Generative Software-based Memory Error Detection and Correction for Operating System Data Structures

Christoph Borchert, Horst Schirmeier and Olaf Spinczyk
Department of Computer Science 12
Technische Universität Dortmund, Germany
e-mail: {christoph.borchert, horst.schirmeier, olaf.spinczyk}@tu-dortmund.de

***Abstract*—Recent studies indicate that the number of system failures caused by main memory errors is much higher than expected. In contrast to the commonly used hardware-based countermeasures, for example using ECC memory, software-based fault-tolerance measures are much more flexible and can exploit application knowledge, such as the criticality of specific data structures. This paper presents a software-based memory error protection approach, which we used to *harden* the eCos operating system in a case study. The main benefits of our approach are the *flexibility* to choose from an extensible toolbox of easily pluggable error detection and correction schemes as well as its *very low runtime overhead*, which totals in a range of 0.09–1.7 %. The implementation is based on aspect-oriented programming and exploits the object-oriented program structure of eCos to identify well-suited code locations for the insertion of generative fault-tolerance measures.**

I. Introduction

Errors in main memory are one of the primary hardware problems for failures of today's computer systems [1], [2], [3], [4]. A recent study [1] reports that about one third of all machines in Google's server fleet suffer from at least one DRAM error per year. This, already severe, problem is expected to worsen in the future [5], [6], as VLSI technologies move to higher chip densities and lower operating voltages, dramatically increasing sensitivity to electromagnetic radiation.

A remedy to this problem is the use of memory-error protection hardware, nowadays common on almost every server system. A widespread form of protection is found in memory chips with *error-correcting codes* (ECC), such as the *single-bit-error correcting and double-bit-error detecting* (SEC-DED) extended Hamming code [7]. By its very nature, this code cannot recover from word-wise multi-bit errors, which nonetheless contribute to at least 17 % of all DRAM errors in practice [3], [4]. IBM's high-end *Chipkill* [8] technique tolerates such multi-bit errors (typically 4 adjacent bits) by interleaving a word to independent DRAM chips. However, this comes at the cost of reduced performance and up to 30 % higher energy consumption due to forced narrow-I/O configuration [9], so that Chipkill remains useful solely for very expensive, highly reliable systems. Low-cost systems, primarily addressed in this paper, cannot afford such advanced protection.

Hardware-based solutions usually protect the entire memory space – the overhead of redundancy is paid for every single bit, even if never used by the software. Often, bit errors in the used parts of the main memory do not affect the system's behavior, for example if not read before the next write access. This highly depends on the application software (including the operating system) that, as we show in Section II, can be analyzed to find a partition into *critical* and *non-critical* memory spaces.

For these analyses, we assume a transient single-bit and burst fault model of the dynamic RAM. The read-only text segment, holding the program instructions, is stored in a far more reliable ROM. For instance, Flash memory used commonly in embedded systems is 3–5 orders of magnitude less susceptible to radiation than DRAM and SRAM [10].

We propose a purely software-based memory-error protection that exploits the application's knowledge of *critical* memory accesses, which are extracted from the software's source code at compile time and enforced by compiler-generated runtime checks. The greatest challenge is the placement of the runtime checks in the control flow of the software, that is, to analyze which instructions work on which parts of the memory. In general, this is an undecidable problem. Therefore, we focus our analysis on object-oriented software, for which this problem becomes solvable with certain restrictions (see Section III).

In the following sections, we apply our software-based memory-error protection to the *embedded Configurable operating system (eCos)* [11], which is written in object-oriented C++. Our software-based approach offers great flexibility in error detection and correction mechanisms, as it is configurable at compile time whether errors should be detected or additionally corrected, whether single-bit or multi-bit errors should be detected/corrected, and whether permanent or transient errors are considered. These decisions can be taken independently on each object-oriented data structure.

We make three contributions:

- We analyze the vulnerability of the *embedded Configurable operating system (eCos)* to memory errors on bit-level granularity (Section II). It is shown that only a small fraction of RAM is susceptible in a sense that the operating system crashes or misbehaves. Errors in large parts of RAM do *not* affect the operating system's stability. We find that most error-susceptible memory is aligned to the operating system's internal data structures, such as *Scheduler* and *Thread* objects. However, this highly depends on the user applications that run on eCos.

- We precisely describe a generative algorithm for software-based error detection and correction in object-oriented data structures (Section III and IV). This approach offers the flexibility to choose from an extensible toolbox of error-correcting codes, for example Hamming codes. By exploiting *aspect-oriented programming* [12], our algorithm can be easily applied to arbitrary C++ software, as a compiler automatically inserts the chosen protection mechanisms. This is the primary contribution of this paper.
- By applying our algorithm to the eCos kernel test suite, we prove the effectiveness and efficiency of our approach (Section V). The likelihood that the operating system fails due to transient single-bit and multi-bit errors is significantly reduced at a *very low total runtime overhead* of 0.09–1.7 %. Our evaluation points out the trade-offs between several error-correcting codes, showing that a two's complement addition checksum plus replica is very efficient.

II. Problem Analysis

"... the only DRAM bit errors that cause system crashes are those that occur within the roughly 1.5 % of memory that is occupied by kernel code pages." [2]

In general, the operating-system (OS) *kernel* is the most important piece of software with regard to dependability, as all other software components depend on the OS. Surprisingly, in spite of their impact on total system resiliency and – compared to the rest of the system – their very small memory footprint, state-of-the-art OS kernels are not equipped with software-based protection against memory errors: An efficient software-based fault-tolerance technique would offer an enormous potential to reduce system failures. Unfortunately, most earlier attempts to apply software-based memory protection suffer from excessive runtime overhead, ranging between 30 % and 260 % [13], [14], [15], [16]. These studies only address user-level applications; such extreme performance degradations are considered unacceptable for the OS layer, especially in the case of general-purpose OS.

The key to efficient software-based memory protection is to exploit knowledge on the application's behavior and its OS usage profile. Focusing on special-purpose embedded systems, this profile can be assumed to remain largely unchanged over a system's lifetime. Our working hypothesis is that only a small – application-dependent – subset of the OS's state is actually "mission critical", and faults in other parts of memory do not affect the system's stability. Accordingly, only the *critical memory space* needs protection, calling for a configurable and highly localized application of error detection (EDM) and error-recovery mechanisms (ERM).

A. Baseline Assessment: eCos Fault Susceptibility

To assess the validity of this working hypothesis, we examined the fault resiliency of a set of benchmark and test programs for *eCos* by fault-injection experiments. Both the benchmark programs, bundled with eCos itself, and the eCos kernel are implemented in object-oriented C++, and compiled

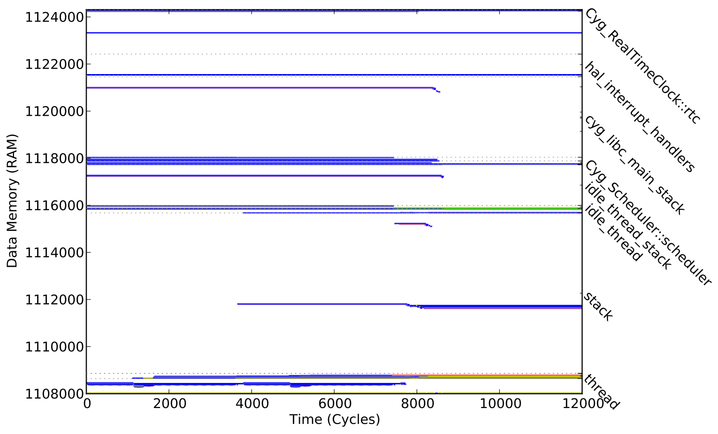

Figure 1: eCos fault susceptibility: Each point denotes the outcome of an independent benchmark run (THREAD1), after injecting a single-bit flip at a specific time and data-memory coordinate. Injections in *white* areas have no observable effect. *Blue* marks illegal memory accesses and jumps. CPU exceptions are colored *red* and timeouts *yellow* respectively. *Green* data points show benchmark runs that finish, but yield wrong output (silent data corruption).

for an i386 target. We used FAIL* [17], a versatile fault-injection (FI) and experimentation framework, to inject single-bit flip faults (in Section V we additionally evaluate a burst-error fault model) into Bochs, an IA-32 (x86) emulator, and to observe the benchmark behavior afterwards.

