Exploiting Selective Placement for Low-cost Memory Protection

MOJTABA MEHRARA and TODD AUSTIN
Advanced Computer Architecture Lab, University of Michigan, Ann Arbor

14

Many embedded processing applications, such as those found in the automotive or medical field, require hardware designs that are at the same time low cost and reliable. Traditionally, reliable memory systems have been implemented using coded storage techniques, such as ECC. While these designs can effectively detect and correct memory faults such as transient errors and single-bit defects, their use bears a significant cost overhead. In this article, we propose a novel partial memory protection scheme that provides high-coverage fault protection for program code and data, but with much lower cost than traditional approaches. Our approach profiles program code and data usage to assess which program elements are most critical to maintaining program correctness. Critical code and variables are then placed into a limited protected storage resources. To ensure high coverage of program elements, our placement technique considers all program components simultaneously, including code, global variables, stack frames, and heap variables. The fault coverage of our approach is gauged using Monte Carlo fault-injection experiments, which confirm that our technique provides high levels of fault protection (99% coverage) with limited memory protection resources (36% protected area).

Categories and Subject Descriptors: B.8.1 [**Hardware**]: Performance and Reliability—*Reliability, testing and fault-tolerance*

General Terms: Design, Reliability, Experimentation

Additional Key Words and Phrases: Partial memory protection, selective placement, transient faults, fault-tolerant design, memory system design

ACM Reference Format:

Mehrara, M. and Austin, T. 2008. Exploiting selective placement for low-cost memory protection. ACM Trans. Architec. Code Optim. 5, 3, Article 14 (November 2008), 24 pages. DOI = 10.1145/1455650.1455653 http://doi.acm.org/10.1145/1455650.1455653

This article extends an earlier version that appeared in the 2006 ACM SIGPLAN Workshop on Memory Systems Performance and Correction (MSPC'06), October 2006.
Contact author's address: University of Michigan, Advanced Computer Architecture Lab, Ann Arbor, MI 48109; email: {mehrara, austin}@umich.edu.

1. INTRODUCTION

Providing protection from soft errors is a growing concern in nearly all computing markets, from high-end servers to embedded processors. This concern is underscored by scaling trends which reduce transistor sizes and supply voltages. These trends result in diminishingly small charge stored in transistor devices and correspondingly higher exposure to soft error events [Baumann 2002]. While it is the case that both memory and logic are affected by these trends, memory failures are still considered to be the dominant cause of soft error failures, because unlike logic, memory is continuously exposed to the effects of soft errors. (In contrast, logic is only exposed to soft errors when it is actively computing a result, a comparatively tiny fraction of time [Mukherjee et al. 2003].) As a result, several widely adopted techniques have been developed for memory protection [Bossen and Hsiao 1980; Cardarilli et al. 2003; Chen and Hsiao 1984; Saleh et al. 1990; Vargas and Nicolaidis 1994]. In general, these techniques impose moderate to high levels of power, area, and performance overhead on the system. For example, some cell-hardening techniques increase the memory access time by 6% to 8% and increase the area by 13% to 15% [Derhacobian et al. 2004]. Protecting memory by error correcting codes (ECC) is also expensive as the area overhead could be as much as 31% to 50% for 16-bit and 8-bit systems [Derhacobian et al. 2004]. Furthermore, generating the code bits and run-time checking of data and codes consumes extra energy and without careful design can increases cycle time. Unfortunately, the cost and power overheads often become problematic for cost-sensitive embedded system designs, forcing embedded designers to either sacrifice reliability or increase system cost.

Reliable memory can be provided for cost-sensitive embedded systems (and other cost-sensitive reliable system design applications) with a significant cost savings if only the critical fraction of the main memory is protected. The challenge in creating a partially protected memory design is (1) determining which variables are most critical to program correctness, (2) orchestrating placement of these variables into protected memory, and (3) implementing a memory system that efficiently provides partial memory protection.

To motivate the possibility of a partially protected memory system, we examined the variables of the *lex* benchmark. A closer look at this application's variables reveals that not all variables need the same level of protection. For example, we found two global variables with dramatically different vulnerability to soft errors. The first variable is "ZCH" which is used in the parser section to process the character set specifier. The second variable is "yychar" which is the input token number. Both occupy 4 bytes in memory. To assess the vulnerability level of these two variables, we injected a single fault to a random bit of each variable at a random cycle during the program execution in two separate experiments. We conducted each experiment 5,000 times. The results were quite promising. In the experiment with fault injection to "yychar" variable, only four runs out of 5,000 failed (i.e., produced incorrect output), whereas in the experiment for "ZCH," we noticed 4,753 failures. This result demonstrates that, compared to "yychar," "ZCH" is nearly 1,188 times more vulnerable to transient faults. This observation underscores our assertion that different variables in

a program may exhibit various vulnerabilities to transient faults. Therefore, in presence of tight constraints on area and power in embedded systems, it is possible to add protection to only a fraction of the memory without significantly sacrificing overall fault coverage, as long as the most critical program elements are identified and put into protected storage.

1.1 Contributions of This Article

In this article, we study the behavior of various embedded applications and evaluate the potential for application-based selective memory protection against soft errors. In considering partial memory protection, we examine the efficacy of selectively placing both program code and data. In particular, we make the following contributions:

- We *propose the first reliability-aware placement scheme for selective placement of code and variables in the main memory.* With this approach, we utilize a profile-based criticality analysis to determine the most vulnerable program components, which are then placed into a small protected storage region. Our technique employs a global placement strategy in which program code, global variables, stack storage, and heap variables all compete for limited protected storage resources.
- We *develop profile-based criticality metrics* based on the liveness of code and data, such that code and variables which are frequently used are placed first into limited protected storage resources.
- We *introduce a simulator evaluation framework that accurately emulates the placement functionality of the compiler, linker, and dynamic run-time system.* This allows us to carefully analyze the behavior of the program and explore the full potential of varied partial memory protection schemes without the undue burden of implementation details.
- We *develop an accelerated fault injection simulation technique that can reduce the overall analysis time by up to an order of magnitude*. In this framework we inject multiple faults in a single simulation pass and in case of failure we perform several single fault injection experiments using the same fault information as the multiple injection pass.
- We *describe an architecture for integrating partial protection support in traditional memory protection systems.* The extra hardware incurs minimal amount of area, power, and performance overhead. Furthermore, we propose a solution to hide the extra performance penalty.

The remainder of the article is organized as follows. In Section 2, we present an in-depth analysis of memory behavior in embedded applications to justify the applicability of our method in this area. In addition, we describe our reliability-aware placement technique. The proposed architectural support for partial memory protection is presented in Section 3. Section 4 details the evaluation methodology and simulation framework, and experimental results and discussions are presented in Section 5. Section 6 explores some related works in memory protection area, and finally, Section 7 concludes the article.

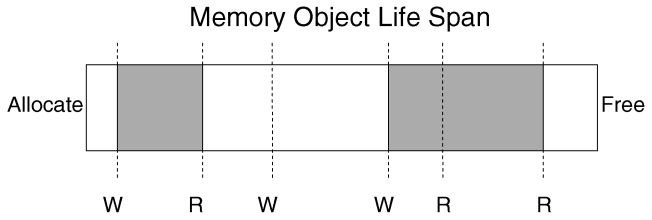

Fig. 1. Memory object lifespan. The object is *live* only during the shaded intervals.

2. SELECTIVE RELIABILITY-AWARE PLACEMENT SCHEME

In this section, we describe our selective profile-driven approach for placing memory objects in a partially protected memory system. Throughout the following sections, we use the term *object* to refer to various memory elements.

In general, a data object's life span during the program execution is partitioned into several phases, which are allocation-to-write, write-to-read, read-to-write, write-to-write, read-to-read, read-to-free, and write-to-free intervals. Traditional memory protection schemes consider memory elements to be vulnerable to transient faults during all these phases. Therefore, they tend to protect the entire memory in the system. However, soft errors can only lead to system failure if element's stored value is live at the time of fault strike. Among the above intervals, the data object value is only live during write-to-read and read-to-read. Furthermore, the code and constant objects' liveness interval would be only from the start of the program to the last read (if any). Figure 1 shows a data object's lifespan where liveness intervals are highlighted in shaded regions.

