A Run-Time Memory Protection Methodology

Udaya Seshua
NXP Semiconductors, BL Personal
Millenia 'D' Block, # 1
Murphy Road, Ulsoor
560008, Bangalore, India
udaya.seshua@nxp.com

Nagaraju Bussa
Philips Research
Philips Innovation Campus, #1
Murphy Road, Ulsoor
560008, Bangalore, India
nagaraju.bussa@philips.com

Bart Vermeulen
NXP Semiconductors, Research
High Tech Campus 5
5656 AE Eindhoven
The Netherlands
bart.vermeulen@nxp.com

Abstract - In this paper we present a novel methodology to help debug memory corruption errors during application debug. In this methodology an optimal balance between hardware and software instrumentation is chosen to check at run-time all memory accesses made by an application. To achieve this balance a set of benchmark applications is first analyzed to determine their memory access patterns. The analysis results are used to make our approach low-cost both from a software performance penalty and a hardware area point-of-view. Experimental results show that our innovative approach typically requires less than 2% of a CPU in silicon area for a less than 1% run-time performance overhead. Our method is both low-cost and applicable to high performance microprocessors as well as time-constrained embedded systems.

I. INTRODUCTION

To meet the demanding performance requirements for audio and video applications, the industry is integrating a complete system on a single chip, the so-called system-on-chip (SoC). A typical SoC comprises of multiple processor cores and dedicated hardware peripherals. On top of this hardware, several layers of software (drivers, operating system, streaming and control layers) are stacked to provide complete audio and video applications for use in domestic and mobile appliances. With the development of these applications comes the need to verify their correctness. Even though verification is successful in screening out the majority of errors in software using the simulation models of the SoC, it cannot guarantee that all software errors are removed before the real silicon becomes available.

In-situ debug of SoC applications is complicated because of the very limited observability into the various layers making up the complete system. Consequently a growing industry-average of 50% of the overall project duration is spent on debug after first silicon becomes available [1]. This can lead to higher development cost, slipping deadlines and the loss of (potential) customers. The US-based National Institute of Standards and Technology has reported that the cost of software defects in the United States alone was approx. $59.5 billion in 2002[2].

Memory-related bugs are among the most prevalent and difficult to catch of all software bugs, particularly in applications written in an unsafe language such as C/C++. Therefore, a good debug infrastructure that is capable of locating memory-related software bugs quickly is key to reducing the effort and resources spent on software debug.

In this paper we present a new methodology and the supporting infrastructure to help debug memory corruption errors during application development.

The paper is organized as follows. Section II gives an introduction to the run-time memory corruption problem. In section III we provide an overview of existing solutions for this problem. Section IV describes our new approach and in subsequent subsections the implementation details of the supporting infrastructure are provided. Section V describes the available options to make an optimal trade-off between the required silicon area and performance cost. Section VI summarizes our experimental results and Section VII concludes this paper.

II. PROBLEM DESCRIPTION

In an embedded system, a single memory access error can cause an application to behave unpredictably or even crash [3]. The application's code or data gets corrupted, causing wrong data to be used or incorrect instructions to be executed. These memory corruptions often do not directly cause the system to crash, but instead cause so-called delayed crashes. Because the distance in time between the actual root-cause and when the incorrect behavior is visible on the outputs of the system is very large in a delayed crash, it is very difficult for engineers to debug this type of error and find the root-cause.

The ability to detect memory corruption the instance it occurs would allow direct and appropriate action to be taken. During the application development process, a debugger tool can then guide the software engineer more accurately to the software bug and fix it. As a result a substantial amount of debug time and resources is saved.

III. PRIOR WORK

To address memory corruption problems, existing memory protection schemes define each memory region allocated by the application as valid and accessible memory. For each subsequent memory access, the address used is compared against the list of known valid regions. To implement this check, two separate approaches are commonly used: (1) a software-based approach, or (2) a hardware-based approach.

A. SOFTWARE-BASED PROTECTION METHODS

The software-based approaches either (a) instrument the application at compile and/or link time, or (b) execute the unmodified application code on an extended, virtual machine.

a) During application instrumentation, additional instructions are added to the application code to check every memory access, and generate an error when an invalid access is detected. Modifying the application often requires access to the source code of the application and/or its libraries, unless the instrumentation can be done at the object level.

Source-to-source translators, such as Ccured [4], can analyze and translate the application's code to insert run-time checks to validate all memory accesses. The resulting application will stop on an illegal access, rather than write into invalid memory space.