The fault-injection result excerpt in Figure 1 shows that the vast majority of injections have no effect on the benchmark run (white areas), and that failures often originate in the same memory locations (horizontally aligned, colored address-space/time coordinates): These locations seem to represent the *critical memory space* for this particular benchmark and the chosen eCos configuration. Table I confirms this assumption – the top ten program symbols respectively contiguous memory areas that caused the THREAD1 benchmark to fail amount to 99.87 % of all observed abnormal program terminations. The MUTEX1 results (in the same table) display a similar address-space clustering, yet with a different distribution: As MUTEX1 exhibits a different OS usage profile, a different subset of the program state is the most critical.

This baseline assessment reveals that in the chosen set of benchmarks, the kernel and application stacks and the scheduler-related kernel data structures (`thread`, `scheduler`, `cyg_libc_main_thread`, `idle_thread`, `thread_obj`, ...) are the most susceptible. For the remainder of this paper, we will focus on the scheduler data structures that are more static in nature, and postpone the protection of dynamically growing and shrinking stack data to future work.

B. Solution Requirements

Our analysis in this section shows that, depending on the structure of the OS and the way it is used, the memory space exhibits "neuralgic spots", i.e., data objects that are much more critical than the remaining memory regions. By protecting only these critical objects, a protection mechanism could improve

THREAD1				
Symbol	**Address**	**Size**	**#Failures**	**(%)**
thread	1,108,640	224	1.008×10^{10}	(39.4 %)
stack	1,108,864	6,816	5.730×10^{9}	(22.4 %)
Cyg_RealTimeClock::rtc	1,124,256	64	3.865×10^{9}	(15.1 %)
Cyg_Scheduler::scheduler	1,117,760	132	8.537×10^{8}	(3.3 %)
Cyg_Interrupt::dsr_list_tail	1,117,744	4	8.530×10^{8}	(3.3 %)
hal_interrupt_objects	1,123,328	896	8.530×10^{8}	(3.3 %)
hal_interrupt_handlers	1,121,536	896	8.530×10^{8}	(3.3 %)
cyg_scheduler_sched_lock	1,108,016	4	8.529×10^{8}	(3.3 %)
pt1	1,115,684	4	8.528×10^{8}	(3.3 %)
Cyg_Scheduler_Base::current_thread	1,117,748	4	7.197×10^{8}	(2.8 %)

MUTEX1				
Symbol	**Address**	**Size**	**#Failures**	**(%)**
stack	1,109,664	10,224	3,189,208	(19.1 %)
thread_obj	1,109,312	352	3,107,236	(18.7 %)
hal_vsr_stats	1,108,800	256	2,721,620	(16.3 %)
cvar1	1,109,264	8	827,282	(5.0 %)
Cyg_Scheduler::scheduler	1,121,984	132	764,988	(4.6 %)
cvar0	1,109,256	8	761,936	(4.6 %)
m1	1,109,244	12	546,710	(3.3 %)
m0	1,109,232	12	523,754	(3.1 %)
Cyg_Interrupt::dsr_list	1,121,964	4	508,128	(3.1 %)
cvar2	1,109,272	8	499,968	(3.0 %)

Table I: Quantitative fault-injection results: Top ten fault-susceptible symbols (or, contiguous memory areas) for the unmodified THREAD1 and MUTEX1 benchmarks.

the system's dependability significantly with only minimal overhead.

However, the approach poses some software engineering challenges: As the set of critical objects depends on the application scenario, the mechanism has to be implemented in a generic way so that it can be reused in all possible scenarios. Ideally, the solution would be modular and completely separated from the protected software component. This would allow developers to reuse the generic protection mechanism in different operating systems or even on the application software level.

III. Generic Object Protection

"A little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless." [18]

Fine-grained protection of kernel objects calls for a fault-tolerance measure that can monitor the data flow between main memory and the protected software component – in our case the eCos operating system. Ideally, the solution would guarantee that when the software reads data, its value is always the same as the last value written into the memory cell, regardless of any bit flips that happened in between. Yet, data flow monitoring is difficult to realize, as we have to deal with low-level infrastructure software that is compiled to machine code. Running the entire eCos in a virtual machine would be one way to approach the problem. However, this is not only a very costly approach but also infeasible for many embedded hardware platforms that do not support virtualization. Therefore, the memory-error protection has to become an integral part of the protected software itself. This could either be achieved by using an extended compiler or, as we did, by means of aspect-oriented programming [12].

During the design of the mechanism, special care has to be taken to avoid any runtime overhead. For instance, comparing all accessed memory locations with a set of monitored address ranges in software at runtime is out of the question. Thus, to provide a highly-efficient mechanism, our solution follows two main design principles:

1) We exploit application knowledge at compile time and, thus, minimize the number of runtime checks.
2) We abandon the aforementioned goal to detect every bit flip in accessed data, but try to balance the trade-off between the cost of injected checks and the gained error-detection rate.

A. Exploiting Object-Oriented Structure

A running program generates a sequence of *read* and *write* operations on memory cells with different addresses. Additionally, each memory cell may be subject to a non-deterministic hardware *fault*, which causes one or more bit flips. While a *write* is not susceptible to a preceding *fault*, a *read* with a preceding *fault* will make the program use wrong data. This might cause program failure. To avoid this, the *write* operation can additionally store some redundant data about the written value, which can be used to detect and even correct bit flips by the *read* operation. However, doing this for every memory cell that is occupied by a critical data structure is very expensive. We thus follow design principle 2 by identifying groups of subsequent *read* and *write* operations with temporal locality. When we find such a group, the check can be performed only once before the first operation of the group and the redundant information about the memory cell can be saved once after the last operation. The underlying assumption is that there are long periods in which the memory cell is unused between one group and the next. If a *fault* happens at a random point in time, the probability that it hits such an inter-group time frame is very high. This means that we can still detect most faults, but have a drastically reduced overhead.

The overhead can be reduced even more if we also merge groups that share a similar memory access pattern. In other words: Two groups whose operations access different memory locations can be merged into a *multi group* if they overlap in time significantly. The implementation can then calculate shared redundancy information over multiple memory cells, which is more efficient. Furthermore, checks and calculations can be performed only once per *multi group*.

The key question for the efficient implementation of the sketched mechanism is how to detect the temporal and spatial connections of *read* and *write* operations at compile time (design principle 1). Object orientation is the most natural solution: If the program was designed in an object-oriented manner, there is an implicit connection between its data objects (instances of classes) and the program code that manipulates them (methods of the class). We can thus approximate a *multi* *group* as the sequence of *read* and *write* operations performed by a method of a class while it manipulates an object. This means that …

- before a method of a critical object is executed, our mechanism checks whether the object suffered from a memory fault.
- after the execution of the method, redundancy information about the object's state is calculated and stored.

Aspect-oriented programming is the implementation technique that we find most suitable for this task.

B. *Exploiting Features of Aspect-oriented Programming*

The idea behind aspect-oriented programming (AOP) is to provide language features that support the modular implementation of *crosscutting concerns*, i.e., concerns of the implementation that affect various different locations of the program in a systematic way. This is achieved by defining rules such as the following:

> *"In programs P, whenever condition C arises, perform action A."* [19]

As *P*, *C*, and *A* can be provided by the programmer, AOP offers a very generic mechanism to instrument arbitrary programs (*P*) with error detection and correction code (*A*) whenever a member function of a critical object is executed (*C*). A tool called *aspect weaver*, which typically performs a code transformation at compile time, makes sure that the demanded adaptation of the control flow is actually performed. Besides this *adaptation* mechanism, aspect-oriented languages typically also provide an *introspection* mechanism, which allows the programmer of the rules to write generic actions that may depend on the target program's structure. Aspect-oriented language extensions are, for instance, available for Java (AspectJ [20]) and C++ (AspectC++ [21]). The latter is strongly focused on compile-time code adaptation and can exploit the C++ template mechanism for powerful *code synthesis*. With these three ingredients, namely the code *adaptation* feature, the *introspection* mechanism, and C++'s code *synthesis* capabilities, AspectC++ is a powerful tool for writing reusable fault-tolerance mechanisms, which are *woven* into the protected software component at compile time.