Considering these facts, we can identify memory object's level of vulnerability to transient faults after performing a liveness analysis for the whole memory system during the program execution. Subsequently, we would be able to protect only the objects with longer lifetimes.

Figure 2 shows the cumulative distribution function of lifetime versus object size for three sample benchmarks to depict various possible memory behaviors in actual programs. Clearly, many of the program variables found in these applications have short lifetimes, showing the potential for partial memory protection in a wide range of applications. Our technique is especially useful in programs that have a similar distribution to the *bitcount* benchmark. As can be seen, the CDF value for this benchmark starts from 82%, which means that most of the program's global variables were not used at all for the analyzed execution. In fact, our studies on several benchmarks from MiBench [Guthaus et al. 2001] and MediaBench [Lee et al. 1997] suites reveal the fact that a considerable amount of global storage in these benchmarks has very low or even zero lifetime (Figure 3). Therefore, there would be no need to protect the entire allocated memory during program execution.

Figure 4 illustrates the analysis flow for selective memory object placement. In this article, we have categorized the memory objects into global, heap, stack, and text segments. To achieve a better understanding of the effects of soft errors on each category, we have done a separate profiling and coverage analysis for

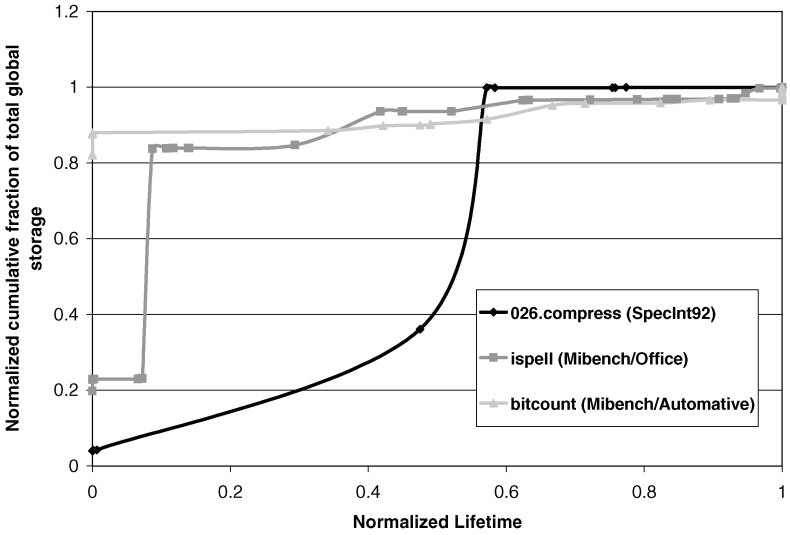

Fig. 2. *Cumulative distribution function* of average lifetime versus size for global variables in three benchmarks. The average slope of the curves over each section shows the concentration of storage at the corresponding life span (e.g., in *026.compress*, a large fraction of storage has a normalized lifetime value between 0.5 and 0.6). Objects' size and lifetime values are normalized to total global section size and total program duration respectively.

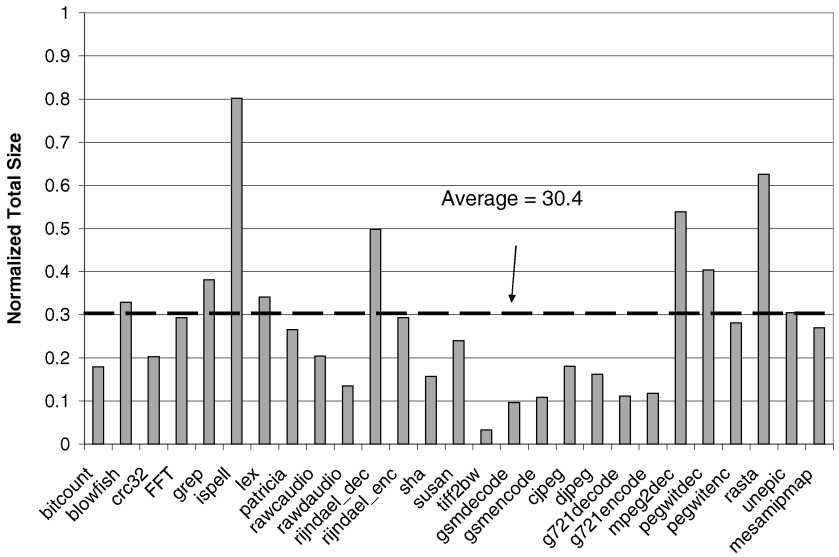

Fig. 3. Normalized total size of global variables that have a nonzero lifetime. On average, only 30.4% of the total global objects become live at least once during program execution.

each segment. Subsequently, we derived the total coverage using individual size and coverage data. Furthermore, a mixed analysis of global and text segments has also been done to give a more in-depth insight for sharing protected storage between these two types.

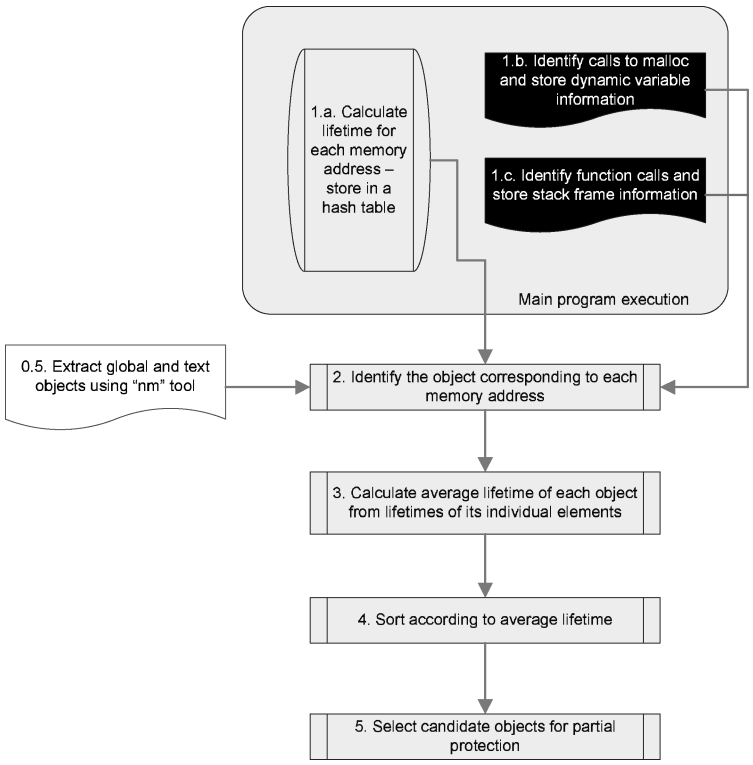

Fig. 4. Analysis flow for creating lifetime profile and performing data placement. Stage 0.5 is done outside simulator for global and text segments. Stage 1.b. is only executed for dynamic storage and stage 1.c. is only executed for stack segment.

2.1 Global and Text Segment Selective Placement

The compiler and linker assign the starting address and size of each structure in global and text segments, and they do not change with different inputs. Therefore, using the profile information, these objects can be reordered statically at compile time.

To extract the lifetime information for global variables and text objects, we initially add up individual liveness intervals for each memory address in these segments during the program execution. These intervals are the total cycles of write-to-read and read-to-read periods for global variables and the number of cycles from the start of the program to the last read access for the text objects. After matching addresses to their corresponding object (which might be a single element or an array), we compute the average lifetime of each object by summing up lifetimes of its individual elements and dividing the result by the number of elements that have had a nonzero lifetime. Subsequently, we sort them based on their average lifetimes as a metric for determining an object's exposure to transient faults. This technique has been applied to both individual global and text placement and a mixed analysis of the two types. Finally, considering the availability of protected storage and the exposure priority (which

is determined according to the ranking in the lifetime-based sorted list of all variables), we place the objects into either protected or nonprotected storage.

