Fast Bounds Checking Using Debug Register

Tzi-cker Chiueh

Computer Science Department
Stony Brook University
chiueh@cs.sunysb.edu

Abstract. The ability to check memory references against their associated array/buffer bounds helps programmers to detect programming errors involving address overruns early on and thus avoid many difficult bugs down the line. This paper proposes a novel approach called *Boud* to the array bounds checking problem that exploits the debug register hardware in modern CPUs. *Boud* allocates a debug register to monitor accesses to an array or buffer within a loop so that accesses stepping outside the array's or buffer's bound will trigger a breakpoint exeption. Because the number of debug registers is typically small, in cases when hardware bounds checking is not possible, *Boud* falls back to software bounds checking. Although *Boud* can effectively eliminate per-array-reference software checking overhead in most cases, it still incurs a fixed set-up overhead for each *use* of an array within a loop. This paper presents the detailed design and implementation of the *Boud* compiler, and a comprehensive evaluation of various performance tradeoffs associated with the proposed array bounds checking technique. For the set of real-world network applications we tested, including Apache, Sendmail, Bind, etc., the latency penalty of *Boud*'s bounds checking mechanism is between 2.2% to 8.8%, respectively, when compared with the vanilla GCC compiler, which does not perform any bounds checking.

1 Introduction

Checking memory references against the bounds of the data structures they belong to at run time provides a valuable tool for early detection of programming errors that could have otherwise resulted in subtle bugs or total application failures. In some cases, these software errors might lead to security holes that attackers exploit to break into computer systems and cause substantial financial losses. For example, the buffer overflow attack, which accounts for more than 50% of the vulnerabilities reported in the CERT advisory over the last decade [4, 20, 15], exploits the lack of array bounds checking in the compiler and in the applications themselves, and subverts the victim programs to transfer control to a dynamically injected code segment. Although various solutions have been proposed to subjugate the buffer overflow attack, inoculating application programs with strict array bounds checking is considered the best defense against this attack. Despite these benefits, in practice most applications developers still choose to shy away from array bounds checking because its performance

P. Stenström et al. (Eds.): HiPEAC 2008, LNCS 4917, pp. 99–113, 2008.

overhead is considered too high to be acceptable [14]. This paper describes a novel approach to the array bounds checking problem that can reduce the array bounds checking overhead to a fraction of the input program's original execution time, and thus make it practical to apply array bounds checking to real-world programs.

The general problem of bounds checking requires comparing the target address of each memory reference against the bound of its associated data structure, which could be a statically allocated array, or a dynamically allocated array or heap region. Accordingly, bounds checking involves two subproblems: (1) identifying a given memory reference's associated data structure and thus its bound, and (2) comparing the reference's address with the bound and raising an exception if the bound is violated. The first subproblem is complicated by the existence of pointer variables. As pointers are used in generating target memory addresses, it is necessary to carry with pointers the ID of the objects they point to, so that the associated bounds could be used to perform bounds checking. There are two general approaches to this subproblem. The first approach, used in BCC [5], tags each pointer with additional fields to store information about its associated object or data structure. These fields could be a physical extension of a pointer, or a shadow variable. The second approach [13] maintains an index structure that keeps track of the mapping between high-level objects and their address ranges, and dynamically searches this index structure with a memory reference's target address to identify the reference's associated object. The first approach performs much faster than the second, but at the expense of compatibility of legacy binary code that does not support bounds checking. The second subproblem accounts for most of the bounds checking overhead, and indeed most of the research efforts in the literature were focused on how to cut down the performance cost of address-bound comparison, through techniques such as redundancy elimination or parallel execution. At the highest compiler optimization level, the minimum number of instructions required in BCC [5], a GCC-derived array bounds checking compiler, to check a reference in a C-like program against its lower and upper bounds is 6, two to load the bounds, two comparisons, and two conditional branches. For programs that involve many array/buffer references, software-based bounds checking still incurs a substantial performance penalty despite many proposed optimizations. In this paper, we propose a new approach, called *Boud*[1], which exploits the debug register hardware support available in mainstream CPUs to perform array bounds checking *for free*. The basic idea is to use debug registers to watch the end of each array being accessed, and raise an alarm when its bound is exceeded. Because debug registers perform address monitoring transparently in hardware, *Boud*'s approach to checking array bounds violation incurs no *per-array-reference overhead*. In some cases, hardware bounds checking is not possible, for example, when all debug registers are used up, and *Boud* falls back to traditional software bounds checking. Therefore, the overhead of *Boud* mainly comes from debug register set-up required for hardware bounds checking, and occasional software-based bounds checking.

[1] BOunds checking Using Debug register.