for that object must first be retrieved from the object's metadata. This is done by the `getShadowAddresses` function, which given a pointer within a memory object, retrieves the pointers to the shadows of the object (after checking and repairing them if necessary) and finds the equivalent shadow object locations that mirror the location within the original object.

The pseudocode of the `getShadowAddresses` function is shown in Figure 6. In `getShadowAddresses` the hash table is accessed only when the metadata is invalid or if the check-

[5]Note that the critical memory semantics in Figure 3 maintains only one additional copy as the abstract model assumes that the critical memory is separate from the normal memory.

sum is incorrect. Otherwise, the offsets within the shadows are computed from the metadata itself, which is likely to be the common case.

```
pair getShadowAddresses(void* ptr, int numBytes) {
    void* base = getBase(ptr);
    metadata* meta = (metadata*)base;
    // Check if the object is a valid one
    if  ( meta->valid != validFlag) {
        // Check  if the object was allocated
        if  (!isAllocated(base)) return (NULL, NULL);
        meta->valid = validFlag;
    }
    // Check if metadata checksum matches
    if  ( computeChecksum(meta) != meta->checksum ) {
        // Reload version from hash table
        (shadow1, shadow2, size) = retrieveHashTable(ptr);
        // Update metadata with shadow1, ...
        ...
    }
    // The meta data is correct
    // Check the bounds of the access here
    if (ptr+numBytes >= base+metadataSize+meta->size)
       return (NULL, NULL);
     //  Compute the offset from the base ptr
     offset = ptr - (base + metadataSize);
     // Return the corresponding offsets
     // within the shadows
     return (meta->shadow1+offset, meta->shadow2+offset);
}
```

Figure 6: Pseudo-code of getShadowAddress

The critical load and critical store operations use the function `getShadowAddresses` to retrieve the offsets corresponding to the memory address within the shadows. `critical_load` compares the contents of the shadow objects at the offsets retrieved by the function `getShadowAddresses` with the contents of the original object that is being loaded. If there is a mismatch, a repair routine based on majority voting is initiated. `critical_store` copies the contents that are being stored to the offsets within the shadows returned by `getShadowAddresses`. Unlike the semantics shown in Figure 3, both `critical_load` and `critical_store` functions return immediately if called on a pointer that was not allocated with `critical_malloc` (not on the Samurai heap[6]) because our implementation assumes the non-critical load or store remains in the program.

3.1 Optimizations

We implemented two optimizations to the base Samurai operations to speed up memory accesses. These optimizations focus mainly on loads, as our experiments indicate that loads are the most numerous operations in the applications considered.

The first optimization is based on the observation that it is sufficient to compare the original object with one of the shadows during a `critical_load` in order to detect an error. Then, if there is a mismatch, the second shadow can be used to repair the error using voting. However, if the second shadow is never checked, it can potentially accumulate errors over time, and when a mismatch between the object and the first shadow is detected, the second shadow may also be corrupted, making repair impossible. We solve this problem by switching the pointers to the shadows after every N memory accesses. This allows both shadows to be checked periodically and prevents accumulation of errors in any one replica. At the same time, it incurs a branch mis-prediction in only one of N accesses. By choosing a sufficiently large value of N (=100), the cost of this mis-prediction can be amortized.

We also maintain a one-element cache for the metadata of the last-accessed object. For many applications, repeated consecutive accesses to the same object are common, and we avoid the cost of the metadata lookup on each access by keeping a cache and checking if the access was from within the cached object. However, there needs to be a balance between the size of the cache and the relative benefits of a cache-hit, because larger caches make the cost of a cache-miss higher (the entire cache needs to be searched on every access). We found that a single-element cache significantly improved performance and larger caches degraded performance.

3.2 Discussion

A software implementation of critical memory has limitations with respect to possible hardware implementations.

One limitation is that Samurai does not detect the case when multiple replicas are corrupted in exactly the same way (by a random or malicious error). Although Samurai mitigates this possibility by randomly distributing the replicas on the heap, the protection provided by Samurai is probabilistic, and Samurai cannot handle large-scale corruptions of the heap that occur in a short period of time.

Also, Samurai cannot recover when the hash table or the allocation bitmap is corrupted. Since the table and bitmap are accessed less frequently than the metadata itself (during `critical_malloc` and `critical_free` or during repairs), they are protected with page-level mechanisms, hence mitigating the possibility of corruption.

Samurai does not hide the location of the replicas of an object as it stores the pointers to the replicas in the object metadata (for fast access). A malicious attacker can locate the replicas by reading the metadata and potentially overwrite them, therby causing critical data corruption. One way to prevent this kind of attack would be to encode the pointer values in the metadata with a secret mask that is randomly generated during each execution of the program. This can prevent attackers from finding the replicas. However, this scheme incurs a constant time overhead to decode the locations of the replicas on each load or sstore. The encoding can take multiple forms from simple XOR to complex schemes such as MD5 hashing, depending on the level of security required and the acceptable performance penalty.