2.2 Heap Segment Selective Placement

Unlike global variables and text objects, the size and address of each heap variable is determined by dynamic memory management functions (e.g., malloc) throughout the program execution and may change with various inputs. Therefore, the profiling process for heap is more complicated and challenging than global and text segments. In fact, it is difficult and sometimes impossible to keep track of dynamic variables between the initial profile run and the actual program execution. Seidl and Zorn [1998], proposed several approaches for characterizing heap variables behavior in the profile run as a guide for predicting the behavior in the actual run of the program. Using that prediction, they tried to decrease the usage of virtual memory pages in programs by placing dynamic objects into different areas of the heap segment to improve the spatial locality of their references. These approaches are mainly based on the return address stack pointer value, the call path leading to the allocation function and the return address stack contents at the time of variable allocation. The following is an overview of these mechanisms and how they can be employed in this work for predicting heap variables' exposure priority:

- Stack Pointer. The value of the stack pointer at the time of function call to *malloc* (or other allocation functions) can be used as a reference to the allocated heap object. For instance, during the profile run, we can identify the required level of protection for a heap variable and store it along with the corresponding stack pointer in the profile table. Subsequently, during the actual run, we assign the same level of protection to all dynamic variables with the same stack pointer reference.
- Path Point. The basic motivation for this technique is the fairly high correlation between the behavior of the heap objects and their respective call sites. The implication of this approach in our context would be to initially assign the protection level of each dynamic variable to all call sites that precede the call to the corresponding dynamic allocation function. At the end of the program execution, gathered information is explored to find similar protection level patterns in each call site. If any protection level is dominant, we mark it as the call site's protection priority predictor. During the actual program run, we assign call site's protection priority to all variables that are dynamically allocated inside that site.
- Return Address Stack Contents. Utilizing the address stack contents at the time of allocation as the reference to dynamic variables is another approach for behavior prediction. When the dynamic allocation function is called, several return address stack entries which represent a subset of the call chain are combined to be used as the reference for the corresponding heap object. Seidl and Zorn [1997] have shown that using three or four stack entries is reasonably accurate for this purpose.

As Seidl and Zorn [1998] suggested, using the call chain information from RAS is more accurate than the other two techniques in assigning references to dynamic variables. A similar approach has also been used by Calder et al. [1998] for generating names for dynamic variables.

In this article, we use the third mechanism and assign references to dynamic variables by XOR-folding the last three stack entries. During the profile creation, we keep track of heap allocation by identifying calls to dynamic allocation functions (i.e., *malloc, calloc*, and *realloc*). After each call, we add the allocated object along with its reference to the list of heap variables, and using its size and starting address, we identify the corresponding memory transactions. Subsequently, we update each object's lifetime in the same way as we do for global variables. Finally, we mark heap objects as protected or nonprotected based on the overall liveness analysis and available protected storage. During the actual execution of the program, each heap object's reference is again computed by XOR-folding the last three stack entries. If it matches any previous entry in the profile information, we use the profile prediction for placing data in protected or nonprotected memory. Otherwise, the object is placed in the nonprotected area.

However, during profile creation, the computed reference for two or more different objects might become the same. In these cases, we conservatively use the prediction for the object with a longer lifetime (i.e., with higher protection priority).

2.3 Stack Segment Selective Placement

The dedicated size of the stack frame for various functions in a program is determined during compilation and subsequently the function's local variables can be accessed using the stack pointer during program execution. Our notion of local variable lifetime is slightly different from other types of the memory objects. We choose not to profile each local variable lifetime independently. Instead, we treat each stack frame as a single variable, and the lifetime of each frame would be from the time of function call to function return. Our results show that, by using this abstraction, we are still able to predict fault exposures with high accuracy.

Therefore, the profiling process for stack frames works as follows. Initially, during the profile run, we name each stack frame by the starting address of the corresponding function in the code segment. At the same time, we compute the lifetime of the frame by measuring the function call duration. If the function had been called before, the new lifetime would be added to the previous value. Otherwise, we insert a new entry into the profile table. Finally, our metric for transient fault exposure of the stack frame is calculated by adding up the lifetimes of all calls to the corresponding function and dividing the value by the total number of calls.

To employ selective stack placement, we propose using a noncontiguous stack [Iwamoto 2006], which is composed of protected and nonprotected sections. Using the profile data, the compiler would be able to assign the proper protection to the each stack frame in various functions based on the level of transient fault

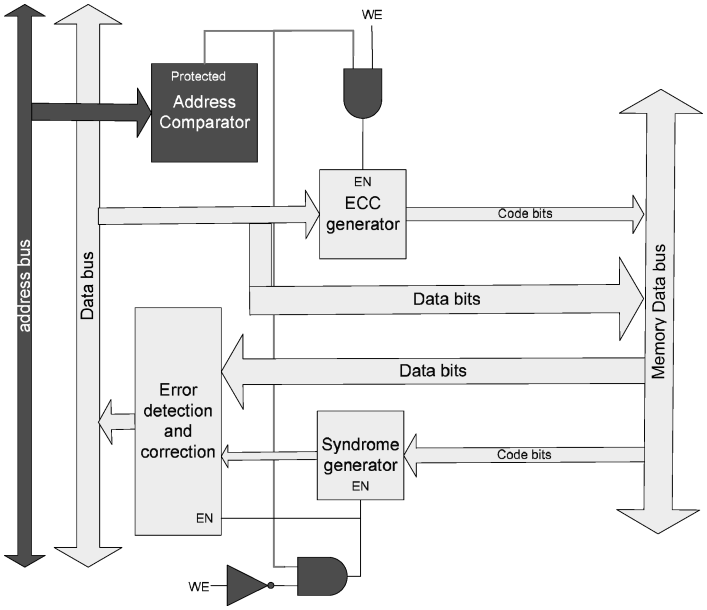

Fig. 5. Partial Memory Protection Architecture. The address comparator identifies the addresses that belong to the protected area of the memory and activates ECC generation and correction circuits accordingly.

exposure. A noncontiguous stack has a straightforward implementation if the compiler enforces allocation of the stack frame pointer (often eliminated during optimization). The stack frame pointer indicates the location of the previous stack frame and as a result, there is no need for individual stack frames to be stored contiguously in memory.

3. ARCHITECTURAL SUPPORT FOR PARTIAL MEMORY PROTECTION

Our placement approach can be used with both cell-hardening [Derhacobian et al. 2004; Vargas and Nicolaidis 1994] and ECC protection [Chen and Hsiao 1984; Derhacobian et al. 2004] techniques. In cases of cell-hardening, there is no need for any extra hardware to maintain partial protection. The only obvious requirement is notifying the compiler about the percentage and address ranges of the protected parts of the memory. However, to add partial ECC protection support, we need to make some changes to the traditional hardware.

In a traditional ECC memory system, ECC bits are computed as parities of different subsets of data bits such that each data bit contributes to the computation of more than a single ECC bit. Therefore, by keeping track of all the data bits that contribute to each ECC bit, single error can be detected, and its location can also be identified. In fact, these codes are usually designed so that single-bit errors can be corrected and double-bit errors can be detected without correction (SEC-DED ECC).

At the time of writes into ECC-protected memory, the ECC bits are computed by a set of XOR trees (the ECC generator in Figure 5). When the word is read

back, the XOR trees recompute the ECC using the data from memory and compare it to the ECC bits read from the memory. Any discrepancy indicates an error. By checking the discrepancies in the ECC bits, the error in data or ECC bits can be identified. These checks are performed by an XOR of the recomputed ECC bits and the ones read from memory. The result of these checks is called the syndrome (output of the syndrome generator in Figure 5). A syndrome of zero shows that no error has occurred. However, a nonzero syndrome indicates an error and can be used in the error detection and correction block to determine which bits are in error, or suggest that the error is uncorrectable.