Figure 2 on the following page shows a simplified version of our generic object-protection mechanism written in the AspectC++ language. The aforementioned rules are defined with the `advice` keyword, as in lines 3, 8 and 10. In AspectC++, rules (advice definitions) that implement a common concern are grouped in an *aspect*. The definition of our `GenericObjectProtection` aspect starts in line 1 with the keyword `aspect`. One of the benefits of aspect-oriented programming over other implementation techniques is that for crosscutting concerns the source code almost directly reflects the software developer's intention. For example, the pieces of advice in lines 8 and 10 are almost a literal translation of the two rules mentioned at the end of Section III-A on the previous page: In line 11, a function `check()` is called *before* any *call* to a member function of a *protected class*. In line 9, a function `update()` is called *after* the *construction* of a protected class' instance or a member function *call*.

The built-in pointer `tjp`[1] can be used by advice code to access context information about the condition that triggered its execution in a generic way. `tjp->target()` yields the target object of the construction or function call, respectively. Besides the `target()` function, the AspectC++ JoinPoint-API [22] provides much more context information, especially static type information such as the type of the calling and the called object (`JoinPoint::That` and `JoinPoint::Target`).

Advice definitions are also generic in the sense that they use the *pointcut* `protectedClasses()` to address the points of adaptation. A *pointcut* is merely an alias for a reusable part of a condition. In line 2 it is defined to match the `Cyg_Scheduler` class and the `Cyg_Thread` class.[2]

In AspectC++ the adaptation mechanism can not only affect the control flow – it can also inject structural extensions. The advice in line 3 demonstrates this feature. Here the protected classes are extended by three new members: A data member `replica`, which will store the object's information redundantly, and the two member functions `check()` and `update()`. The details of the implementation can be easily replaced to support different protection algorithms, e.g., using Hamming code, cyclic redundancy checks (CRC), or triple-modular redundancy (TMR). An essential language feature needed by our protection mechanisms is, again, the `JoinPoint` type. For structural extensions, this built-in type is the interface to the introspection mechanism of AspectC++, which provides the injected members with information about the target type of the extension. For example, it describes all data members of the target class including their type. We can exploit this information by using it as a parameter for *generative* C++ template metaprograms, such as the `JPTL::MemberIterator`. It is important to note that a template metaprogram "runs" at compile time. This means that it does not consume any runtime. It is a powerful, *Turing-complete* [23] code-synthesis mechanism.

C. *Summary of the Benefits*

In summary, our approach is an economic kind of software-based fault-tolerance mechanism, which can be used to detect and correct transient memory faults. It exploits the temporal locality of memory access operations in member functions of classes in object-oriented software. Section V will present the achieved error detection and correction rate in comparison to the performance overhead and increased code size.

From the software-engineering point of view the implementation is also interesting. Due to aspect-oriented programming the solution is very convenient for users. The source code of the protected software component does not have to be changed for deploying a reusable memory-error-protection aspect. It suffices to list the most critical classes in a pointcut definition. The design also facilitates the selection of application-specific fault models. For example, in a very harsh or critical environment one could deploy an aspect that can deal with multiple bit flips in the same object. Finally, as the approach is based on source-to-source transformation with AspectC++ (*weaving*), it is inherently portable.

[1] *tjp* is an abbreviation for "this joinpoint".

[2] AspectC++ also allows programmers to declare *pointcuts* as "pure virtual", which means that they can be defined in a *derived aspect*. We have not shown this here to simplify the listing, but in practice it is a basic mechanism needed for writing completely generic and reusable aspect libraries.

```
aspect GenericObjectProtection {
  pointcut protectedClasses() = "Cyg_Scheduler" || "Cyg_Thread"; // list of critical eCos classes
  advice protectedClasses() : slice class { // generic class extension ("introduction")
    char replica[JPTL::MemberIterator<JoinPoint, SizeOfNonPublic>::EXEC::SIZE];   // redundancy data
    void check()  { JPTL::MemberIterator<JoinPoint, CheckReplica>::exec(this); }  // detect/handle errors
    void update() { JPTL::MemberIterator<JoinPoint, UpdateReplica>::exec(this); } // recalculate 'replica'
  };
  advice construction(protectedClasses()) || call(protectedClasses()) : after() {
    tjp->target()->update(); } // generic advice
  advice call(protectedClasses()) : before() {
    tjp->target()->check(); } };
```

Figure 2: An implementation of the generic object-protection mechanism written in AspectC++.

IV. Design Challenges and Decisions

This section discusses the challenges that we encountered during design and implementation of the generic object-protection algorithm. These challenges relate to particular details of the algorithm that have not been addressed in the previous section.

A. Selection of Data Members to Protect

The fundamental idea of our approach is to exploit that data members are only accessed within member functions of the same class. However, this is not the case for `public` data members, which are readable and writable by arbitrary program statements. Anyway, `public` data members are rarely used in object-oriented software[3], and we are convinced that it is feasible to exclude such members from the generic object protection. Therefore, we need information about the protection level (either `public`, `protected` or `private`) of each data member to determine whether it is supposed to be covered by the object protection. This information is provided by AspectC++'s compile-time introspection feature for structural extensions: `JoinPoint::MEMBERS` reflects the number of data members of the target class, and `JoinPoint::Member<I>::prot` encodes the protection level of the I^{th} member. In a similar way, its `Type` as well as a `pointer` to the member can be obtained. Given this introspection information, it is feasible to write a single *generative* C++ template metaprogram that iterates over all data members of any class instance. Then, arbitrary computations – for example, checksum calculations – can be generated, additionally filtering out `public` data members. Thus, the error detection (EDM) and error-recovery mechanisms (ERM) used throughout this paper are formulated as generative template metaprograms that rest upon AspectC++'s introspection feature.

B. Allocation of Redundancy

The amount of redundancy needed for a particular data structure depends on the deployed EDM/ERM. To guarantee a certain error detection and correction probability, the total amount of redundancy has to grow with the protected data's size (in terms of bits). Therefore, the structural extension (see Section III-B) of a target class $\mathcal{C}$ by a data member that holds the redundancy $\mathcal{R}$ implies that the redundant-data-member's size depends on the size of the target class:

[3]The object-oriented paradigm encourages developers to declare important data as `protected` or `private` to restrict access and prevent unwanted modifications by other software components.

$$\mathcal{C} = \{data\,members,\ \mathcal{R}\}$$

$$sizeof\,(\mathcal{R}) = r \cdot sizeof\,(\mathcal{C})$$

The factor r is specified by the particular EDM/ERM, for example, r is 2 for TMR. Such an equation is obviously a paradox, as a growth of $\mathcal{C}$ leads to a growth of $\mathcal{R}$ and the other way around. A solution for this problem is to define the size of $\mathcal{R}$ to depend only on the size of the *data members* of $\mathcal{C}$ without $\mathcal{R}$. Unfortunately, there is no generic way to express this solution in plain C/C++, but the compile-time introspection feature of the AspectC++ compiler allows to implement a template metaprogram that iterates over all data members *prior* to the introduction of redundancy. By this means, the built-in `sizeof` operator can be applied to every data member independently and the results are added up by the metaprogram. This sum is a compile-time constant and, for instance, can be used to introduce an array of redundant bytes. The main benefit of this solution is that unwanted data members can be excluded from the size calculation, for example `public` data members as well as compiler-generated alignment padding inside a data structure.

C. Object Composition

The next challenge that we encountered is the composition of objects. Let the class $\mathcal{C}$ contain a class-type member $\mathcal{C}_{sub}$ plus redundancy: $\mathcal{C} = \{\mathcal{C}_{sub}, \ldots, \mathcal{R}\}$. Given this definition, the subobject – an instance of $\mathcal{C}_{sub}$ – would be protected twice, both by $\mathcal{R}$ and its own redundancy $\mathcal{R}_{sub}$. Thus, we decided to exclude subobjects from the generic object protection, so that subobjects are only protected once. We implemented the exclusion of subobjects by C++ *type traits* [24], which is a template-based technique that allows to make decisions based on types, for instance by testing whether a data member is of class type (subobject), a pointer, an integer, and so on. Additionally, this technique offers a way to tailor the generic object protection to cover only particular data members, for instance just pointers. We have not further investigated this opportunity yet.