3.3 Multi-threading

Samurai has been designed to work correctly in a multi-threaded context, as evidenced by its use in a multi-threaded server application (see Section 5.4). However, it requires the program to be free of race conditions with respect to critical memory. In other words, every thread must take a lock before performing a critical store to a shared critical memory location. This is because Samurai performs multiple shared memory updates during a critical store, and the stored values can go out of sync if another thread performs a simultaneous critical store to the same location. Critical loads,

[6]The operation of checking if an address is on the Samurai heap is a simple range-check and takes constant time.

however, do not have this restriction as no memory updates are performed (except in the rare case when an inconsistency is detected).

The current implementation of Samurai does not support multi-threading for allocations/deallocations and repairs. However this was not a problem with the webserver application in Section 5.4 as all allocations and deallocations were made from a single thread. We detail the modifications that need to be made to the existing Samurai algorithms to support multi-threading in its full form.

1. The *critical_malloc()* and *critical_free()* operations must acquire a global lock L before performing updates to the hash table that maintains the mapping from an object to its replicas. No locking is necessary when performing mallocs and frees on the DieHard heap as the underlying DieHard allocator is engineered to be thread-safe [5]. Hence, the global lock needs to be held only when the hash table is being updated.

2. The repair routine has to be modified to acquire the global lock L prior to performing a repair on the critical data. This is because the repair can potentially access the hash table in order to correct inconsistencies in the heap metadata. Further, it is imperative that only one repair routine be active on an object and its replicas at any given time.

3. No modifications are necessary to the *critical_store* routine as we assume that the program is free of data races with regard to the critical data. This implies that either the critical data is always updated within a single thread in the program or that the program has appropriate synchronization mechanisms to prevent multiple threads from simultaneously updating the same piece of critical data. Since each critical object has its own unique shadow copy on the heap, the synchronization mechanisms also ensure that the updates to the shadow objects are synchronized.

4. No modifications are necessary to the *critical_load* routine as neither the critical object nor its shadows are modified by it (unless an inconsistency is detected, in which case, see (2) above). However, the optimizations described in section 3.1 may modify the critical object's metadata. These are considered as follows:

   - The use of a cache may result in multiple threads overwriting the cache contents and rendering the cachec contents invalid. This problem can be avoided by storing a checksum along with the cache and checking if the checksum of the cache contents matches the stored checksum upon a cache access. If they do not match, the cache contents are discared and the object is read from the heap. Implementing the checksum for the cache contents had negligible effects on the overall performance of the scheme.

   - The operation of periodically swapping shadows can be performed in a thread-safe manner using an atomic *compare_and_swap* instruction found on most processor architectures. Since this operation is performed only during one of every *N=100* accesses, it would not impact the overall performance of Samurai.

4. EXPERIMENTAL METHODS

This section describes the benchmarks used to evaluate Samurai and the methodology we used to measure its performance and fault tolerance.

4.1 Benchmarks

We evaluate Samurai in two ways. We modified four SPEC2000 benchmarks [13], and a ray-shading application [21] to use Samurai directly for the purpose of measuring performance overhead and fault tolerance. We chose specific heap-allocated data structures in each application and made them critical.

We also modified two libraries, an STL list class [22] and a memory allocator [33], to use Samurai in order to make them more reliable. For these libraries, we measure the performance of client applications of the libraries to see the impact of using Samurai. The client of the STL list class is a multi-threaded web server [2]. We measured several clients of the memory allocator, including programs that have been used in prior work to measure the performance of memory allocators [6].

Table 2 describes each benchmark application's functionality, describes the data we chose to make critical, and explains our rationale. Results from the modified libraries are discussed in Sections 5.4 and 5.5. Table 3 shows the percentage of critical data and the percentage of critical loads and stores in each benchmark application Depending on the choice of critical data, we see the fraction of loads that are critical ranges widely, from 0.01% to 12.8% (in gzip).

App	Critical bytes allocated (KB)	Total bytes allocated (KB)	Critical loads (%)	Critical stores (%)
vpr	93867	248564	0.009	0.000043
crafty	118	118	0.25	0.60
parser	61	31518	0.010	0.000013
gzip	5271	5271	12.8	0.28
rayshade	4	57	1.91	0.000040

Table 3: Application execution characteristics

4.2 Performance Evaluation

We measured the performance overhead of Samurai on a dual 3.4 Ghz Intel Pentium(R) 4 Desktop system with 2GB RAM running Windows XP SP2 under light load. Because we currently do not have a compiler that will automatically determine statically which loads and stores should be made critical when critical data is declared, we instead check for critical data at runtime. The Phoenix compiler infrastructure we use [9] allows us to instrument all heap loads and stores in the benchmark applications. At runtime, our system dynamically determines which loads and stores access critical data and calls the appropriate Samurai function. If the data is non-critical, normal loads and stores are performed. To calculate the overhead of Samurai, we subtract out the runtime cost of checking whether a particular load or store is critical from the overall execution time as follows.