Figure 5 shows our proposed architecture for a partially protected memory system. The shaded parts highlight the additional components compared to the traditional hardware. In this architecture, the address comparator checks whether the incoming read or write addresses belong to the protected section of the memory or not. In that case, the *protected* signal goes high and enables the ECC generator for write operations and syndrome generator and correction circuit for read operations. To make the partially protected memory hardware less complicated, we propose having protected and unprotected space as two distinctive chunks in the memory. In this way, the address comparator just needs to check the incoming address with one value. If it is more than that value, that memory address is protected. Otherwise, it is unprotected. This address is loaded from the memory modules at the system start-up and the address comparator just needs a single register to keep it. In this way, the storage costs in the address comparator would be minimal. Furthermore, the comparator adds just one cycle to the memory access time (which is typically tens to hundreds of cycles) for the compare operation. This single register is also protected using ECC.

Using this architecture, protection support circuits are kept inactive when they are not needed and as a result, their power consumption is minimized. If the target processor has a mechanism for checkpointing and recovery, we can put the syndrome generation and error correction blocks out of the main data path [Dupont et al. 2002]. Therefore, the only performance penalty would be the extra roll back and recovery period in case of a failure compared to the constant clock cycle degradation due to the error detection and correction circuits in conventional systems.

4. EVALUATION METHODOLOGY

In this section, we introduce our novel accelerated fault injection mechanism. The main pitfall of previous evaluation mechanism in Mehrara and Austin [2006] that we strive to overcome is the long duration of simulation time for fault injection experiments. Furthermore, since we noticed a fairly high derating factor for most of the benchmarks in Mehrara and Austin [2006], we had to run a large number of experiments to see actual failures and reach statistical confidence for the results. On the other hand, we could not use techniques similar to architectural vulnerability factor analysis (AVF) [Mukherjee et al. 2003] to speed up our evaluation because, according to the AVF definition, these factors for memory elements should be computed using variable lifetime values.

Since these values are the key factor in our selective placement process, we would have got potentially misleading results if we had evaluated our technique by measuring the same parameter that we are directly trying to minimize (i.e., variable lifetime).

Therefore, we designed an accelerated fault injection system in which we inject multiple random faults to a single instance of program execution. Subsequently, if we notice a failure, we rerun the experiment multiple times, and during each run, we inject one fault out of the same previous multiple fault set. As a result, we can readily determine which fault from the multiple fault set originally caused the failure.

There is a possibility that the program fails in the execution pass with multiple fault injections and does not fail with any of the single fault injection runs. In this case, we conclude that two or more faults caused the problem, and since a single fault model is employed for modeling transient faults here, we can ignore the failure in this particular fault scenario. Another possibility is that multiple injected faults mask each other, and while no failure is noticed in the multiple fault injection, each individual transient error could have caused a failure. However, in our framework, we inject only 10 faults per execution, and they are uniformly distributed in both execution time and memory location. Therefore, due to the relatively long execution time and large memory usage in the benchmarks, we assume that the probability of this masking effect is extremely low and negligible. The fact that our results using accelerated framework for global and dynamic sections follows the results in Mehrara and Austin [2006], shows that this assumption is valid.

The accelerated fault injection approach is particularly beneficial in benchmarks with high derating factors.[1] Since a high derating factor corresponds to less vulnerability to transient faults, using the accelerated fault injection framework, we can reach statistical confidence with a much less number of experiments. Figure 6 shows our profiling and accelerated fault injection framework.

As stated previously, a separate profiling analysis has been done for different memory segments. For the global and code segments, we extract the size and address of each object after linkage. Then, we create a run-time profile of object lifetime behavior in the program and feed back the protection information to the fault-exposed instance of the simulation. For heap storage, the address, size, and lifetime information are gathered dynamically during the profile run. Finally, for the stack, we extract the lifetime of each stack frame for different functions during the profile execution. To achieve statistical confidence for the results, we run each accelerated fault injection simulation in a Monte Carlo framework for 600 times. This will add up to a total of 6,000 injected faults per experiment. Since the number of failures in stack fault injection experiments were quite high in most of the benchmark, we decided to use the single fault injection framework for stack analysis. For the same reason (low derating factor), less number of fault injections were required to achieve statistical confidence. Therefore, we injected 2,000 faults per benchmark in stack analysis experiments.

[1]Derating factor metric is explained in more depth in Section 5.1.

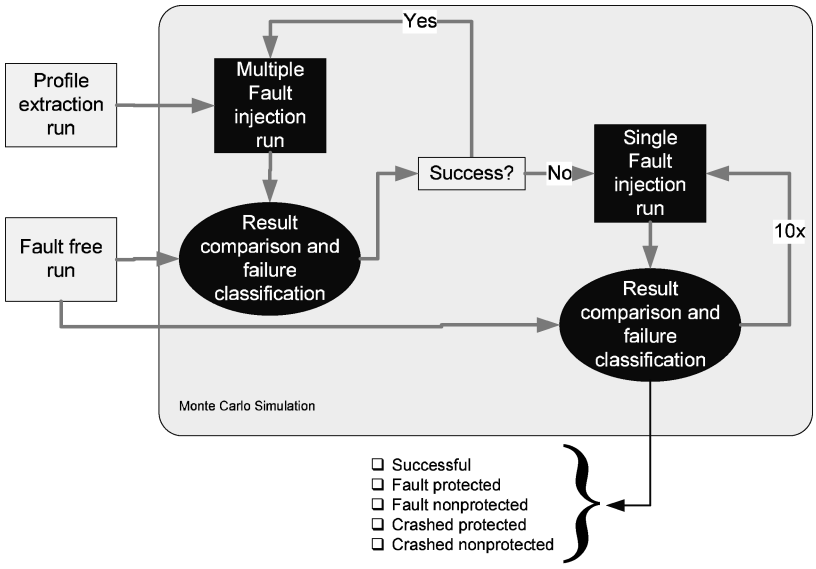

Fig. 6. Accelerated Fault Injection framework. Profile creation and fault free runs are executed once. In each multiple fault injection run, arbitrary number of faults (10 in this work) are injected into the program memory. If the program fails, the faults are injected again one by one. Single fault injection passes are classified into one of the five categories based on the outcome of result comparison phase.

We have instrumented the *sim-safe* simulator in the SimpleScalar toolset [Austin et al. 2002] to perform our experiments. *Sim-safe* is a functional simulator in the toolset, and while providing an acceptable level of accuracy, it is fast enough to make the Monte Carlo experiments feasible in terms of total simulation time. Since we are injecting fault to the memory space rather than the underlying processor microarchitectural elements, using a high-level architectural simulation gives a good approximation of the actual effects of soft error propagation and effects of memory-level transient faults in the final program outcome [Wang and Patel 2005].

Using fully protected caches is the only architectural improvement that may affect our results. We have not considered this effect in this work, since it only serves to reduce the protection requirements in the memory. It should be noted that a fully protected cache can hardly provide high levels of protection when there is no protection in the main memory. Because we noticed that there are many critical memory elements that are not frequently accessed and would incur cache misses most of the time. Therefore, if the main memory is not protected at all, an already corrupted value may be fetched from the memory and negatively affect reliability. Finally, if a fully protected cache is used in the target system, our profiling analysis should be changed to accommodate the cache effects by adding a cache simulator to the profiler and refine the lifetime values according to cache hits and misses.

The compiler, linker, and dynamic allocation function (e.g., malloc) needed to be changed to implement the proposed placement approach using profile

information in a real system. However, the purpose of this article is to explore the potential prospect for partial protection of the memory. Therefore, we decided to perform our evaluations without changing address assignment for memory objects and emulate the placement technique in the simulator. To accomplish this for global and text storage, we mark each variable during the profiling pass as protected or nonprotected based on the relative vulnerability level and the available protected memory. For heap variables and stack frames, since we do not know the allocated space *a priori*, the process of marking objects as protected or nonprotected is accomplished throughout the fault injection experiment, using the profile information as the placement prediction. To make the best use of available protected memory, if a protection prediction could not be found in the profile data (because the dynamic variable had never been allocated or the function corresponding to a specific stack frame had never been called in the profile run), our placement approach places the variable in the nonprotected section of the memory. Due to this fact, the prediction quality of our approach directly depends on how accurate the profile input represent typical inputs of the given application.