GCC's Bounds Checking [5] and Mudflap [6] extensions add instrumentation during the compilation step of the application code. They intercept all calls to the standard memory-allocation functions, such as *malloc*, *free*, *new*, and *delete*. In addition, they check whether allocated memory blocks are still valid and whether out-of-range read or write operations occur. In Section VI we compare the run-time performance of our method with the GCC Mudflap approach.

Purify [3] is an example of a memory usage debugging tool for C applications that inserts protection code at the assembly level instead.

b) A virtual machine that emulates the target processor can be extended to offer memory corruption detection during the execution of the unmodified application code. The advantage of this method is that no access to the source code of the original application is required. A drawback is that a sufficiently accurate and modifiable virtual machine implementation of the target processor has to be available.

The software-based methods described above all introduce a performance penalty during the execution of the application. This penalty is the direct result of performing all memory access checks completely in software. Typical numbers quoted in literature and measured in practice range from a 2x to 10x performance cost [8]. Most consumer applications however have to meet strict, real-time constraints. The severe performance cost introduced by the methods described above restricts, if not excludes, the use of these methods for consumer applications and are therefore not good enough for use in consumer devices.

B. HARDWARE-BASED PROTECTION METHODS

Hardware-based approaches shift the burden of checking the memory accesses from software to hardware, thereby reducing the performance penalty but incurring a new hardware cost. Common components that are (re)used for this purpose include (a) a Memory Management Unit (MMU), (b) a dedicated Memory Protection Unit (MPU) or (c) a processor's breakpoint module.

a) Each allocated memory buffer is placed in a separate page in the MMU. MMU page flags are used to prevent an application from accessing pages, which it is not allowed to read from or write to. The MMU page granularity still imposes restrictions on the use of this type of protection. For example a single memory buffer requires an entire page in the MMU. Given typical page sizes of 4K or 64K, allocating many smaller memory buffers can quickly become very wasteful on MMU resources and memory.
b) Using a dedicated memory protection module, such as ARM's MPU [9], the memory address space can also be divided into a fixed number of regions, each with a size selectable from a pre-defined, but fixed set. The base address is programmable and defines the start of the memory region. In the ARM implementation, it must be aligned to the selected region size. The usage of these support modules is limited by the typically small number of regions, the restrictions on the region alignment and their size.
c) Processor address breakpoints can also signal an illegal access to a memory region. The address range is then specified in two control registers for start and end address. An additional register is used to control the enabling/disabling of the breakpoint, choosing the virtual or physical address, and generating a breakpoint event for either in-bound or out-of-bound addresses. The advantage of this approach is that the granularity of the memory bounds can be a word or even a single byte. A memory buffer is therefore protected with very tight bounds, causing no wastage of memory. The disadvantage is that a processor only offers a very limited number of these breakpoints in hardware.

These hardware-based methods significantly improve the performance over the software-based methods, but as a draw back require both an increased amount of silicon area and a skilled programmer to utilize them for memory access checking. As a result these solutions all tend to be used on an ad-hoc basis.

One general drawback, shared by all methods described above is that they assume that either the software or the hardware as a fixed component that cannot be modified. In other words, either for a given hardware chip, the software code is instrumented, or for a given software application the available hardware functionality is reused to detect memory corruption at run-time. Both lead to sub-optimal implementations.

IV. INTEGRATED HW/SW APPROACH

Our approach combines the benefits of both existing hardware-based and software-based methods, while taking special care to make it more cost-efficient than both. In our approach, any memory access made by an application is checked at run-time against all allocated memory regions using a hybrid solution of hardware and software, allowing for a better trade-off to be made between the resulting application performance penalty and required silicon area.

We provide an on-chip Region Protection Module (RPM), to check all memory accesses in hardware at run-time, and an application programmer's interface (API), to efficiently control the RPM from application software.

The open-source GCC compiler has been extended to automatically instrument any given application's source code with the appropriate API calls to effectively monitor for any region violations at run-time. In the following subsections, we discuss in detail the on-chip region protection module, the API for the application software, and the flow for making the optimal trade-off between hardware and software instrumentation.

A. ON-CHIP REGION PROTECTION MODULE

An example, system-level usage of our on-chip Region Protection Module (RPM) is shown in Figure 1.