Let T_{Base} be the execution time of the program without instrumenting references and without using Samurai. Let $T_{InstCheck}$ be the execution time with all loads and stores in the program instrumented with Phoenix, and including

App	Functionality	Critical Data	Rationale
vpr	Performs Routing of circuit blocks onto a FPGA chip. Only used in route mode.	*rr_graph* data structure to maintain global routing constraints that are updated after each assignment.	An error in the structure could impact future routing decisions and result in incorrect routing assignment.
crafty	Plays a game of chess with the user	Cache of previously seen board positions that is used to speed up evaluation of the game tree.	Error in cache can result in program making an erroneous or non-optimal move.
parser	Reads in a list of sentences and parses them into syntactic constructs.	Tree structure used for looking up words in the dictionary.	If the tree structure is corrupted, parsing of all subsequent sentences in list would be incorrect.
gzip	Decompress a file read in from the disk using Huffman decoding. Only used in decompression mode.	Huffman decompression tree used for expanding encoded blocks read in from disk.	Any corruption of the tree would make the expansion of code words in a block meaningless.
rayshade	Renders a scene file on the screen using ray-tracing.	List of scene objects along with their positions in the scene.	Minor changes in the scene may go unnoticed, but changes in the list of object will likely result in objects being changed or disappearing, and is hence more likely to be noticed in the scene.

Table 2: Description of applications and critical data

the dynamic check that determines if a reference is critical. We define $T_{\Delta InstCheck} = T_{InstCheck} - T_{Base}$ as the total instrumentation and checking cost. Let f be the fraction of critical loads and stores divided by total loads and stores. We approximate the cost of instrumenting and checking only the critical loads and stores as $T_{\Delta CritInstCheck} = f * T_{\Delta InstCheck}$.

Finally, let $T_{Samurai}$ be the execution time of the application with the critical data allocated using the Samurai allocator and critical loads and stores in the program executed in Samurai. We define $T_{\Delta Samurai} = T_{Samurai} - T_{InstCheck}$ as just the overhead to execute the additional critical mallocs, frees, loads and stores. We then estimate the total execution time of an application using Samurai as:
$T_{SamEst} = T_{Base} + T_{\Delta CritInstCheck} + T_{\Delta Samurai}$.
We used the approach described above to calculate the performance results from the four SPEC benchmarks and the rayshade application presented in Section 5.

4.3 Fault-injection Experiments

Fault-injection is a standard technique used to evaluate the resilience of fault-tolerance mechanisms and to measure their coverage [31]. We evaluate Samurai's fault tolerance in two cases: when critical data itself is corrupted and when non-critical data is corrupted and that corruption propagates to critical data.

To measure the effect of corruptions of critical data, the structure of the fault-injector is as follows. First, a random critical object on the heap is chosen according to a fault-injection policy (see below). A random offset is chosen uniformly from within this object, along with a random number of bytes (from a uniform distribution) no larger than the size of the object. Then the memory locations within the object starting from the random offset are overwritten with random values up to the random number of bytes chosen. The net effect of the injected fault is to corrupt a single object on the Samurai heap, be it allocated or unallocated. Note that multiple faults can be injected per run, leading to corruption of more than one object during a single execution of the program.

Our fault-injection policy probabilistically chooses objects from the Samurai heap, based on the number of allocated objects in each partition. Partitions that have more allocated objects are given higher weight in the sampling process, and hence objects are more likely to be chosen from more heavily allocated heap regions, eliminating a sampling bias towards large heap objects.

We also want to quantify the probability of an error in non-critical data propagating to critical data. We focus on three ways an error in non-critical data can propagate to critical data: (1) a critical store can write an incorrectly computed value (*data error*), (2) the program can follow the wrong control path and perform incorrect critical stores (*control error*), (3) a critical store can write to an incorrect but valid critical address (*pointer error*). To classify these errors we first obtain the memory trace of all critical stores to critical data with no faults injected. This trace (the *golden trace*) contains the program counter of the instruction performing the store, the address being stored to and the contents that are being stored.

We filter the golden trace to find the program counter addresses that can potentially write to critical data in the program. This is the set of critical stores in the program. We then inject faults into the non-critical data and compare the memory trace of stores performed from critical store locations with the golden trace. A mismatch indicates a corruption of the critical data due to error propagation. A mismatch in the program counter field indicates a control error, a mismatch in the contents field indicates a data error and a mismatch in the address field indicates a pointer error.

5. RESULTS

This section presents our experimental results, including measurements of performance overhead, fault tolerance, and experience with using Samurai to harden libraries.

5.1 Performance

Figure 7 shows the relative execution time of the four SPEC benchmarks and rayshade measured with and without being modified to use critical memory. The overheads are normalized to 1 (baseline = without Samurai) and lower is better. For the four SPEC benchmarks, the ref inputs were used [13], while for rayshade, an animation of a coin being flipped was used to measure the slowdown.