For the rest of the evaluation process, we use the following approach in all segments. During the fault injection experiments, we pick a random memory location and identify the corresponding object to which it belongs and the protection prediction of that object according to the profile data. Subsequently, we flip a single bit in that location in a random cycle during program execution. The fault injection methodology is based on the assumptions that faults are uniformly distributed in both time and memory space [Saggese et al. 2005]. Finally, we compare the outputs with the results of a fault-free instance of the same program. Comparison for exact similarity is pessimistic for some benchmarks such as audio or video applications. Because a slight inconsistency in most parts of the output would not be even noticed by the user [Li and Yeung 2007]. However, to generalize the method for all types of programs, we conservatively consider the worst-case scenario where the execution is sensitive to any slight output discrepancy. Due to this worst-case analysis, we anticipate that the actual protection requirements of many applications are even less than what we report here. One could argue that if we make the error analysis in these applications more realistic, the amount of required protected memory could change a lot. Because many protected areas in the current configuration may not lead to true errors. However, that will not happen even in a realistic analysis due to the high number of fault injections. Because in the current experiments, the number of injections is so high that most failure cases are covered and a significant change in the required protected memory is quite unlikely.

After the comparison, if the results match, we mark the simulation pass as *successful*. Otherwise, based on the protection level of the faulty address, we mark the experiment as *fault-protected* or *fault-nonprotected*. Fault-protected means that if we had applied the selective placement strategy, the simulation would not have failed. With the same analogy, fault-nonprotected means that even if we had used this technique, the result would have been different compared to the fault-free run.

There is a possibility that the program crashes after fault injection. We address this issue by monitoring the simulation duration and marking the simulation pass as crashed (which might be protected or nonprotected) if its execution lasts four times longer than the program's typical simulation time.

Since most parts of the reserved memory space in the simulator are not allocated to anything, we avoid injecting faults to the whole memory space. Instead, for global variables, text segments, and mixed global/text experiments, we only inject faults to the memory locations that were allocated in each benchmark after linkage. For heap and stack experiments, we inject faults to the area allocated until the randomly picked fault injection cycle for each run. In this way, due to the higher number of failed instances, we would need fewer simulations to reach statistical confidence.

5. EXPERIMENTAL RESULTS

We perform two sets of experiments for each type of memory object in each benchmark. In one set, the same input is used for profile creation and fault injection to get a better understanding of the benchmark's characteristics. The results of this set show the optimal efficiency of the placement approach and our lifetime-based criticality metric. In the second set, the profile is created with the first input and the selective placement and fault injection are performed on the benchmark with the second input. The latter set reveals the actual potential for protecting memories partially. We use 12 benchmarks from MiBench benchmark suite [Guthaus et al. 2001] as representatives of real-world embedded applications. In addition, we add *026.compress* benchmark from SPECint92 and *grep*, which is a general utility, to our benchmark set. Since we need to run each fault injection experiment for many times, simulation time is one of the most important factors in selecting the target benchmarks for evaluation.

5.1 Protection Ratio vs. Fault Coverage

To assess the relative intrinsic exposure level in various segments of a given benchmark, we refine the definition of derating factor [Constantinides et al. 2005] by introducing a new metric, the normalized memory derating factor (NMDF). The original derating factor was defined to be the inverse of total error rate, representing the application's resilience against transient faults in memory. For instance, a derating factor of 10 shows that one out of every 10 transient faults in the memory would lead to system failure. However, since different segments have different sizes in the program and the the probability of transient fault strike is directly proportional to the target area, the original derating factor definition does not give a fair estimate of the relative intrinsic resiliency of various segments. Therefore, we define the new NMDF metric to be the inverse of total error rate divided by the total segment size for global data and text segments and average allocated size for heap and stack segments. This factor gives a measure of actual vulnerability of each segment regardless of its size and represents the possibility of a fault in any single unprotected byte leading to program failure. The main reason that we use average active size for heap and stack is related to our fault injection method. As mentioned

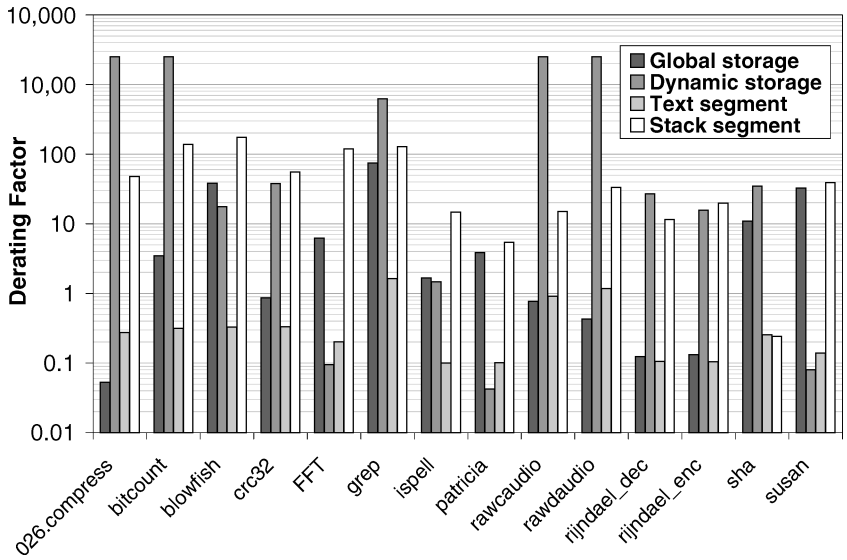

Fig. 7. Normalized memory derating factor of text, global, heap and stack segments for each benchmark. This factor is the inverse of error rate divided by average segment size and shows the benchmark's intrinsic resilience to soft errors in memory. Lower factors indicate more exposure to transient faults. The Y axis is in logarithmic scale.

in the previous section, for these two segments, the target memory location for fault injection is randomly selected from all allocated memory locations until the fault injection cycle, which is, in general, smaller than the total segment size. Therefore, if we had used total heap and stack segment size in our metric, we would have overestimated the effect of soft errors on these segments.

Figure 7 shows NMDF values for the four segments of the memory in each benchmark. According to this analysis, various memory segments show quite different behaviors based on the benchmark characteristics. However, the text segment is, in general, the most vulnerable segment in memory, because even a single change in the code can lead to inexecutable applications or wrong path of execution. In addition, the values in text segment, unlike data segment, are constants and the corrupted location has absolutely no chance of getting overwritten. Among the four segments of the memory, the heap segment generally has the least exposure to transient faults. One thing to note here is that in some of the benchmarks (*026.compress, bitcount, grep, rawcaudio, and rawdaudio*) we did not observe any heap storage vulnerability to transient faults during our fault injection experiments. This is due to the heap storage usage pattern in these programs and the fact that most of their allocated dynamic variables are not critical in determining the result or they are live only for a very short interval compared to the total program duration. Having no failures would lead to a NMDF of infinity, which translates to no vulnerability to soft errors in heap storage for the corresponding benchmarks. To avoid underestimating the overall effect of soft errors in heap and keep the results statistically sound, we assume a single failure for these benchmarks.

Figure 8 shows the minimum amount of necessary protection for coverage of more than 95% and 99% for each benchmark. These results imply that, on average, by protecting 28.6% and 30.3% of the text segment, 25.7% and 31.4% of the global data, 22.1% and 23.9% of the dynamic storage, and 28.7% and 34.3% of the stack, we can achieve transient fault coverage of more than 95% and 99%, respectively, in these areas.

As stated previously, two sets of experiments are performed for each type of memory object in each benchmark. We call the first set P1F2, which means that we use the first input for profiling and the second input for fault injection experiments. Likewise, P2F2 means that second input is used for both profiling and fault injection. It is interesting to note that in most of the benchmarks, the predicted protection levels are quite close for for P1F2 and P2F2 experiments, which shows the high accuracy of the profile-based analysis in general and the return-address-stack-based naming convention for heap variables in particular.