D. Static Call-site Analysis

The fundamental idea of generic object protection is to associate error-correction-and-detection with member functions of data structures. Before a member function is called, checks are performed, and after return from that function, the redundancy information is recalculated. These actions can take place either at the *caller* or the *callee*. The former approach results in $\mathcal{O}(\#call\,sites)$ complexity, whereas the latter leads to $\mathcal{O}(\#member\,functions)$. In general, when the number of call sites and member functions are unknown, both solutions are feasible. However, the call-site approach offers the advantage of providing knowledge about the caller. Consider two member-functions f_1 and f_2 of the same data structure, and f_1 calls f_2 at some point in the dynamic control flow, for example: $main() \rightarrow obj.f_1() \rightarrow obj.f_2()$. Then, concerning the call from f_1 to f_2, any checks and redundancy recalculations should be omitted, because the check would immediately be succeeded by the recalculation. The decision to omit checks/recalculations on such call sites can be taken at compile time by a static analysis of the call relationships. Basically, the *caller* and the *callee* must be compared, and only if both refer to same data structure, the protection mechanism can be skipped.

The JoinPoint API of AspectC++ (see Section III-B) provides the necessary information to implement such a static call-site analysis. In the body of `call`-advice code, the class type of the caller (`JoinPoint::That`) as well as the class type of the callee (`JoinPoint::Target`) are exposed by the JoinPoint API. These class types can be tested on equality by C++ type traits. Moreover, pointers to the caller/callee objects can be obtained by `tjp->that()` and `tjp->target()` respectively, which can be compared directly. The comparison of types is necessarily a compile-time decision. Comparing object-pointers boils down to testing C++'s built-in `this` pointers, which can be optimized out when the result is known at compile time. Therefore, we decided to exploit this static information to minimize the runtime checks by choosing the call-site approach paired with static analyses.

This design decision further enables the minimization of the time window between checks and redundancy recalculations, because outgoing function calls that leave a protected data-structure can be handled as well. As an example, consider a member function f that calls the C-library function `printf()`. Then, inside f, the call to `printf()` can be enclosed by inverted EDM/ERM-actions: $f_{recalculate()} \rightarrow printf() \rightarrow f_{check()}$. Thus, during the execution of `printf()`, the data structure of f is safe. These additional checks/recalculations clearly increase the overhead of our approach (mostly code size), but greatly improve error detection and correction capabilities. This is especially true for calls that block the running process.

E. Inheritance

Special attention has to be paid to the object-oriented principle of *inheritance*. In C++, a *base* class can be extended by a *derived* class, and the derived class inherits all members from the base class. Members of the base class are directly accessible in the derived class (except for members that are declared `private`). When a function of a derived class is called, the data members of *all* its base classes have to be verified. This is the case for the aforementioned classes `Cyg_Scheduler` and `Cyg_Thread` of eCos, which inherit from four base classes each.

The information about base classes of an arbitrary class cannot be retrieved in plain C++, and again, requires compile-time introspection for being implemented. The AspectC++ compiler provides a template-based list of all base classes (`JoinPoint::BaseClass<I>`) in a similar way as the information about data members is provided. By this means, a generative C++ template metaprogram can iterate over all base classes and invoke check/recalculate actions on each of them. Additionally, such an iteration has to be performed recursively, that is, to iterate over the bases of the bases, and so on.

The second challenge with respect to inheritance is the dynamic dispatch of virtual functions. A virtual function, implemented in a base class, can be *overridden* by several derived classes. When such a function is called, the actual function's implementation is not chosen before runtime. Hence, it is impossible to determine the callee object type of a virtual-function call at compile time. This uncertainty conflicts with our static call-site analysis approach. Therefore, we decided to complement the static analysis by a dynamic dispatch of check/recalculate actions. For classes with inheritance, the functions that check/recalculate the redundancy become virtual functions, so that their invocation is dispatched to the most derived class belonging to a particular object. After the dynamic dispatch to the most derived class happened, the base classes are processed as described above. In summary, data structures that are built from several base classes are treated holistically by the base-class iteration plus dynamic dispatch. However, the decision whether a class is covered by the generic object protection can be taken independently of its inheritance relations. Furthermore, any function calls inside such an inheritance hierarchy can be detected by the static call-site analysis, and the object-protection mechanism can be optimized out for such cases.

F. Multi-threading

An important requirement to the generic object protection is the support for multi-threading, as needed when applied to the eCos operating system. By support for multi-threading we mean that our approach works *correctly* for data structures that are used concurrently by multiple threads – and *not* that our implementation itself spawns several threads to perform its duty in parallel.

Our first observation is that every EDM/ERM operation (check/recalculate) on shared data-structures must be *atomic*, considering a thread that is preempted while verifying a checksum. In the meanwhile, other threads could alter the particular data structure and its checksum, so that, when the suspended thread is resumed, it finds an inconsistent state. Thus, concurrent EDM/ERM operations must be serialized by synchronization primitives, such as a semaphore/mutex, a kernel lock, or by suppressing interrupts. The choice of locking mechanism depends on the actual data structure, e.g., data structures used during interrupts require to suppress them while performing EDM/ERM actions. User-level-only data structures, on the other hand, can be synchronized by a mutex, and obviously, non-shared data structures need no synchronization at all. We implemented this differentiation by configurable `synchronized()` pointcuts[4]. Such a pointcut is a textual list, provided by the user, of all classes that are covered by a particular synchronization mechanism – in our case `Cyg_Scheduler`, `Cyg_Thread` and their base classes.

Our second observation is that, as soon as a data structure is being modified, its checksum is no longer valid, so that

[4]The definition of `synchronized()` pointcuts is carried out in the same way as the `protectedClasses()` pointcut in Section III-B.

all concurrent checks must be skipped until the checksum is valid again. Therefore, we introduced a per-object counter into each shared class instance. This counter is incremented before a thread enters a potentially modifying, that is non-`const`, member function. The counter is decremented after the thread leaves the non-`const` member function. In fact, such a per-object counter reflects the number of concurrent threads that may modify an object. All EDM/ERM operations are skipped unless this counter is 1, which means, until there is only a single thread accessing the object. In summary, each function call to a shared object is synchronized by 9 steps:

1) **acquire** lock
2) increment per-object counter
3) `check()` the checksum if counter == 1
4) **release** lock
5) enter the function ... until it returns
6) **acquire** lock
7) `recalculate()` the checksum if counter == 1
8) decrement per-object counter
9) **release** lock

This sequence guarantees atomicity due to the locking and furthermore elides unnecessary EDM/ERM operations. Another important property is that this sequence cannot cause deadlocks, since the second *Coffman condition* (hold and "wait for" resources) [25] is not satisfied. After acquiring the lock, the EDM/ERM operation runs to completion, and the lock is released straightaway without waiting for additional resources.

Finally, the per-object counters have to be resilient against memory errors on their own respect. We implemented these counters by tripling their state plus majority voting. This solution is feasible, since a counter's state is only an integer variable, so that tripling is inexpensive.

V. Implementation and Evaluation

In the following, we describe the implementation of five concrete error-detection and error-recovery algorithms based on the generic object-protection mechanism. Subsequently, we quantitatively evaluate their effectiveness in a set of benchmarks with fault-injection (FI) experiments, and measure the induced static and dynamic overhead. This allows us to predict the suitability of specific EDM/ERM variants for yet unknown scenarios, and to draw conclusions on the overall methodology.

A. EDM/ERM Variants

To exemplarily evaluate the generic object-protection mechanism described in Sections III and IV, we implemented five generic EDM/ERM aspects. In Table II we introduce acronyms for each variant for reference in the rest of this section ("Baseline" is the unmodified version without protection), and present canonicalized[5] lines-of-code per module to convey an estimate of the complexity.

Each variant is implemented as a generic module and can be configured to protect any subset of the existing C++ classes of the target system.

[5]Effective lines of code (excluding empty lines and comments), obtained with *cloc*: http://cloc.sourceforge.net/

B. Evaluation Setup & Fault Model

We evaluated each protection variant on eCos 3.0 with a set of 21 benchmark and test programs that is bundled with eCos itself, and which constitutes a subset of all eCos benchmarks that is implemented in C++ and utilizes threads. Table III briefly describes each benchmark and records its number of dynamic function calls to the protected scheduler and thread classes. Including the baseline variant, this set totals at 126 variant/benchmark combinations. All binaries were compiled for i386 with the GNU C++ compiler (GCC, eCosCentric GNU tools 4.3.2-sw, optimization level -O2); eCos was set up with its default configuration, *grub* startup and the *bitmap* scheduler variant. Additionally, we disabled both serial and VGA output, as the benchmarks report on success or failure before finishing, and such time-consuming output would completely mask out any EDM/ERM runtime overhead.