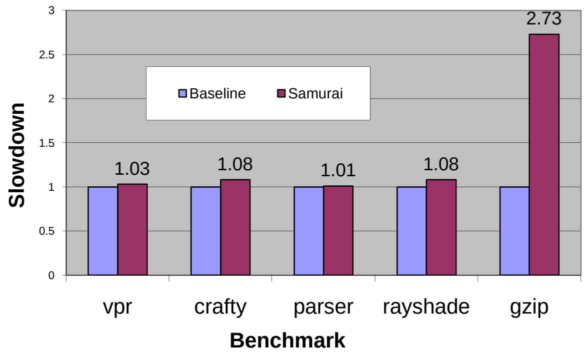

Figure 7: Performance overhead of Samurai

The results show that for all applications except gzip, the performance overhead is less than 10%. For gzip, the overhead is around 2.7x (averaged across all the ref-inputs). This is because the fraction of critical loads and stores to critical data in gzip is relatively high (10-15%) compared to the other benchmarks. Further, the critical data consisting of the Huffman decompression table constitutes all the heap data for this application, and the accesses to this table are on the performance-critical path.

Based on the numbers in Table 3, one can observe that the three programs *crafty*, *gzip* and *rayshade* have more than 1% combined critical loads and stores. These programs exhibit overheads of between 4 to 15 % for every percentage of critical loads and stores. This suggests that the critical data for an application must be chosen with care, and blindly making a large percentage of the data critical can result in prohibitive performance overheads.

5.2 Injections into Critical Data

We inject faults into the critical data of the four SPEC benchmarks (using the test inputs) and rayshade to understand the effectiveness of Samurai's fault tolerance. During the execution of the application, faults are injected into the heap one every N memory accesses, where N is the period of the injection. For each value of the period N, the application is executed 10 times and the outcome is classified into failure (crash or incorrect output) and success. The fault period is varied from 100,000 to 1,000,000 in increments of 100,000.

Faults are injected into the critical data of the application both with and without Samurai. The results of the fault-injection experiments with one application, *vpr*, are shown in Figure 8. We observe that the number of trials in which the application completes successfully increases as the fault period increases (red/dark bars), although the curve itself is quite jagged indicating high variance because of the relatively small number of trials performed.

Figure 9 shows the result of the same fault-injection experiment performed with an application protected with Samurai. We observe that Samurai allows the application to complete successfully for all fault-rates. There is one case when the fault-period is one every 100000 accesses (the highest in our experiments) where Samurai detects an inconsistency that it is unable to correct due to multiple copies of an object getting corrupted. However, the application goes on to continue and produce correct output showing that the error was benign. This is considered a false positive for Samurai.

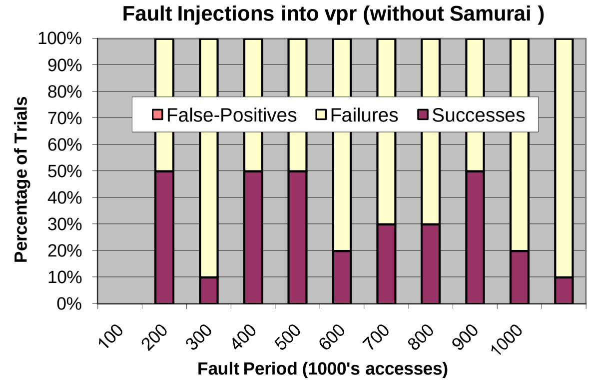

Figure 8: Fault tolerance of *vpr* without Samurai

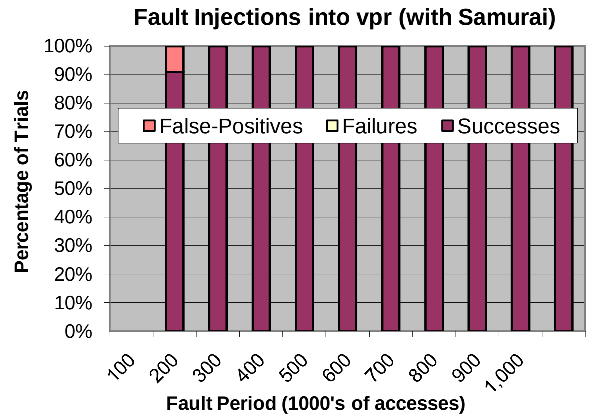

Figure 9: Fault tolerance of *vpr* with Samurai

5.3 Injections into Non-critical Data

We now consider the effects of injecting faults in the non-critical data of our five applications (again, using the SPEC benchmark test inputs). The outcomes of the injections are classified as data, control or pointer errors as explained in Section 5.4. The results are shown in Table 4.

App	Trials	Errors in Critical Data				Assert.
		Data	Cntrl	Ptr	Total	Errors
vpr	550 (199)	203 (0)	0 (0)	1 (1)	204 (1)	2 (2)
crafty	55 (18)	9 (3)	12 (7)	4 (3)	25 (13)	0 (0)
parser	500 (380)	3 (1)	0 (0)	0 (0)	3 (1)	14 (14)
gzip	500 (52)	4 (4)	0 (0)	5 (5)	9 (9)	37 (37)
rayshade	500 (68)	5 (1)	0 (0)	0 (0)	5 (1)	1 (1)

Table 4: Fault Injections into non-critical data

The numbers in the table that are not in parentheses indicate how many trials produced a particular result. The numbers in parentheses below those numbers indicate how many of those trials resulted in a failure (e.g., the program either crashing, hanging, or producing an incorrect result). We also report the number of assertion violations or consistency check errors reported by each application. These assertions/checks were already present in the applications' code and indicate the amount of error resilience built into the applications.