Figure 9 shows the required memory in the combined analysis of global data and text segments for achieving more than 99% coverage. This figure also shows the relative amount of protected storage of these segments for a given protected memory. According to this analysis (and the 95%+ analysis, which is not shown here), nearly 32.5% and 35% of the combined global and text segments should be protected to achieve more than 95% and 99% coverage in these areas.

5.2 Storage Requirements

Figure 10 depicts the relative sizes of various segments in each benchmark. Using these values and previous data about the coverage in individual types of variables, the total percentage of necessary protected memory is derived and is shown in Figure 11. According to these results, the total required amount of protected memory to achieve 95% and 99% transient fault coverage is 32.6% and 35.8%, respectively. As a result, by using this technique, the area overhead of memory protection can be lowered to nearly one third of typical protection mechanisms to achieve nearly complete coverage. Furthermore, protection requirements of all applications except *026.compress*[2]—which is not an embedded application—is between 10% and 60% and shows a moderate deviation around the mean. Since most embedded platforms run a limited number of applications, the final amount of protected storage in any particular embedded system can be tuned by the manufacturer—or ordered by the user—based on the requirements of typical target applications.

Since our scheme tends to protect the most critical variables with high access counts, we do not expect it to make a notable contribution to the dynamic power reduction. However, due to the considerable area savings, the leakage power, which tends to increase by shrinking feature sizes, would be reduced significantly.

The final results in Figure 11 show great promise for partial protection techniques. It is interesting to note that the code section has been the dominant

[2]*026.compress* from SPECint92 has been added to the benchmark suite to describe an unusual case in the global data segment and does not represent a normal behavior of embedded applications.

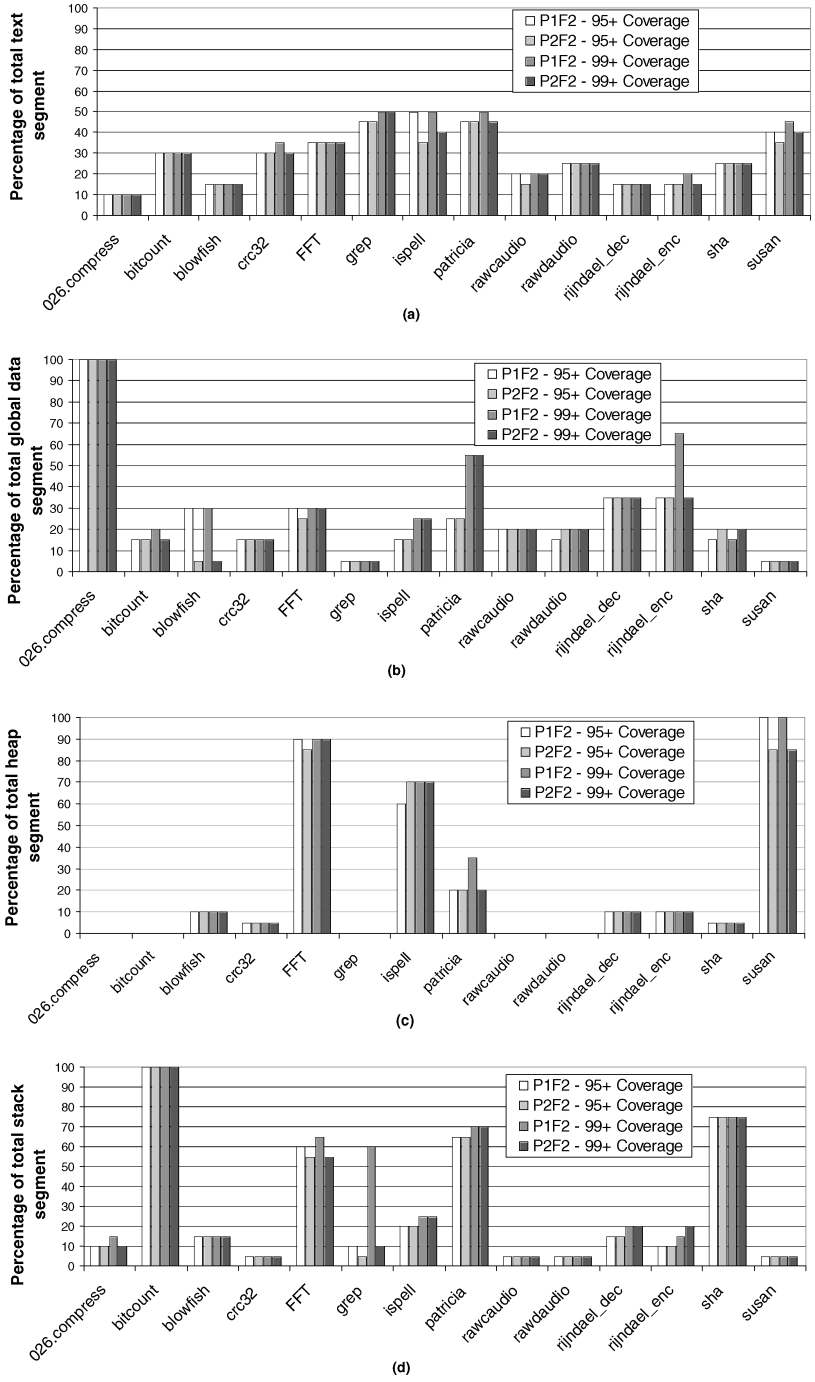

Fig. 8. Minimum amount of protected memory to satisfy more than 95% and 99% transient fault coverage in (a) text segment, (b) global data segment, (c) heap segment, and (d) stack segment. PxFy indicates that the profile is created with input x and the data placement and fault injection experiments are done with input y.

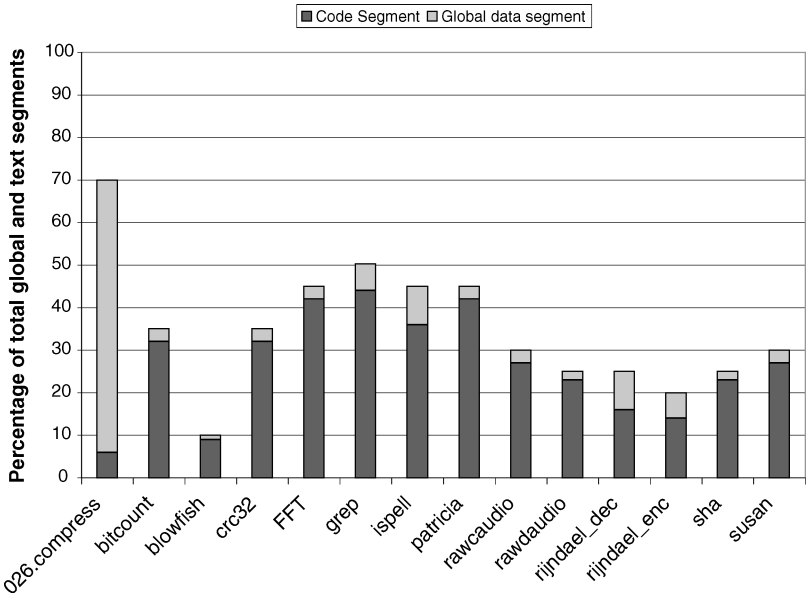

Fig. 9. Combined analysis of global data and text segment. This figure shows required amount of memory for achieving a coverage of more than 99% in P1F2 experiments with mixed global/text analysis.