We chose two different fault models to emphasize the flexibility in our solution space. First, we used the common uniformly-distributed **transient single-bit flip fault model** in data memory, i.e., we consider program runs in which a single bit in the data/BSS segments flips at some point in time. This model seems reasonable for low-cost embedded systems where read-only data and code (text) is stored in far less susceptible (EEP)ROM or Flash, and global objects and the program stack is kept in non-ECC RAM. Second, we used a **burst bit-flip model**, which flips *all eight* bits at a memory address at once. This models, e.g., multi-bit impacts of high energy events.

Bochs, the IA-32 (x86) emulator back end that the FAIL* experimentation framework [17] currently provides, was configured to simulate a modern 2.666 GHz x86 CPU. It simulates the CPU on a behavior level with a simplistic timing model of one instruction per cycle (with the exception of the HLT instruction, which spans multiple cycles until the next interrupt), and does not provide any insights on caching and pipelining effects. Therefore the results obtained from injecting memory errors in this simulator are very pessimistic: We expect that a contemporary cache hierarchy would mask many main-memory bit flips (especially for longer-running benchmarks).

C. Effectiveness: Error Detection & Correction

To keep the FI experimentation efforts within reasonable limits, we skipped the three longest-running benchmarks (KILL, MUTEX3 and CLOCKTRUTH – the latter running for almost 2.7 billion instructions) and the Hamming protection variant with its extremely high runtime overhead (cf. Section V-D). Furthermore we modified the BIN_SEM2 and SYNC2 benchmarks to run for 100 instead of 1,000 main loop iterations, and extrapolated the results for the original version by multiplying the failure counts by 10; this simplification is valid due to the highly repetitive nature of these benchmarks. Additionally we constrained the fault space to the scheduler and thread data structures within the kernel: This naturally biases the FI results towards a better coverage, but as we do not expect any resiliency-wise different behavior from all other memory areas, we would not gain additional insights from injecting faults there. Therefore, the following results are only valid for the protected data structures, while the *non-critical* rest of the memory space (cf. Section II) remains as susceptible as before.

Aspect/Module	Description	LOC
CRC	A CRC32 implementation leveraging Intel's SSE4.2 instructions (EDM).	134
TMR	Triple-modular redundancy, using two copies of each data member and majority voting (EDM/ERM).	86
CRC+DMR	CRC (EDM, see above), plus one copy of each data member for additional error correction (ERM).	171
SUM+DMR	A 32-bit two's complement addition checksum (EDM), plus one copy of each data member (ERM).	157
Hamming	Software-implemented Hamming code (240 data bits, 8 parity bits; single-bit EDM/ERM).	166
Framework	Generic object-protection infrastructure, the basis for all concrete EDM/ERM implementations.	1,581

Table II: EDM/ERM variants.

Benchmark	Description / Testing domain	Syscalls	Benchmark	Description / Testing domain	Syscalls
BIN_SEM1	Binary semaphore functionality *(2 threads)*	323	**MUTEX1**	Basic mutex functionality *(3 threads)*	743
BIN_SEM2	Dining philosophers *(15 threads)*	92,711	**MUTEX2**	Mutex release functionality *(4 threads)*	743
BIN_SEM3	Binary semaphore timeout *(2 threads)*	602	**MUTEX3**	Mutex priority inheritance *(7 threads)*	19,812
CLOCK1	Kernel Real Time Clock (RTC) *(1 thread)*	2,851	**RELEASE**	Thread release() *(2 threads)*	641
CLOCKCNV	Kernel RTC converter subsystem *(1 thread)*	379	**SCHED1**	Basic scheduler functions *(2 threads)*	94
CLOCKTRUTH	Kernel RTC accuracy *(1 thread)*	39,839	**SYNC2**	Different locking mechanisms *(4 threads)*	437,314
CNT_SEM1	Counting semaphore functionality *(2 threads)*	370	**SYNC3**	Priorities and priority inheritance *(3 threads)*	385
EXCEPT1	Exception functionality *(1 thread)*	171	**THREAD0**	Thread constructors/destructors *(1 thread)*	72
FLAG1	Flag functionality *(3 threads)*	1,356	**THREAD1**	Basic thread functions *(2 threads)*	266
KILL	Thread kill() and reinitalize() *(3 threads)*	874	**THREAD2**	Scheduler and thread priorities *(3 threads)*	685
MQUEUE1	Message queues *(2 threads)*	922			

Table III: eCos kernel test benchmarks. The number of dynamic system calls (Syscalls) to the protected scheduler and thread classes is shown in the last column.

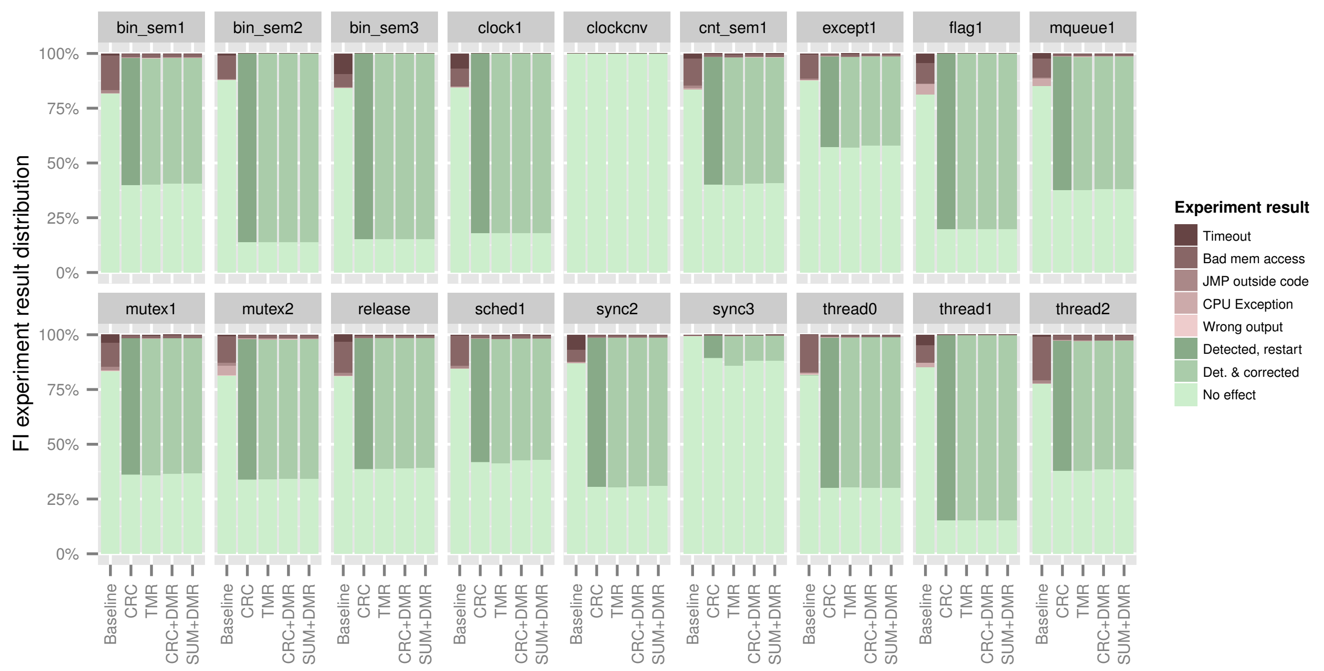

Figure 3: Single-bit flip FI campaign results (KILL, MUTEX3 and CLOCKTRUTH benchmarks omitted due to their extremely long runtime) in percentages of their respective fault-space size (benchmark runtime × critical-data memory size).