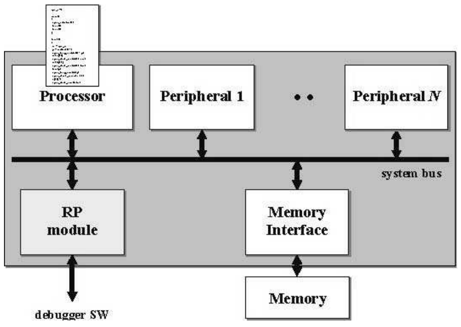


Fig. 1. Hardware Architecture Overview.

The processor executes the application software and is attached to a system bus. Also a set of dedicated peripherals and a memory interface to external memory attach to this same system bus. To this generic computer architecture we have added a RPM, which takes care of validating all memory accesses made by the processor via the system bus. The RPM control registers are memory mapped and accessible from the processor. The RPM can also be controlled and observed by external debugger software through a dedicated debug port, e.g. an IEEE 1149.1 Test Access Port. A block diagram of the RPM is shown in Figure 2.

A bus adapter translates the bus-specific protocol to basic read and write operations for the RPM core. The RPM controller is connected between this bus adapter and to a group of so-called Region Protection Units (RPUs). Each RPU is responsible for validating the address of a memory access on the system bus against a single, and programmable memory region. Note that we make a distinction between heap-based and stack-based memory regions. Details on the reasons for this difference are given in Subsection IV.B. Each bus operation is translated to RPU instructions by the controller. Valid memory regions are programmed using MMIO write operations from the processor to the RPM.

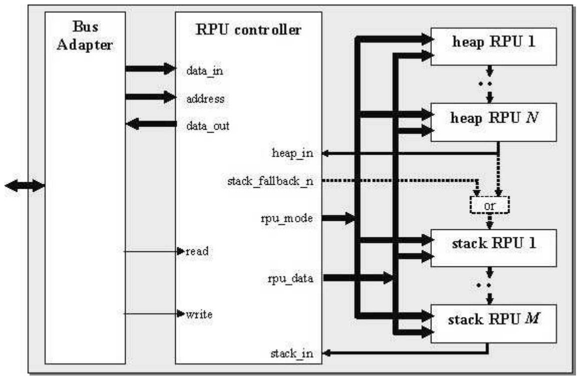


Fig. 2. Block diagram of the Region Protection Module.

We will focus first on the block diagram of a heap RPU, as given in Figure 3.

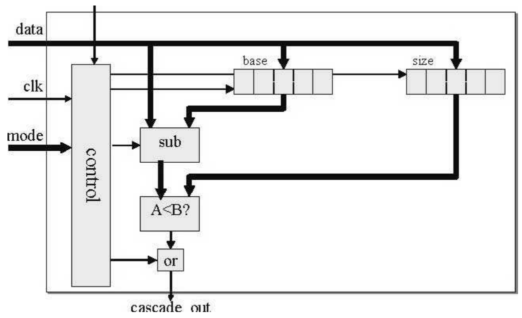


Fig. 3. Block Diagram of a heap RPU.

The heap RPU has two registers coupled to a data communication bus. The first register stores the base address value of the memory region, and the second register stores the size of the memory region. The heap RPU also contains a subtraction unit, which takes in an address value from the data communication bus and subtracts the value of the base register. The output of the arithmetic functional unit is compared to the value of the size register. The comparator output is routed to the output of the RPU via an OR-gate. This OR-gate enables the cascading of multiple RPUs inside a RPM. When one of the RPUs validates the memory operation, the RPM will indicate this.

The heap RPU controller allows programming of the base and size registers, validation of a given memory address, and clearance of the base and size registers when the associated buffer is released by the software application. The width of the size register and associated arithmetic logic is determined at design time, as discussed in Section V.

The stack RPU contains an additional register to store a function scope identifier with the base address and size of the memory buffer. A separate MMIO address is reserved for communicating this identifier to the RPM.

B. SOFTWARE API

We use a lightweight software API to allow easy and efficient use of the RPM hardware from the processor. This API consists of six functions:

- `void rpus_initialize()`
  This functions clears all settings from the RPM.
- `bool rpus_heap_enable(base,size)`
  This function is called when the application software allocates buffers on the heap by functions such as `malloc` and `new`. This function passes the base address and size of the buffer to the RPM. The size of the buffer is broadcast to all RPUs first to find and program the smallest, available RPU with this size value. Subsequently the base address is sent from the application code to this same RPU. When an RPU is programmed to protect this buffer, this function returns a true value. It returns a false value when no suitable RPU is available. The false return value is used to activate a fallback mechanism in software instead when all hardware resource are already in use.
- `bool rpus_stack_enable(base,size,id)`
  This function is called when buffers are allocated on the stack. The allocation of an available RPU and the return value are similar to those of the rpus_heap_enable function. However, in addition to the base address and size, also a function scope identifier is passed to the RPM and stored with the other buffer data. This allows all RPUs that were assigned to stacked-based memory buffers, to be released when the program leaves the corresponding function scope.
- `bool rpus_check_access(address)`
  Whenever there is a memory access, this function is called and communicates the address of the memory access to the RPM. It returns either a true value if a RPU validates the memory address, or a false value, if no RPU could. The latter value can optionally be used to subsequently validate the memory address against valid memory regions protected by software.
- `bool rpus_heap_disable(base)`
  When the program de-allocates heap-based memory buffer, this call frees up the RPU assigned to this buffer. The return value is true if an RPU existed in hardware that was protecting this buffer, or false if there was no such RPU. The latter value is used to determine whether the list of buffers protected in software also needs to be searched for the assigned RPU.
- `void rpus_stack_disable(id)`
  When the program leaves the scope of a particular function, its identification code is communicated to all RPUs. Any RPU that was assigned inside this function scope is freed up as a result.

V. HW/SW COST TRADE-OFF

The trade-off between how much hardware protection to include in a system chip design can be made based on an analysis of a set of benchmark applications that are to be executed on the target processor (see Figure 4).

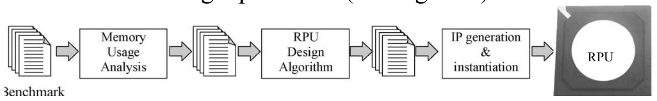


Fig. 4. Design Flow for Optimized RPU Hardware.

Benchmark applications are analyzed to extract their run-time memory allocation and access behaviors and obtain statistics on the maximum number of memory buffers that need to be simultaneously protected along with their sizes. Based on this analysis, an RPM is generated automatically with this number of RPUs to cover the hardware resource requirements of these benchmark applications. An extended GCC compiler is used for this analysis. Algorithm SINGLE_APP_RPM_DESIGN is used to design the RPM for a single benchmark application.

Algorithm 1: SINGLE_APP_RPM_DESIGN

As long as a benchmark application still performs a memory allocation or de-allocation, do

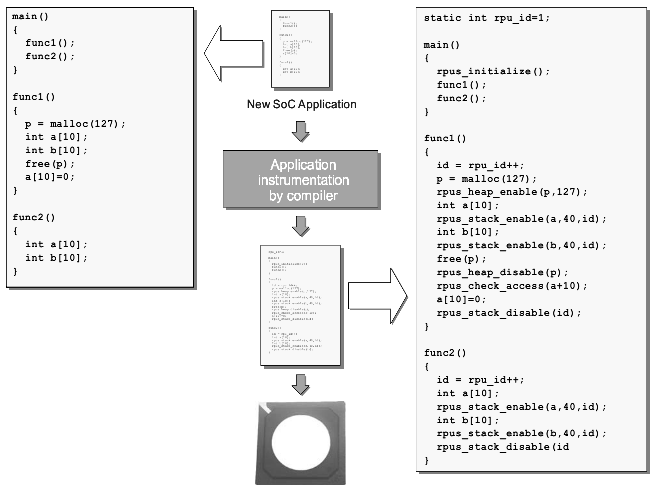
a) Upon a memory allocation, check if there are free RPUs large enough to protect the new buffer.
    i. If such RPUs exist, assign the buffer to the smallest RPU in this set.
    ii. If they do not exist, find the largest, free RPU that can handle only buffers smaller than the new buffer. Resize that RPU to the buffer size and assign the buffer to this RPU.
    iii. If there are no free RPUs, instantiate a new RPU with the size corresponding to the new buffer size, and assign the buffer to that RPU.
b) Upon a memory de-allocation, mark the corresponding RPU as free.

To design the RPM for multiple benchmark application, we use the iterative algorithm MULTI_APP_RPM_DESIGN.

Algorithm 2: MULTI_APP_RPM_DESIGN

a) The set of required RPUs is initially empty.
b) Run the SINGLE_APP_RPM_DESIGN algorithm using the memory statistics from the first benchmark application to yield an initial set of RPUs.
c) Successive runs use the memory statistics of the other benchmark applications and modify this set of RPUs. The resulting set of RPUs covers the memory protection requirements of all benchmark applications.