For all benchmarks except *vpr* and *crafty*, we observe that the number of faults propagating from non-critical data to the critical data and resulting in a data, control or pointer error is relatively small (less than 2%).

For *vpr*, the number of data-errors is high, but none of these resulted in a failure. This is because in *vpr*, the address of a region allocated using regular malloc is stored in the critical data, and corrupting the value of a random store is highly likely to corrupt the data structures used in the standard allocator (not the Samurai allocator). The corrupted structures result in an address being returned by the call to malloc that is different from the address in a correct execution. Nevertheless, this different address does not result in an error for this application as it does not impact the correctness of the data. Hence, these errors do not impact the correctness of the application, although they propagate to the critical data.

In *crafty* there are a relatively large number of cases in which errors in non-critical data propagate to the critical data, and result in failures. This is because the *crafty* application performs repeated computation and stores the results in the cache. Therefore, in this application, a memory error results in incorrect computation, which in turn is stored in the cache, resulting in error propagation from non-critical to critical data.

The last column of Table 4 shows the number of assertions/consistency checks violated in each application due to the errors injected. *Crafty* has the fewest assertion violations, which indicates that this application has few built-in consistency checks. *Gzip* has the most assertion violations as it uses cyclic-redundancy checks (CRCs) on the data.

5.4 STL List Class and Web Server

We used Samurai to implement a reliable version of an STL list class [22] based on critical memory. We then modified a multi-threaded web server to use our reliable list class to protect the list of threads associated with connections. This implementation gave us the opportunity to see how easy it is to build and use a library based on critical memory and directly measure the cost of using it in a real server application.

We modify a standalone version of HP's STL List class [25] to make it use critical memory for storing the list contents[7]. We chose to modify the class so that both the list elements and the list backbone (e.g., pointer structure) were critical. The required changes included:

- We modified the custom allocator for the class to use *critical_malloc* and *critical_free* to allocate and deallocate list objects.
- We modified the member functions of the class, such as *insert* and *erase* to call the Samurai API functions in order to check and update the list contents (including the list pointers) being accessed in the functions.
- We modified the custom iterators for the class to call the Samurai functions to ensure that list elements were consistent prior to the iterator being dereferenced. We added a new call-back function in the iterator to update an object's replicas in Samurai (i.e., an object-level version of promote), which must be called by the client if the object is modified directly through an iterator rather than through a member function of the list class.

[7]The list class was downloaded from `http://www.cs.rpi.edu/~musser/gp/lists.html`

We modified a multi-threaded web server [2] to use the critical STL list class. This web server spawns a new thread to service each incoming request from a client, and uses STL lists to maintain a list of all active threads in the system. We replaced these lists in the application with our Samurai list class to protect them from corruption. This list is important because if it is corrupted, it can result in runaway threads whose behavior is different from the intended behavior of the application. Further, when a thread completes execution, it returns control to the main thread, which removes the thread from the global list of threads. A corruption in the thread list could result in the main thread crashing or going into an infinite loop, which would result in the server not being able to accept new connections.

In order to evaluate the performance overhead of the modified web server, we developed a multi-threaded client program that sends HTTP requests to the server in batch mode. The client program spawns multiple threads, each of which opens an HTTP connection to the server, sends a request and waits for a response. On receiving a response, the client thread closes the connection and the entire process is repeated. The server processes each incoming request (in a separate thread) and sends the results back to the client. For the purposes of our evaluation, both the server and the client execute on the same machine, so there is no network delay. We measure the time taken by the client to complete processing a fixed number of requests using a certain number of threads. The slowdown experienced by the client as a function of the number of client threads and total number of requests issued is shown in Table 5. Each number in the table was obtained by computing the average of three trials.

No. of	No. of	Time (seconds)		Percentage
Requests	Threads	Samurai	Baseline	Slowdown
10	1	10.0	10.0	0.0
10	10	1.42	1.36	8.8
100	1	101.25	100.99	0.0
100	10	13.50	12.50	7.9
100	100	3.36	3.23	4.0
1000	1	1008.06	1007.81	0.0
1000	10	126.67	122.89	3.0
1000	100	27.85	27.51	1.2

Table 5: Client request times for web server

As the table shows, the slowdown is within 10% for the range of requests and client threads considered. Further, the slowdown first increases as the number of threads increases from 1 to 10 and then decreases as the number of threads increases from 10 to 100 at the client. For the single-threaded case, the slowdown is close to zero as the client issues requests one at a time, which are then handled by separate server threads. However at any time only a single thread is active at the server to service the request, and therefore the overhead of adding or removing threads from the thread list is negligible (0% in the table). As the number of threads at the client increases from 1 to 10, the number of active threads at the server also increases and therefore the overhead of adding and removing threads from the thread list results in the application slowing down by about 8% (for 100 requests). However, when the number of threads at the client increases to 100, the overall slowdown at the client drops to 4% (for 100 requests). This is because the increase in the performance overhead in accessing the list elements is counterbalanced by the increase in parallelism at the client which results in better overlapping of request latencies.