	Baseline	CRC	TMR	CRC+DMR	SUM+DMR		Baseline	CRC	TMR	CRC+DMR	SUM+DMR
BIN_SEM1	2.395×10^{6}	5.416×10^{6}	8.546×10^{6}	6.446×10^{6}	6.355×10^{6}	**MUTEX1**	4.516×10^{6}	9.472×10^{6}	1.491×10^{7}	1.132×10^{7}	1.108×10^{7}
BIN_SEM2	5.075×10^{12}	2.637×10^{9}	3.880×10^{9}	3.178×10^{9}	3.150×10^{9}	**MUTEX2**	1.262×10^{7}	1.690×10^{7}	2.661×10^{7}	2.039×10^{7}	2.001×10^{7}
BIN_SEM3	6.575×10^{10}	1.042×10^{7}	1.583×10^{7}	1.243×10^{7}	1.232×10^{7}	**RELEASE**	2.535×10^{6}	5.324×10^{6}	8.376×10^{6}	6.309×10^{6}	6.236×10^{6}
CLOCK1	1.288×10^{12}	1.112×10^{7}	1.577×10^{7}	1.267×10^{7}	1.278×10^{7}	**SCHED1**	1.580×10^{6}	1.724×10^{6}	2.850×10^{6}	2.045×10^{6}	1.993×10^{6}
CLOCKCNV	6.594×10^{5}	8.997×10^{5}	1.461×10^{6}	1.066×10^{6}	1.037×10^{6}	**SYNC2**	8.551×10^{8}	4.287×10^{9}	6.645×10^{9}	5.127×10^{9}	5.024×10^{9}
CNT_SEM1	2.130×10^{6}	4.526×10^{6}	7.168×10^{6}	5.345×10^{6}	5.252×10^{6}	**SYNC3**	5.289×10^{6}	9.654×10^{6}	1.532×10^{7}	1.157×10^{7}	1.135×10^{7}
EXCEPT1	8.498×10^{5}	1.060×10^{6}	1.665×10^{6}	1.240×10^{6}	1.206×10^{6}	**THREAD0**	9.211×10^{5}	7.232×10^{5}	1.139×10^{6}	8.048×10^{5}	7.805×10^{5}
FLAG1	2.066×10^{11}	1.911×10^{7}	2.899×10^{7}	2.274×10^{7}	2.243×10^{7}	**THREAD1**	5.894×10^{9}	7.134×10^{6}	1.089×10^{7}	8.479×10^{6}	8.363×10^{6}
MQUEUE1	5.957×10^{6}	8.509×10^{6}	1.338×10^{7}	1.012×10^{7}	9.967×10^{6}	**THREAD2**	6.499×10^{6}	1.839×10^{7}	2.957×10^{7}	2.223×10^{7}	2.176×10^{7}

Table IV: Absolute failure counts for FI campaign (KILL, MUTEX3 and CLOCKTRUTH benchmarks omitted due to their extremely long runtime).

For both fault models and the remaining 90 variant/benchmark combinations (including the baseline), and after applying FAIL*'s conservative fault-space pruning techniques to efficiently cover the complete fault space, we conducted a total of about 274 million FI experiments. Figure 3 shows the FI result distribution for the single-bit flip fault model, divided into positive (*no effect*, *error detected and corrected*, and *error detected*) and various negative outcomes: The **CRC** variant successfully detects (the only protection variant that only *detects* but not corrects) scheduler and thread data-structure errors in almost all cases – over all benchmarks, an average of 15.3 % faults have no effect, 84.7 % are detected, and only 0.01 % (from a previous average of 12.8 % in the baseline) still fail. The other evaluated variants (**TMR**, **CRC+DMR**, **SUM+DMR**) turn out equally good (± 0.1 %), with the difference that 84.7 % are detected *and corrected*. With the burst fault model, the numbers only vary insignificantly (plot not shown): The baseline fails in marginally more cases (13.6 %), but the evaluated EDM/ERM schemes detect/correct the same percentage of errors.

The experiment outcome numbers *relative to the fault-space size* from the previous paragraph are contrasted by the *absolute* numbers in Table IV. As software-implemented error protection introduces additional runtime overhead, protected variants naturally *run longer*, increasing the chance of being hit by memory bit-flips (which we assume to be uniformly distributed). Consequently, there exists a break-even point between, metaphorically, quickly crossing the battlefield, and running slower but with heavy armor. Table IV suggests that for 13 of the benchmarks, this break-even point is approximately met (the error counts with and without protection are within the same order of magnitude), but we do not readily gain a real advantage from running slowly. For the remaining 5 – note these are among the longest-running among the 18 analyzed! – the break-even was clearly outpaced and the absolute failure numbers are reduced by several orders of magnitude. We conclude that our protection mechanisms can be applied to almost any long-running embedded application with a significant net resiliency improvement.

D. Efficiency: Static and Runtime Overhead

Although the previous subsection illustrated that our protection mechanisms increase the system resiliency in many cases, they come at different static and dynamic cost. In the following we present code size and runtime measurements to put this cost in relation with the benefits gained.

Figure 4 shows the static binary sizes of (due to space constraints) a selection of variants. The DATA sections of all binaries are negligibly tiny (around 500 bytes) and stay constant in size; BSS also remains mostly constant with different protection variants (max. increase compared to the baseline is 3.6 %). The code size (TEXT) increases vary extremely between the different variants: While **CRC** increases the code by an average of 58 % (**SUM+DMR**: 74 %, **CRC+DMR**: 79 %, **TMR**: 105 %), the **Hamming** variant costs a whopping average of 146 %.

Depending on the benchmark, the protected code sections are executed more or less often, resulting in highly varying runtime overhead. We deployed all variant/benchmark combinations on a contemporary Intel Core i7-M620 notebook running at 2.66 GHz and measured their real-world timing behavior (with the `RDTSCP` CPU instruction). The total real-world runtime corresponds accurately (99.8 %) to the simplistic timing model of our simulation, aside from the EXCEPT1 benchmark, which triggers machine-dependent CPU exceptions that execute hundredfold slower on real hardware. Figure 5 shows that the results can be classified into two categories: While some benchmarks continuously invoke the scheduler/thread objects and therefore lead to heavy impact on the runtime overhead, others access the scheduler infrequently and execute in almost the same runtime. The former category of benchmarks, in particular SYNC2, constitutes the *pathologic use case* for our protection scheme: It continuously bombards the scheduler with accesses and spends almost no time in the benchmark application itself. SYNC2 stands out extremely, running around 18 times longer for the **CRC** and **SUM+DMR** protection variants, 57 times for **TMR** and even 468 times for the dreaded **Hamming** code. Because the **Hamming** protection scheme consistently causes an order of magnitude higher overhead, we omitted to plot this variant. The second category of benchmarks – running for 10^7 clock cycles or longer in Figure 5 – shows way more encouraging results. These benchmarks contain a realistic application profile, that is, a mix of computation, idle phases and scheduler invocations. For these long-running benchmarks, the runtime costs stay well below 1 % in most cases and can be considered negligible. In conformity with Amdahl's law, the runtime overhead for the whole benchmark and test suite totals at only 0.09 % for the **SUM+DMR**, **CRC** and **CRC+DMR** protection variants, followed by **TMR** and **Hamming** with 0.23 % and 1.75 % respectively.

E. Discussion

The evaluation shows that, for our set of benchmarks, the EDM/ERMs come at different levels of overhead. Most protection mechanisms have little total runtime overhead. Overall, the **SUM+DMR** variant seems to offer the best cost/benefit ratio in most cases, and shows a negligible slowdown on real hardware; if a detection-only mechanism suffices for the use case, the **CRC** protection may be a reasonable choice. Both mechanisms work fast and efficiently at machine-word granularity (in our case 32 bit). The only EDM/ERM that operates at bit level is the **Hamming** protection, which turns out to be by far the most inefficient choice. **TMR** has no real benefit – at least when single-bit and 8-bit-burst faults are considered – and should not be used in favor of **SUM+DMR** and **CRC+DMR**.

VI. RELATED WORK

Protecting computer systems' memories against errors is a concern remaining from the mainframe era. The persistence of this problem indicates that there might be no "one size fits all" solution. Hence, there is a large body of work attacking this problem. We classify related work into three categories: susceptibility analyses of operating systems to memory errors, hardware-supported memory protection, and software-based memory protection.