Afterwards, the RPM is included in the chip design process using the standard design flow. During application development and debug, we can now utilize the on-chip RPM to shift the burden of memory address checking from software to hardware. What remains is the run-time (re)configuration of the RPM at each memory allocation and de-allocation. This task is automatically performed for any source code provided by an extended GCC compiler.



```
main()
{
  func1();
  func2();
}

func1()
{
  p = malloc(127);
  int a[10];
  int b[10];
  free(p);
  a[10]=0;
}

func2()
{
  int a[10];
  int b[10];
}
```

```
static int rpu_id=1;

main()
{
  rpus_initialize();
  func1();
  func2();
}

func1()
{
  id = rpu_id++;
  p = malloc(127);
  rpus_heap_enable(p,127);
  int a[10];
  rpus_stack_enable(a,40,id);
  int b[10];
  rpus_stack_enable(b,40,id);
  free(p);
  rpus_heap_disable(p);
  rpus_check_access(a+10);
  a[10]=0;
  rpus_stack_disable(id);
}

func2()
{
  id = rpu_id++;
  int a[10];
  rpus_stack_enable(a,40,id);
  int b[10];
  rpus_stack_enable(b,40,id);
  rpus_stack_disable(id
}
```

Fig. 5. Application Instrumentation Flow.

VI. EXPERIMENTAL RESULTS

Figure 6 shows the analysis results of the amount of memory allocated and de-allocated by a small example application during its execution. Figure 7 shows the number of unique memory buffers allocated and de-allocated over time by the same example application as used in Figure 6. Figure 7 shows for example that a second memory buffer is allocated at clock cycle 312,928 and de-allocated at clock cycle 359,233. The darker bars represents allocations on the stack, the lighter bars allocations on the heap. Figure 8 shows how only five RPUs (1 stack, and 4 heap RPUs) can be used to protect all 11 allocated buffers by exploiting the fact that not all memory buffers need to be simultaneously protected.

We have also taken a subset of applications from an independent, embedded application benchmark set called MiBench [10] to compare the impact of our method with GCC's existing software-based memory protection method, called MudFlap. We have looked at the impact on the speed of the application and the required amount of silicon area. We have normalized the speed of the original application to 1, and normalized the area required for a CPU to 1. The normalized speed when each protection method is implemented for the benchmark applications is shown in Figure 9.

Figure 9 shows that the GCC software-based method slows down the speed of the application considerably (first bar), sometimes by as much as 50x. In contrast our approach yields application speeds that are for most cases less than 10% from the original (second bar). If furthermore, the `rpus_check_access(address)` functional is instead implemented by directly observing bus accesses (i.e. direct bus 'snooping'), the performance penalty is reduced even more to often less than 1-2% (third bar). The required normalized silicon hardware is shown in Figure 10.

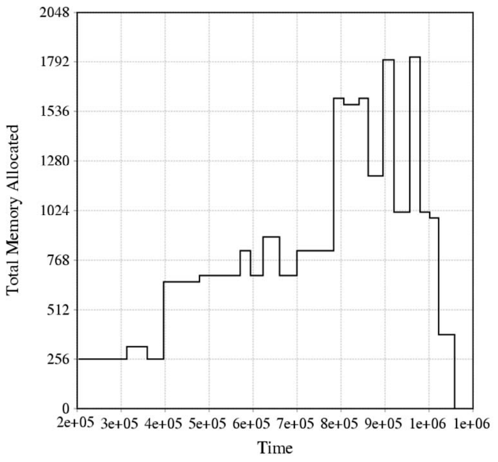


Fig. 6. Memory allocation over time.

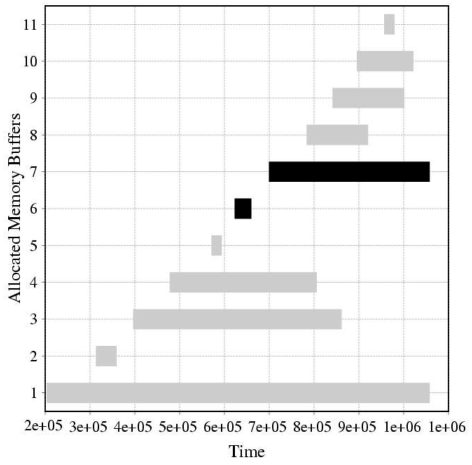


Fig. 7. Buffer allocation over time.

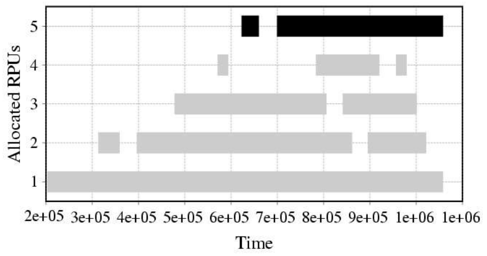


Fig. 8. RPU allocation over time.