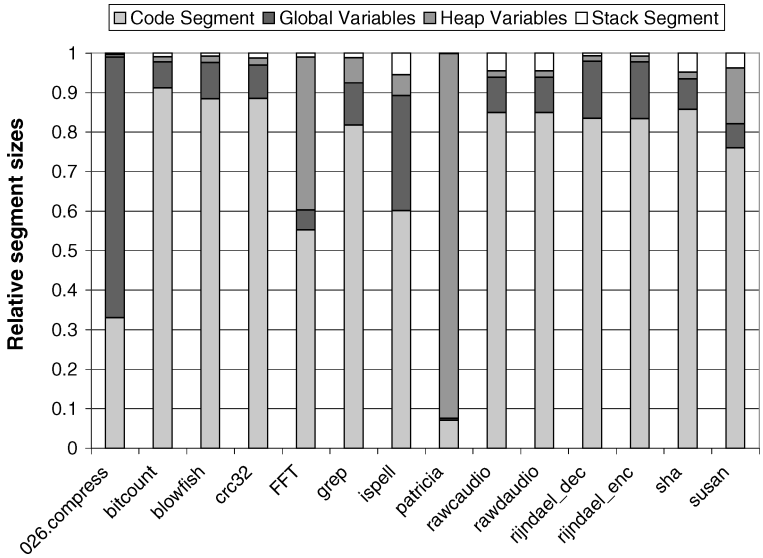

Fig. 10. Relative storage usage. Each segment size is normalized to the total amount of memory

contributor to the protected storage demand. Since the code size in these programs is comparatively larger than the data size, partially protecting this amount increases the total absolute protected storage to a greater extent. Another important fact is that the code is intrinsically more vulnerable to soft errors than data. Data has the opportunity to be corrupted without affecting

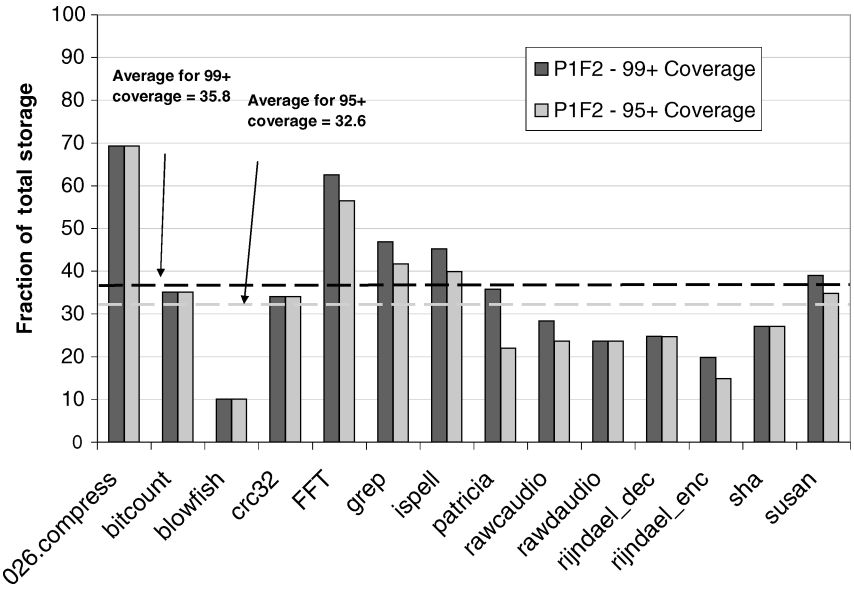

Fig. 11. Overall required protected memory to achieve the coverage of more than 95% and 99%. The average values are 35.8% for 99+% and 32.6% for 95+% coverage.

program correctness. For example, a data value can be corrupted and still be not equal to zero. Furthermore, the corrupted data value might get overwritten without being ever read. The code, on the other hand, is much more fragile. If a bit is flipped in an instruction, it becomes another instruction altogether, unless the flipped bit is unused.

As shown in Figure 11, *026.compress* and *FFT* require relatively high amounts of protected memory. In *026.compress*, more than 96% of global storage is live on average for nearly half of the program duration. Therefore, there is no good way of getting acceptable coverage without protecting nearly all global storage area. Also, global storage accounts for 65% of total memory space in this benchmark. Therefore, we have to protect 70% of the storage to achieve high coverage values.

FFT, on the other hand, has four dynamic array structures for storing the real and imaginary parts of the input and output vectors. These arrays occupy more than 90% of the heap storage, and since heap is nearly 40% of total storage in this benchmark, the overall required protected storage goes up to around 60%.

However, in most of the other benchmarks, since the code segment is the dominant part in terms of storage capacity and it does not have much deviation in protection requirements, adding it to the overall analysis caused the overall deviation of the results between various benchmarks to become much less than the analysis for data segments in [Mehrara and Austin 2006].

6. RELATED WORKS

Using error-correcting codes (ECC) is perhaps one of the most popular memory protection techniques [Chen and Hsiao 1984]. However, traditional ECC

architectures impose a significant amount of overhead in terms of performance, area and power. Several works have been done to address this issue. Dupont et al. [2002] propose a method for concurrent ECC error detection to address the performance issue. They also utilize some low-cost codes to deal with the area and power penalties. Another technique is presented by Ghosh et al. [2004] to reduce power in memory ECC checkers. They use simulated annealing and genetic algorithm for selecting the parity check matrix in order to minimize the switching activity of the checker. This would lead to a moderate reduction in power, while incurring minimal extra overhead on delay and area.

In addition to ECC protection schemes, some works have focused on designing fault tolerant memory cells. Vargas and Nicolaidis [1994] present a fault-tolerant SRAM design that has a built-in current sensor circuit. This circuit detects the current spikes resulting from SEUs in memory power lines and the correction is performed with the help of a single parity bit. Using Deep N-well technology and increasing the capacitance of the critical nodes are two cell-hardening approaches that have been explored and compared to ECC in Derhacobian et al. [2004]. The Deep N-well collects a fraction of charges resulting from particle strikes and reduces the amount of charge that reaches critical nodes. Adding extra capacitance to cell layout increases the amount of critical charge needed to flip a node and thereby makes the node more resilient to particle strikes. The authors showed that these two techniques reduce the failure rate by a factor of 2 and 23 respectively. However, both of them increase the manufacturing costs, and the latter incurs a significant performance penalty as well.

Saleh et al. [1990] introduce a scrubbing technique to increase reliability in noisy environments and large memory systems. Scrubbing is performed by reading the memory words and their parities, checking for errors and writing back the correct data in case of any transient faults. The scrubbing interval can be either probabilistic or deterministic. The probabilistic method checks each memory location only when it is accessed. However, the deterministic approach cycles through the whole memory and checks for correctness in all memory locations. They show that scrubbing improves mean time to failure (MTTF) by an order of 10^8 compared to an unprotected memory. In addition, the MTTF for deterministic scrubbing is more than that of a probabilistic scrubbing by a factor of two.

In an early study [Mehrara and Austin 2006], we proposed the idea of reliability-aware data placement for partially protecting dynamic and global storage. However, we did not take into account the vulnerability of text and stack segment segments. In the present work, we develop a complete scheme for all parts of the memory, and we show that partial protection is a promising approach for text and stack segments as well as heap and global data storage. It is interesting to note that according to our result, the text segment is main factor in determining the amount of required protection in most embedded applications. The idea of protecting a fraction of the total memory and selective compiler-directed data protection was introduced by Chen et al. [2005]. However, in their work, the programmer is considered to be responsible for selecting appropriate variables for protection. Subsequently, the source code is modified

to fit the programmer's decision. One of the problems with this approach is that the programmer might not know a priori which variables are more exposed to faults. Furthermore, our analysis shows that some of the most critical variables in the applications belong to the linked libraries rather than the main program, and the programmer can hardly make changes or identify the level of fault exposure for those variables. In addition, the selective protection of the arrays has been done in *a finer* granularity than the complete array structure. Therefore, some parts of the code have to be duplicated to perform the same operation on both protected and nonprotected parts of the array. This may dramatically increase the code size which is generally considered to be a critical bottleneck in embedded systems. Finally, no analysis has been done to examine the effectiveness of the approach.

A clever idea for low-cost partial protection was proposed by Yan and Zhang [2005] for protecting the register file. The authors define a metric called RVF (register vulnerability factor) that is the probability that a soft error in the register file propagates to other system elements. They introduce two compiler-oriented techniques to reduce this factor. The first one is rescheduling instructions to decrease the registers' liveness window, which leads to less susceptibility to soft errors. The second scheme is reliability-oriented register assignment during compilation that selects the most vulnerable registers based on their RVF value and puts them in the ECC protected parts of the register files. They have shown that by protecting 16 registers out of 64, the average RVF value can be lowered from 14% to 3%. Nevertheless, they have not done any experiments to clarify the relation between their RVF metric and actual error coverage or mean time to failure of the register file.