A. Susceptibility Analyses of Operating Systems to Memory Errors

Several studies have addressed the assessment of operating systems in the presence of hardware faults. Already in the

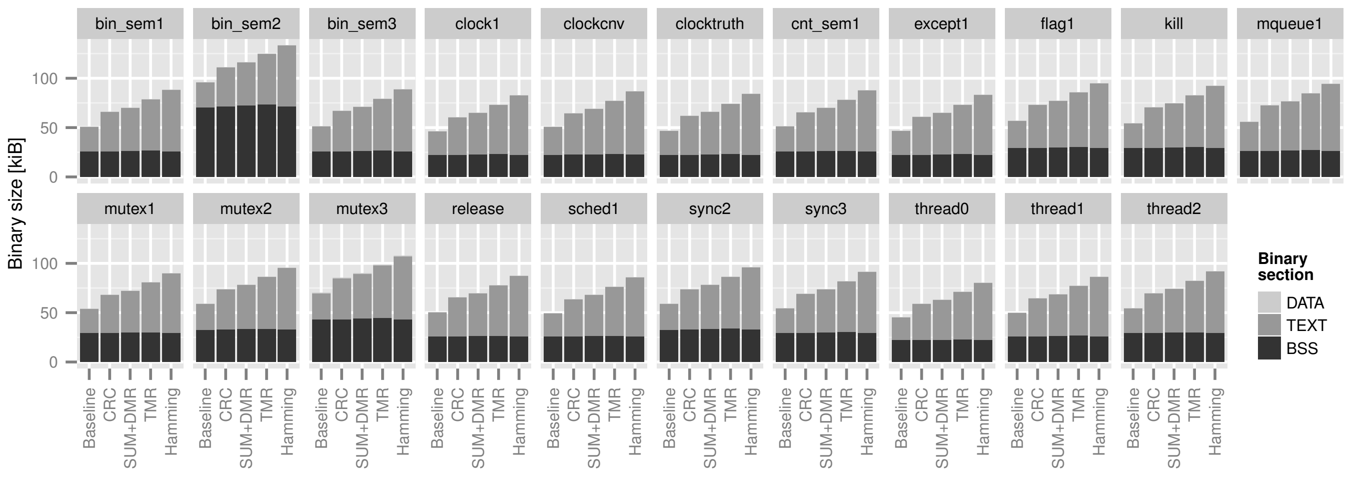

Figure 4: Code size of selected protection variants: The TEXT segment grows due to additional CPU instructions for each EDM/ERM, with Hamming being the most inefficient variant.

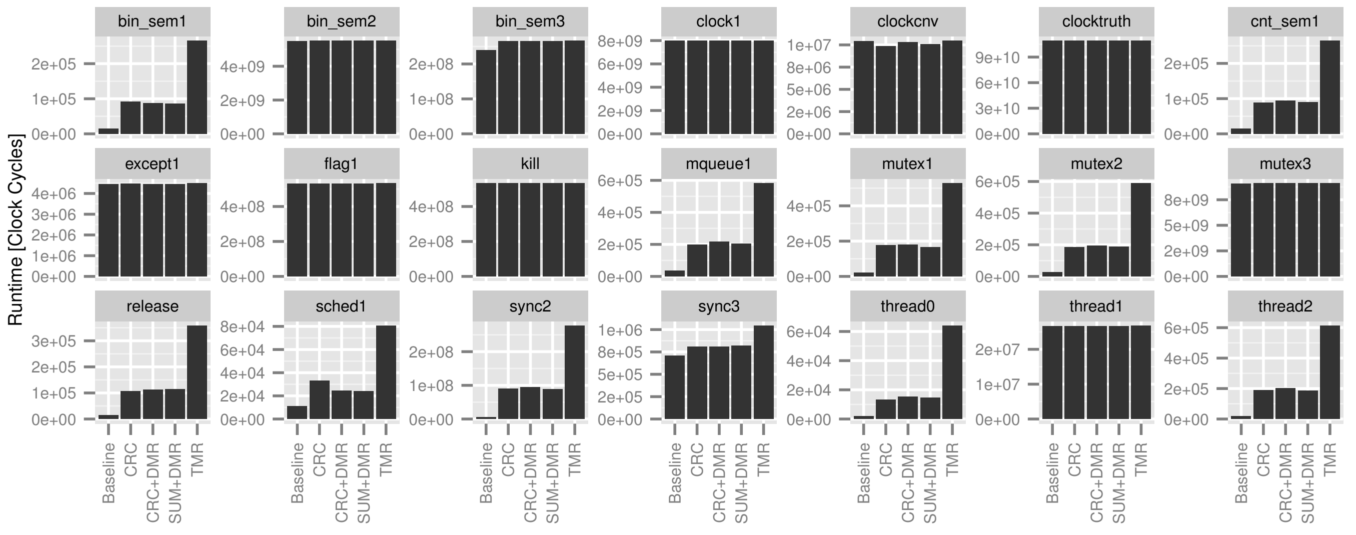

Figure 5: Real-world runtime, measured on an Intel Core i7-M620 notebook: Eleven out of 21 benchmarks run very short (in the order of 10^4 to 10^6 clock cycles) and continuously invoke the operating system's scheduler, resulting in high overhead for each EDM/ERM. These benchmarks are actually the *pathologic use case* for our protection scheme. The remaining ten benchmarks exhibit a more realistic application profile. The scheduler data structures are accessed infrequently in the benchmarks' control flows, so that the runtime costs stay well below 1 % in most cases and can be considered negligible.

nineties, Kao et al. [26] injected memory faults into the kernel address space of a UNIX operating system. Fabre et al. [27] performed similar fault-injection experiments with a microkernel operating system, and so did Madeira et al. [28] with a UNIX-like real-time operating system. More recently, the Linux kernel has been analyzed [29], [30].

These studies ground on a *few thousand* faults being randomly injected, compared to *millions* of instructions executed by the operating systems. We doubt their general applicability, as it is unclear – to the best of our knowledge – to which extent statistically significant conclusions are drawn from sampled fault-injection experiments. Our work differs in that we cover the whole fault space and do not rely on random sampling. Moreover, we provide insight into particular operating-system data structures, such as *Scheduler* and *Thread* objects.

B. Hardware-supported Memory Protection

Commodity ECC DIMMs store 8 bits of redundancy for 64 bits of data, yielding in *single-bit-error correction and double-bit-error detection* (SEC-DED) with uniform storage overhead of 12.5 %. Commercial *Chipkill* [8] improves this rather weak error-correction scheme in tolerating word-wise multi-bit errors (typically 4 adjacent bits) by interleaving a word to independent DRAM chips, at the cost of up to 30 % higher energy consumption due to forced narrow-I/O configuration [9]. Therefore, several hardware modifications to memory controller and memory management units (MMU) have been proposed [9], [31], increasing storage overhead up to 18.75 % and 26.5 %.

The MMU has also been used to manage page-level EDMs [32]. Pages are checksummed and read/write permissions are withdrawn after a timeout, so that the next access on such a page results in a trap, which is used to verify the checksum and restore page permissions. This approach suffers from high runtime-overhead (25 % to 53 %), and, although the authors did not evaluate error-detection capabilities, we are convinced that due to the coarse page-granularity (4 KiB) and the timeout mechanism, many errors are *not* detected.

Another technique is to retire pages that have seen errors [4], [33]. We regard this approach as complementary, because it is only useful against permanent errors and does not prevent transients.

C. Software-based Memory Protection

Researchers investigated the dynamic heap for allocating reliable memory at runtime. *Samurai* [15] is a C/C++ dynamic memory allocator that uses replication of memory chunks. Applications have to be manually modified to use the Samurai API for access to reliable heap memory, which involves checking and updating of the replicas. This approach exposes the heap allocator as single point of failure, which cannot be recovered when hit internally by a memory error. Finally, there is no support for multi-threading, which renders Samurai a poor match for protecting an operating system. Chen et al. [14] describe a heap memory allocator for a Java virtual machine that adds checksums to each allocated object. These checksums are verified and generated on execution of particular byte-code instructions (for example `getfield`/`putfield`). Their evaluation shows less than 40 % error detection at 32–57 % runtime overhead due to many unnecessary checks. Our static call-site analysis (see Section IV-D) avoids such unnecessary checks. Additionally, by its very nature, any reliable heap allocator does not protect data stored in data/BSS segments and on the stack, which is common for operating systems.

Compilers are also an appealing target for transforming non-fault-tolerant software into fault-tolerant implementations. Fetzer et al. [16] use arithmetic *AN*-encoding of memory (among other methods) to detect errors by essentially doubling the storage space for encoded values. Even at this high level of redundancy, recovery is unaddressed. Chang et al. [34] apply triple-redundant execution plus AN-encoding to protect the register file. Their compiler-implemented approach pointed out similar windows of vulnerability compared to our work, that is, "between validation and use" [34] of replicas and "before a value is copied" [34]. Code-transformation rules for source-to-source compilers have been proposed in [35], [13], [36]. These approaches are based on duplicating or even triplicating important *variables* of single-threaded user-level programs. The studies somehow "reinvent the wheel" by implementing a proof-of-concept source-to-source compiler (if ever) – a tedious task for C/C++, being far from complete [35], [13]. Our work differs in that we use the mature general-purpose AspectC++ compiler that allows us to focus on the implementation of software-based EDM/ERMs in the OS/application layer, instead of implementing special-purpose compilers.