Figure 10 confirms that the GCC software-based approach does not require any additional hardware area, as all memory protection is performed in software. As indicated, this also causes the significant performance drop as shown in Figure 8. To implement the same memory protection with our approach requires less than 2% of a RISC CPU in silicon area, as shown in Figure 9.

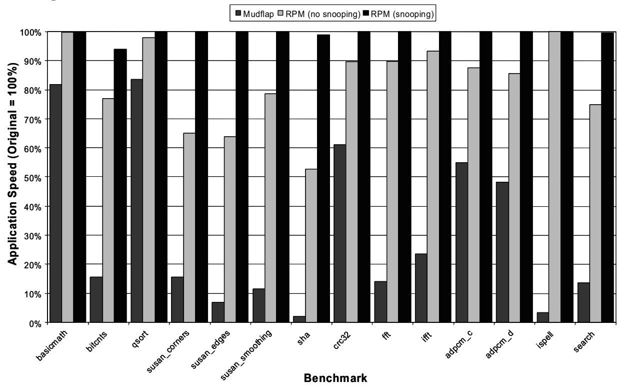


Fig. 9. Application speed with each method implemented.

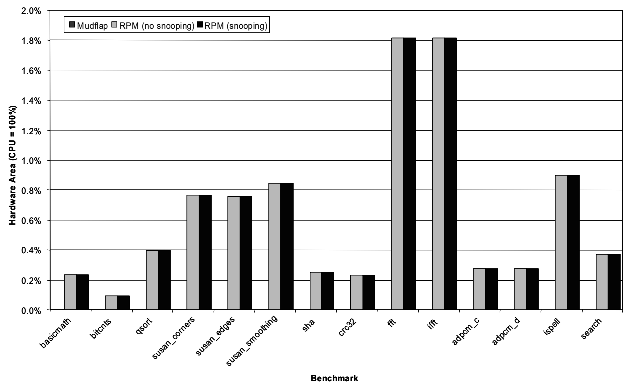


Fig. 10. Required silicon area.

VII. CONCLUSION

We have presented a novel, HW/SW-based memory protection methodology for embedded systems. Our approach has three distinct advantages over prior work:

1. Due to its lightweight API design, the resulting software overhead is significantly reduced compared to software-only protection methods, as most, if not all, the valid region management and validation is shifted from software to hardware.
2. The amount of hardware (e.g. the number of stack and heap RPUs) can be optimized using design-time application analysis.
3. This scheme allows for the software to take over in those cases where the hardware RPU resources are all in use. This offers a seamless transition from hardware-based to software-based protection without requiring user intervention. Consequently our approach only comes with a performance cost for those memory regions that cannot be protected in hardware, whereas current state-of-the-art software-based methods impose this overhead for all memory regions.

By combining an effective memory access monitor module in hardware with efficient application instrumentation, we have shown that run-time memory protection is a low-cost option for both high performance microprocessor design and real-time, embedded applications. The functionality provided by this HW/SW architecture can be used both to reduce the time and effort spent on debugging software code during application development, and to provide enhanced support against security threats.

VIII. REFERENCES

[1] B. Roberts, “The verities of verification”, *Electronic Business*, January 2003.

[2] National Institute of Standards and Technology, www.nist.gov.

[3] R. Hastings and B. Joyce, “Purify: Fast detection of memory leaks and access errors,” in *Proceedings of the Winter USENIX Conference*, 1992, pp. 125–136.

[4] G.C. Necula et al., “CCured: Type-Safe Retrofitting of Legacy Code”, University of California, Berkeley.

[5] R.W.M. Jones and P.H.J. Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs,” in *the third International Workshop on Automated and Algorithmic Debugging*, 1997, pp. 13–26.

[6] F. Eigler, “Mudflap: Pointer Use Checking for C/C++”, GCC Developers Summit, May 25–27, 2003, Ottawa, Ontario, Canada.

[7] N. Nethercote and J. Seward, “Valgrind: A Program Supervision Framework”, in *Electronic Notes in Theoretical Computer Sience 89*, No. 2, 2003.

[8] C. Cowan et al., “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", in *Proceedings of 7th USENIX Security Conference*, 1998, pp. 63-78.

[9] ARM Ltd., “ARM1156T2F-S Technical Reference Manual”, 2005.

[10] M.R. Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite” in *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*, Austin, TX, December 2001