5.5 A Reliable Memory Allocator

Memory allocators typically store metadata about the allocated data on the heap. This metadata is itself susceptible to corruption by the application, either accidentally through an error or maliciously through a security attack. Such metadata corruption is often fatal for the application and can lead to security vulnerabilities [19].

An important consideration in the design of a reliable memory allocator is separation of the application data from the allocator's metadata. This is a challenging problem because memory allocators often store the metadata contiguously with the application data (for reasons of locality and efficiency). Existing approaches sometimes use virtual memory protection to protect the pages around the metadata. The Heap Server project [20] goes as far as to place the allocator metadata in a separate process. These approaches can involve considerable reengineering of the allocator's code and may not be practical to implement for all memory allocators. Further, the Heap Server approach can be expensive because it requires crossing the process-kernel boundary through a system call or through inter-process communication. For allocation intensive programs, Heap Server's overhead can be as high as 60% [20].

We implemented a reliable memory allocator that uses Samurai to protect the allocator metadata. Memory allocators typically use an OS interface (e.g., `sbrk` or `VirtualAlloc`) to allocate and deallocate large chunks of memory, which are then used for storing both application data as well as allocator metadata. We interposed at this layer and replaced the large-chunk allocator with Samurai. By only modifying the allocator itself, no changes to the allocator clients are required. Furthermore, with this approach, we are able to protect the allocator metadata wherever it is, even if it is intermixed with the application data. Thus, using Samurai requires only small changes to the allocator.

The one change that is required is that when the allocator references its metadata, it must use critical load and store operations. Note that with this approach, the allocator client performs normal loads and stores to the heap data even though that data has been allocated on the Samurai heap. The client sees a consistent view of its data in the absence of memory errors, even though it is not using critical loads and stores. However, in the event that an inconsistency is detected in the allocator metadata, we have to modify the repair strategy of Samurai in this library to repair only the allocator's metadata and not the application data. Alternately, we could avoid a repair strategy altogether and simply raise an exception when corruption is detected.

In order to demonstrate this idea, we modified the *HeapLayers* package [6] to use Samurai as the underlying allocator. HeapLayers provides a family of allocation strategies implemented as layers that can be composed together to build memory allocators. This in turn allows the application designer to build custom allocators composed of layers representing the policies best suited to their application's characteristics. Berger et al. [6] describe an implementation of the popular Kingsley allocator [33], using four layers in HeapLayers:

1. *StrictSegHeap*: Provides segregation of objects by the object size. Each object size is rounded to its nearest power-of-two and stored in the bin corresponding to the size.

2. *SizeHeap*: Stores the object size along with the object to facilitate fast size lookups of objects

3. *FreeListHeap*: Stores the list of free objects as a single-linked list on the heap. Recycles objects from this list for fast allocation.

4. *SbrkHeap*: Base heap that allows the Kingsley heap to expand and contract as necessary by allocating and deallocating pages from the operating system.

To implement a reliable Kingsley heap, we replaced the underlying SBrkHeap with the Samurai heap in HeapLayers and added critical load and store calls at appropriate places in the *SizeHeap* and the *FreeListHeap* layers. These modifications involved about 10 lines of the nearly 1000 lines of C code implementing the Kingsley heap. We also interposed a *ChunkHeap* layer between the *FreeListHeap* and the *Samurai* heap, to allocate memory in chunks of 1024 bytes (this did not affect the overall correctness of the design, but resulted in better performance). It took us less than a day to make these changes to the Kingsley allocator, and most of it consisted of understanding the existing code.

We used a series of allocation-intensive programs (most of which have been used in prior work [4][6]) to evaluate the performance of the modified Kingsley heap. The applications were statically linked with the reliable allocator and executed on a dual-core Pentium 1.8 Ghz laptop with 2 GB RAM under light load. Each application was executed with pre-specified inputs and the average of five trials was computed for those inputs that had reasonably high execution times (this ensured low variance among trials) [8].

For comparison purposes, we also used the unreplicated version of the DieHard allocator [5] as the underlying allocator for the Kingsley heap. The DieHard allocator randomizes the location of the allocated object in the heap, but does not maintain multiple replicas of the object. The original Kingsley allocator is used as the baseline and all measurements are normalized to the performance of the Kingsley allocator. For each benchmark, we evaluated the performance of three allocators - Samurai-based, DieHard-based and the original Kingsley allocator.

The overall performance of the applications with the modified allocator(s) is summarized in Figure 10. It can be observed that the average overhead of the Samurai-based allocator was 10%, while the overhead due to the DieHard-based allocator was 6%. The worst-case overhead for the Samurai-based allocator was 23% (for espresso), and the worst-case overhead for the DieHard-based allocator was

[8] Some inputs that had low execution times (<1 sec) exhibited high variance due to measurement noise, and are hence not reported here. These numbers are available upon request.