Fault tolerance in the OS/application layer bears the invaluable advantage of tailored application-specific measures. However, we are convinced that the fault-tolerance concerns should be separated from the "business logic" of the application to reduce complexity. Several researchers attacked this modularity problem by exercising aspect-oriented programming, in particular AspectC++ [37], [38]. For example, Alexandersson et al. [38] implemented triple-time-redundant execution and control-flow checking as a proof of concept, which led to 300 % runtime overhead. However, none of these approaches addresses memory errors.

Finally, robust data structures can deal with memory errors. Taylor et al. [18] proposed linear lists and binary trees that use redundant pointers but leave the payload unprotected. Aumann et al. [39] formalized a similar approach and extended it to fault-tolerant stack data structures. A formal methodology for the specification of *invariants* (constraints that are satisfied by correct data structures) is presented in [40]. Automatic detection and repair of errors by validating user-defined invariants at runtime has been studied in [41]. Thus, the design of robust data structures requires an excellent understanding of the software. Moreover, it is essential that appropriate invariants *exist*, which are often difficult – or even impossible – to specify. Our generic object protection, however, is very easy to apply to existing software, because we do not require comprehensive knowledge of the system nor do we need to modify it.

VII. Conclusions

In this paper we have presented an aspect-oriented approach to software-based fault-tolerance, which can be tailored based on application knowledge and, thus, overhead-wise clearly outperforms related works from other authors. We regard the total performance overhead of 0.09–1.7 % as negligible. At the same time, the number of system failures (both crashes and silent data corruptions) caused by errors in eCos' scheduler and thread data structures could be reduced significantly from 12.8 % to below 0.01 %. Moreover, our approach is completely generic and can be applied to any other object-oriented C++ program as well.

An interesting side effect is that in contrast to hardware ECC solutions, software bugs are also detected. For example, if a parallel thread or an interrupt handler erroneously overwrites the content of an object, the proposed mechanism will detect the problem. Considering the ongoing trend towards multi- and many-core CPUs and multi-threaded code, this property might become a huge benefit.

The main disadvantage of the approach is the significant code bloat, caused by the vast number of instantiations of generic code. This is certainly a problem in some embedded usage scenarios. Code size reduction will therefore be our primary goal for future work. We expect that a better interprocedural analysis inside the AspectC++ compiler would help to further reduce the number of checks, but its implementation is subject to future work.

Acknowledgments

We wish to thank the anonymous reviewers for their very helpful and encouraging comments. This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-2, and the SFB 876 project A4.

References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 193–204.

[2] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs," in *ACM SIGOPS/EuroSys Int. Conf. on Computer Systems 2011 (EuroSys '11)*. New York, NY, USA: ACM, Apr. 2011, pp. 343–356.

[3] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *17th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '12)*. New York, NY, USA: ACM, 2012, pp. 111–122.

[4] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *International Conference on Dependable Systems and Networks, 2006. DSN 2006*, Jun. 2006, pp. 365–370.

[5] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits," in *International Conference on Dependable Systems and Networks, 2002. DSN 2002*, 2002, pp. 205–209.

[6] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[7] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[8] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Whitepaper*, 1997.

[9] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *15th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '10)*. New York, NY, USA: ACM, 2010, pp. 397–408.

[10] A. D. Fogle, D. Darling, R. C. B. II, and E. Daszko, "Flash memory under cosmic and alpha irradiation," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 3, pp. 371–376, Sep. 2004.

[11] A. Massa, *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *11th Eur. Conf. on OOP (ECOOP '97)*, ser. LNCS, M. Aksit and S. Matsuoka, Eds., vol. 1241. Springer, Jun. 1997, pp. 220–242.

[13] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *1st IEEE Int. W'shop on Source Code Analysis and Manipulation*, 2001.

[14] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic, "JVM susceptibility to memory errors," in *Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1 (JVM'01)*. Berkeley, CA, USA: USENIX Association, 2001.

[15] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: protecting critical data in unsafe languages," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 219–232.

[16] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Proc. of the 28th Int. Conf. on Computer Safety, Reliability, and Security*, ser. SAFECOMP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 283–296.

[17] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th Int. Conf. on Arch. of Comp. Sys. (ARCS '12), Workshop Proceedings*, ser. Lecture Notes in Informatics, vol. 200. German Society of Informatics, Mar. 2012, pp. 201–210.

[18] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 585–594, Nov. 1980.

[19] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *W'shop on Advanced SoC (OOPSLA '00)*, Oct. 2000.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *15th Eur. Conf. on OOP (ECOOP '01)*, ser. LNCS, vol. 2072. Springer, Jun. 2001, pp. 327–353.

[21] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, vol. 20, no. 7, pp. 636–651, 2007.

[22] D. Lohmann, G. Blaschke, and O. Spinczyk, "Generic advice: On the combination of AOP with generative programming in AspectC++," in *3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, ser. LNCS, G. Karsai and E. Visser, Eds., vol. 3286. Springer, Oct. 2004, pp. 55–74.

[23] K. Czarnecki and U. W. Eisenecker, *Generative Programming. Methods, Tools and Applications*. AW, May 2000.

[24] A. Alexander, *C++ Design: Generic Programming and Design Patterns Applied*, ser. C++ In-Depth. Adison-Wesley, 2001.

[25] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971.

[26] W. Kao, R. K. Iyer, and D. Tang, "FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1105–1118, Nov. 1993.

[27] J.-C. Fabre, F. Salles, M. Rodriguez-Moreno, and J. Arlat, "Assessment of COTS microkernels by fault injection," in *Proceedings of the conference on Dependable Computing for Critical Applications*, ser. DCCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 25–.

[28] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. A. Rennels, "Experimental evaluation of a COTS system for space application," in *Proc. of the 2002 Int. Conf. on Dependable Systems and Networks (DSN '02)*. Washington, DC, USA: IEEE Computer Society, 2002.

[29] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of commodity systems and software to memory soft errors," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1557–1568, Dec. 2004.

[30] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the 2010 USENIX annual technical conference*, ser. USENIX ATC'10. Berkeley, CA, USA: USENIX Association, 2010.

[31] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *39th Annual International Symposium on Computer Architecture (ISCA), 2012*, Jun. 2012, pp. 285–296.

[32] D. Dopson, "SoftECC: a system for software memory integrity checking," Master's thesis, Massachusetts Institute of Technology, Sep. 2005.

[33] H. Schirmeier, J. Neuhalfen, I. Korb, O. Spinczyk, and M. Engel, "RAMpage: Graceful degradation management for memory errors in commodity linux servers," in *17th IEEE Pacific Rim Int'l Symp. on Dep. Comp. (PRDC '11)*. Pasadena, CA, USA: IEEE, Dec. 2011, pp. 89–98.

[34] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, Jun. 2006, pp. 83–92.

[35] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in *Int. Conf. on Dependable Systems and Networks (DSN)*, 2000, pp. 71–78.

[36] M. Leeke and A. Jhumka, "An automated wrapper-based approach to the design of dependable software," in *The Fourth International Conference on Dependability (DEPEND)*. IARIA, 2011.

[37] F. Afonso, C. Silva, S. Montenegro, and A. Tavares, "Applying aspects to a real-time embedded operating system," in *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, ser. ACP4IS '07. New York, NY, USA: ACM, 2007.

[38] R. Alexandersson and J. Karlsson, "Fault injection-based assessment of aspect-oriented implementation of fault tolerance," in *International Conference on Dependable Systems Networks (DSN 2011)*, Jun. 2011, pp. 303–314.

[39] Y. Aumann and M. A. Bender, "Fault tolerant data structures," in *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, ser. FOCS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 580–589.

[40] K. Kant and A. Ravichandran, "Synthesizing robust data structures—an introduction," *IEEE Trans. on Computers*, vol. 39, no. 2, pp. 161–173, Feb. 1990.

[41] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. of the 18th annual ACM SIGPLAN Conf. on Object-oriented programing, systems, languages, and applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 78–95.