Lee et al. [2006], Kim and Somani [1999], Zhang et al. [2003], and Zhang [2005] have focused on partial cache protection techniques. Kim and Somani [1999] employ parity caching and selective checking for low-cost cache protection. In parity caching, they store the check codes only for most recently used cache lines. Also, in selective checking, they compute and store check codes only for some blocks in a set (e.g., they store check codes for one way in a 4-way set associative cache). In Lee et al. [2006], all multimedia-related data in multimedia applications is considered noncritical and is mapped to the nonprotected cache blocks. The rest of data is assumed to be critical and is mapped to the protected cache blocks. Zhang [2005] has proposed a small replication cache to improve the reliability of data cache against transient faults. These approaches can be used in conjunction with our methodology as well.

Several works have focused on lowering costs in redundant execution schemes [Parashar et al. 2004; Gomaa and Vijaykumar 2005; Parashar et al. 2006; Reddy et al. 2006; Walcott et al. 2007; Reddy and Rotenberg 2007]. These works assume complete protection in the main memory, and they are orthogonal to our techniques in this article. They can be used with partial memory protection to provide a low-cost solution for designing a reliable system.

7. CONCLUSIONS

Recent trends of aggressive reduction in transistor feature sizes and supply voltages, have made transient fault protection a critically important issue in

embedded memories. However, not all embedded systems can tolerate the significant amount of area and power overhead imposed by traditional memory protection mechanisms. In such systems, the designer might be able to sacrifice a slight portion of transient fault tolerance to save power and area. In this article, we presented the first profile-driven placement technique that focuses on reliability and cost rather than performance. We identify the most critical memory objects and place them in the protected area of a partially protected memory. As a result, we would be able to avoid wasting power and area for protecting unimportant variables.

We also introduce an evaluation framework to explore various heuristics for exposure analysis of program variables and their resulting transient fault coverage, without changing the actual address assignments in the compiler, linker, and dynamic allocation function. After placing the variables according to the profile, we ran the benchmarks in an accelerated Monte Carlo simulation framework for 600 runs, each time flipping 10 bits in random places inside the allocated memory space and at a random cycle. In case of failure, we reran the experiment with the same fault set for 10 single fault injection experiments. Subsequently, we monitored protected and nonprotected program failures and crashes of single failures and using these data, we calculated the final coverage of the partial protection scheme for each benchmark. We have shown that, on average, by just protecting 32.6% of the total memory storage we can achieve more than 95% transient fault coverage in the memory for embedded applications. We can increase this amount to more than 99% by adding protection to another 3.2% of the memory. The actual amount of protected storage in a target-embedded system can be tuned by the manufacturer based on a similar analysis on the requirements of expected target applications.

Our results in this work show that except for some special cases, partial protection along with selective data placement proves to be an effective way of providing embedded memories with low-cost transient fault tolerance.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for providing useful comments and feedback on this article. This work was supported by grants from the Gigascale Systems Research Center.

REFERENCES

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: An infrastructure for computer system modeling. *Computer 35*, 2, 59–67.

BAUMANN, R. 2002. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *Proceedings of the International Digest of Electron Devices Meeting*. 329–332.

BOSSEN, D. C. AND HSIAO, M. Y. 1980. A system solution to the memory soft error problem. *IBM Journal of Research and Development 24*, 3, 390–397.

CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 139–149.

CARDARILLI, G. C., LEANDRI, A., MARINUCCI, P., OTTAVI, M., PONTARELLI, S., RE, M., AND SALSANO, A. 2003. Design of fault tolerant solid state mass memory. *IEEE Transactions on Reliability 52*, 4, 476–491.

CHEN, C. L. AND HSIAO, M. Y. 1984. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development 28*, 2, 124–134.

CHEN, G., KANDEMIR, M., IRWIN, M. J., AND MEMIK, G. 2005. Compiler-directed selective data protection against soft errors. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. 713–716.

CONSTANTINIDES, K., PLAZA, S., BLOME, J., ZHANG, B., BERTACCO, V., MAHLKE, S., AUSTIN, T., AND ORSHANSKY, M. 2005. Assessing SEU vulnerability via circuit-level timing analysis. In *Proceedings of the 1st Workshop on Architectural Reliability* (*WAR*-1).

DERHACOBIAN, N., VARDANIAN, V. A., AND ZORIAN, Y. 2004. Embedded memory reliability: the SER challenge. In *Proceedings of the International Workshop on Memory Technology, Design and Testing*. 104–110.

DUPONT, E., NICOLAIDIS, M., AND ROHR, P. 2002. Embedded robustness IPs for transient-error-free ics. *IEEE Design & Test of Computers 19*, 3, 54–68.

GHOSH, S., TOUBA, N. A., AND BASU, S. 2004. Reducing power consumption in memory ECC checkers. In *Proceedings of the International Test Conference*, 1322–1331.

GOMAA, M. A. AND VIJAYKUMAR, T. N. 2005. Opportunistic transient-fault detection. In *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*. IEEE Computer Society, 172–183.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization (WWC-4)*, 3–14.

IWAMOTO, T. 2006. Methods and apparatus for segmented stack management in a processor system. US Patent 20060195824.

KIM, S. AND SOMANI, A. K. 1999. Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, 246–255.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30)*, 330–335.

LEE, K., SHRIVASTAVA, A., ISSENIN, I., DUTT, N., AND VENKATASUBRAMANIAN, N. 2006. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*, 411–420.

LI, X. AND YEUNG, D. 2007. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*.

MEHRARA, M. AND AUSTIN, T. 2006. Reliability-aware data placement for partial memory protection in embedded processors. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC'06)*. 11–18.

MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. 29–40.

PARASHAR, A., GURUMURTHI, S., AND SIVASUBRAMANIAM, A. 2004. A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy. *SIGARCH Computer Architecture News 32*, 2, 376.

PARASHAR, A., SIVASUBRAMANIAM, A., AND GURUMURTHI, S. 2006. SlicK: slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. 95–105.

REDDY, V. AND ROTENBERG, E. 2007. Inherent time redundancy (ITR): Using program repetition for low-overhead fault tolerance. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks*. 307–316.

REDDY, V. K., ROTENBERG, E., AND PARTHASARATHY, S. 2006. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *Proceedings of the 12th*

International Conference on Architectural Support for Programming Languages and Operating Systems. 83–94.

SAGGESE, G. P., WANG, N. J., KALBARCZYK, Z. T., PATEL, S. J., AND IYER, R. K. 2005. An experimental study of soft errors in microprocessors. *IEEE Micro 25*, 6, 30–39.

SALEH, A. M., SERRANO, J. J., AND PATEL, J. H. 1990. Design of fault tolerant solid state mass memory. *IEEE Transactions on Reliability 39*, 1, 114–122.

SEIDL, M. L. AND ZORN, B. G. 1997. Predicting references to dynamically allocated object. (Tech. Rep. CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, Co. January 1997).

SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 12–23.

VARGAS, F. AND NICOLAIDIS, M. 1994. SEU-tolerant SRAM design based on current monitoring. In *Proceedings of 24th International Symposium on Fault-Tolerant Computing*.

WALCOTT, K. R., HUMPHREYS, G., AND GURUMURTHI, S. 2007. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, 516–527.

WANG, N. J. AND PATEL, S. J. 2005. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. 30–39.

YAN, J. AND ZHANG, W. 2005. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM International Conference on Embedded Software*. 203–209.

ZHANG, W. 2005. Replication cache: a small fully associative cache to improve data cache reliability. *IEEE Transactions on Computers 54*, 12, 1547–1555.

ZHANG, W., GURUMURTHI, S., KANDEMIR, M., AND SIVASUBRAMANIAM, A. 2003. ICR: In-Cache Replication for Enhancing Data Cache Reliability. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*. 291–300.

Received March 2007; revised August 2007, April 2008; accepted May 2008