12% (for mudlle). These results were obtained with the version of DieHard used in [5].

The overhead incurred by the DieHard-based allocator represents the overhead due to the loss in locality as a result of randomization. The additional overhead incurred by the Samurai-based allocator is the cost of allocating and deallocating multiple replicas of the object and synchronzing/comparing them.

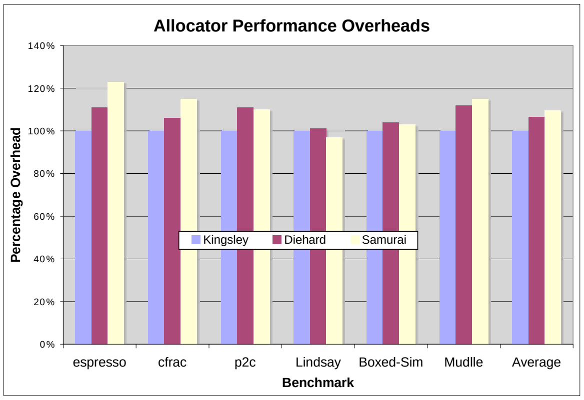

Figure 10: Reliable allocator performance

6. RELATED WORK

We classify related work into three categories: robust data-structures, static and dynamic checking, and error-tolerance for applications.

6.1 Robust Data-structures

There have been a number of papers on data structure checking and repair, as well as synthesizing robust data structures [7, 18]. The main idea of these papers is that the programmer specifies invariants about data-structure properties and the system ensures that these invariants hold. Demsky and Rinard present a planning-based approach to repair data structures transparently in an application [10]. However, the repaired data structure may not be semantically equivalent to the data structure in a correct program resulting in unexpected behavior. Further, the repair is carried out after a program crash (or periodically), by which time the program could have produced incorrect output.

6.2 Static and Dynamic Checking

There has been considerable work on bringing the type-safety properties of languages such as Java to C and C++. CCured uses static analysis and type-inference to classify pointers in the program based on their usage into safe, sequential and wild [23]. Safe pointers and sequential pointers can be checked at compile-time and only wild pointers need to be checked at runtime. However, supporting arbitrary third-party plugins whose source-code is not available at compile-time is challenging for CCured. Also, CCured requires engineering effort to make it compatible with library code, as it modifies the format of pointers in the program [8].

Another approach to provide memory-safety guarantees to C programs is to check every pointer access at runtime to ensure it is within the bounds of the referent object as done by Jones and Kelley [16] and extended by Ruwase and Lam [29]. Dhurjati and Adve propose using a special compiler optimization known as pool-allocation to reduce the overhead of bounds-checking [11]. The main problem with these dynamic approaches is that they need to check every pointer access to ensure it is within bounds. This can be a problem for large applications where checking every pointer dereference at runtime can be prohibitively expensive. Further, most of these approaches stop the program upon encountering a memory error, rather than allowing it to continue. An exception is the boundless-buffers work by Rinard et al. [27], in which a pointer is allowed to go out of bounds of the object, but made to point to a special location.

A third approach to ensure memory safety of C programs is Software-Fault Isolation (SFI) [32]. In SFI, each module in a program is given the illusion of executing in its own address space and the compiler/binary-rewriting tool ensures that one module cannot access memory outside its pseudo-address space. However, this approach requires modifications to the source or binary of the application and its libraries.

6.3 Error-tolerance for Applications

DieHard is a memory allocator that hardens the application to memory errors [5]. DieHard provides probabilistic soundness guarantees for applications that allow them to continue execution in the presence of memory errors. DieHard offers two modes of protection: replicated and unreplicated. Replicated mode, in which an entire process is replicated, provides the strongest protection guarantees in DieHard. Samurai differs from DieHard in that it only replicates critical memory and the operations on it, while still providing strong protection guarantees for the application. However, whereas DieHard replicates processes transparently with no programmer effort, Samurai requires the critical data to be explicitly identified by the programmer.

Failure-oblivious computing aims to continue program execution after a memory error, by ignoring illegal writes and manufacturing values for illegal reads [27]. The problem with this approach is that after a memory error has occurred, the state of the application is undefined and the programmer has no way of knowing if the application will continue correctly after the memory error.

The Rx system combines checkpointing with logging to recover from detectable errors such as crashes [26]. Upon a failure, Rx rolls back to the latest checkpoint and re-executes the program in a modified environment. Rx is unsound in that it cannot detect latent errors that do not lead to program crashes.

The Sprite Operating System attempts to provide fast recovery to applications using the concept of a recovery box [3]. A recovery box is a specially designated area of memory to which an application can write important data that must be recovered after a system crash. In order to access the recovery box, the application must use a structured interface, and hence requires extensive code modifications.

SafeDrive [34] is a system that attempts to provide application recovery in the presence of erroneous extensions. Safedrive uses strong typing and type-safety checks in the extension code to provide fine-grained resource tracking for recovery. However, Safedrive requires all extensions to be type-checked prior to being loaded into the application space, which requires the extensions's source code.

7. SUMMARY

This paper introduces critical memory, a memory model that protects specific data from arbitrary non-local changes. Critical memory enables local reasoning about the consistency of memory in type-unsafe programs and can be used for a variety of purposes. We describe the semantics of critical memory, and discuss how the concept can be exposed to a programmer through APIs, libraries, and language features. Samurai is a runtime system that implements critical memory with the goal of providing probabilistic memory safety guarantees in C and C++ programs. Samurai implements critical memory using replication layered on top of a robust runtime system. Samurai enables programmers to selectively identify and protect key data structures, thus allowing the effort and overhead of using Samurai to be tailored to the application. We demonstrate Samurai by modifying five benchmark applications as well as an STL list class implementation and a memory allocator. We show that the overhead of Samurai is 10% or less for the vast majority of the applications and libraries considered. We also show that Samurai increases application fault tolerance for faults injected both into critical and non-critical data.

Acknowledgements We thank Ted Hart, Emery Berger, Pramod Joisha, Michael Hicks, Shuo Chen and Zbigniew Kalbarczyk for insightful discussions regarding this work. We also thank our anonymous reviewers for their comments on this paper.

8. REFERENCES

[1] Abadi, Budiu, Erlingsson, and Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *CCS'05*, 2005.

[2] S. Abeghyan, "Multi-threaded server class with example of http server." [Online]. Available: http://www.codeproject.com/internet/mhttpsrv.asp

[3] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the UNIX environment," in *USENIX'92*, 1992, pp. 31–44.

[4] E. Berger, B. Zorn, and K. McKinley, "Reconsidering custom memory allocation," in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002.*, 2002.

[5] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *PLDI '06*. New York, NY, USA: ACM Press, 2006, pp. 158–168.

[6] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing high-performance memory allocators," in *PLDI'01*, 2001, pp. 114–124.

[7] J. D. Bright and G. F. Sullivan, "On-line error monitoring for several data structures," in *FTCS*, 1995, pp. 392–401.

[8] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," in *PLDI*. ACM, 2003, pp. 232–244.

[9] M. Corporation, "Phoenix compiler infrastructure." [Online]. Available: http://research.microsoft.com/phoenix

[10] B. Demsky and M. C. Rinard, "Goal-directed reasoning for specification-based data structure repair," *IEEE Trans. Software Eng*, vol. 32, no. 12, pp. 931–951, 2006.

[11] D. Dhurjati and V. S. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *ICSE'06*. ACM, 2006, pp. 162–171.

[12] R. Hastings and B. Joyce, "Fast detection of memory leaks and access errors," in *Proceedings of the Winter '92 USENIX conference*, 1992, pp. 125–136.

[13] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[14] Y. Huang and C. Kintala, "Software fault-tolerance in the application layer," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & sons, 1995, pp. 231–248.

[15] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX'02*, June 2002, pp. 275–288.

[16] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *AADEBUG*, 1997, pp. 13–26.

[17] P.-H. Kamp, "Malloc(3) revisited," in *FreeNIX Track: USENIX'98*, 1998, pp. 193–198.

[18] K. Kant and A. Ravichandran, "Synthesizing robust data STructures - an introduction," *IEEE Trans. Computers*, vol. 39, no. 2, pp. 161–173, 1990.

[19] M. Kempf, "Smashing the heap for fun and profit." [Online]. Available: http://doc.bughunter.net/buffer-overflow/heap-corruption.html

[20] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," in *ASPLOS'06*. ACM Press, 2006, pp. 207–218.

[21] C. Kolb, "Rayshade graphics program." [Online]. Available: http://graphics.stanford.edu/~cek/rayshade

[22] D. R. Musser and A. Saini, *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1995.

[23] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst*, vol. 27, no. 3, pp. 477–526, 2005.

[24] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electr. Notes Theor. Comput. Sci*, vol. 89, no. 2, 2003.

[25] H. Packard, "Stl – hewlett-packard's downloadable standard template library." [Online]. Available: ftp://butler.hpl.hp.com/stl

[26] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies - a safe method to survive software failures," in *SOSP'05*, A. Herbert and K. P. Birman, Eds. ACM, 2005, pp. 235–248.

[27] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," in *ACSAC*. IEEE Computer Society, 2004, pp. 82–90.

[28] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM TOPLAS*, vol. 13, 1995.

[29] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *NDSS'04*, Feb. 2004, pp. 159–169.

[30] C. C. E. R. Team, "Secure coding." [Online]. Available: http://www.cert.org/secure-coding

[31] J. M. Voas and G. McGraw, *Software fault injection: inoculating programs against errors*. New York, NY, USA: John Wiley & Sons, Inc., 1997.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP*, 1993, pp. 203–216.

[33] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *IWMM*, 1995.

[34] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer, "Safedrive: Safe and recoverable extensions using language-based techniques," in *OSDI'06*. Usenix, 2006.