Journal of Instruction-Level Parallelism 9 (2007) 1-26 Submitted 2/07; published 6/07

Accelerating Meta Data Checks for Software Correctness and Security

Weihaw Chuang WCHUANG@CS.UCSD.EDU
Satish Narayanasamy SATISH@CS.UCSD.EDU
Brad Calder CALDER@CS.UCSD.EDU
University of California, San Diego
Department of Computer Science and Engineering
9500 Gilman Drive
La Jolla, CA 92093-0404 USA

Abstract

As high GHZ processors become prevalent, adding hardware support to ensure the correctness and security of programs will be just as important, for the average user, as further increases in processor performance. The goal of our research is to focus on developing compiler and hardware support for efficiently performing software checks that can be left on all of the time, even in production code releases, to provide a significant increase in the correctness and security of software.

In this paper we focus on the performance of checking the correctness of pointers. We focus on pointers since a significant amount of bugs and security issues (buffer overflows) in programs are due to memory bugs resulting from incorrect usage of pointers. To determine if a pointer reference is correct many techniques require additional information to be kept track of called *meta-data*. The meta-data is checked when a pointer is dereferenced to verify some property about the pointer or the object. The first part of our paper focuses on where to efficiently keep track of this meta-data information and the overheads for performing safety checks like bounds checking and dangling pointer checks. We then focus on architecture extensions to reduce the overhead of these meta-data checks. We examine these optimizations in the presence of two meta-data checking applications – bounds checking and dangling pointer checks and show that we can reduce the overhead of these pointer checks from 148% down to 21% on average.

1. Introduction

Computer system trends have increased the importance of providing efficient solutions to finding and preventing software bugs. Lower hardware costs and increasing hardware reliability have significantly reduced hardware's importance in terms of total computer cost [1, 2]. This has increased the software's component in the total cost of ownership of a system, due to software's increasing complexity, and especially bugs. In addition, with the wide spread use of the Internet and how easy it is to release patches, software is released with more potential bugs than in the past. Given these trends it is just as important to examine efficient compiler and hardware support for software correctness, security, and debugging as it is to increase the performance of the next generation of processors.

In this paper we focus on the performance of dynamically checking the correctness of pointers. We focus on pointers since a significant amount of bugs in programs are related to memory corruption bugs dealing with the pointers [3]. To determine if a pointer is correct, many dynamic software checking techniques require additional information to be kept track of along with each pointer, which is called *meta-data* of a pointer. Checks are performed using the meta-data when a pointer is dereferenced to verify some property about the pointer or the object. The two example meta-data checking techniques we examine in this paper for pointer correctness are bounds checking and dangling pointer checks. Bounds checking checks a pointer dereference to make sure it is within the bounds of the object being dereferenced, and if not an exception is raised. A dangling pointer check, checks a pointer dereference to make sure the pointer still points to a valid object and the object it last thought it was pointing to. Meta-data is used for both of these dynamic checks to determine if the pointer's usage is valid.

The first part of our paper focuses on where to efficiently keep track of this meta-data information. The meta-data for some software checks, such as bounds checking, can be stored with the pointer or alternatively it can be stored with the object itself. We find that storing the meta-data with the object, instead of with the pointer, scales better in terms of performance as the amount of meta-data that needs to be kept track of increases. We then examine *Meta-Data Checking* (MDC) architecture extensions to efficiently do the meta-data checks. The goal of all of these techniques is to reduce the overhead of meta-data checks enough so that the checks can be left in the release versions of software. The contributions of this paper are:

- We provide a detailed trade-off (micro-architectural effects) analysis to determine where to store the meta-data for bounds checking and dangling pointer checks. We show that storing the meta-data with the object provides better performance and will scale better if additional meta-data needs to be tracked for doing more checks for a pointer.
- We propose architecture and ISA extensions to reduce the average overhead of meta-data checks to 21%, when performing both bounds checking and dangling pointer checks. In comparison, existing software techniques, result in 148% slowdown for the same checks.

2. Methodology

In this section we describe our compiler that we used to implement the meta-data checks we examined in this paper, and the simulation infrastructure to gather our results. All our simulations are based on x86.

2.1. Compiler

We build our compiler infrastructure out of 2.95 GCC. The meta-data checks we examine in this paper include bounds checking and dangling pointer checks. We implemented these two checks starting from a bounds checking patch provided by Greg McGary [4]. McGary's infrastructure performs bounds checking of C references, including automatic bounds generation for static and dynamically allocated objects using pointer meta-data (which is conventionally referred to as fat pointers) and static bounds information.

We modified the McGary version of gcc in several respects. First, we modified the compiler to optionally generate the object meta-data that will be described in subsection 4.1.. Second, we modified it to use the x86 `bound` instruction, instead of a sequence of compare-branch-trap x86 instructions to do bounds checking. Third, we eliminated redundant bounds instructions by modifying common-subexpression-elimination to remove redundant bounds in a trace region. Forth, we add the dangling pointer check for stack and heap objects. As tag checks for statics objects is not necessary, we skip dangling checks on them. We also model the meta-check instructions described in Section 5.

We verified that McGary's bounds check detects all buffer overflow attacks in Wilander's test case [5]. Subsequent major functionality changes were reverified with this test case. We also verified that the software bounds checker was able to detect bounds violation in the AccMon benchmarks [6].

2.2. Simulation Model

We used SimpleScalar 4.0 x86 Tool Set [7] for simulating our x86 binaries. The configuration is given in Table 1 and based loosely on an AMD Athlon processor, as this represents a widely deployed modern desktop system, and a pipeline that is more reasonable to emulate.

Fetch Width	4 inst
Issue Width	4 inst
Func Units	4-ialu, 1-imult, 2-mem, 3fpalu, 1-fpmult
Reorder buf	RUU: 32, LSQ: 32
L1D	16KB, 2 way, 64B Block, 3 cycle latency
L1I	16KB, 2 way, 64B Block, 3 cycle latency
L2 Unified	2MB, 16 way, 64B Block, 20 cycle latency
DTLB	128 entry, 30 cycle miss penalty
ITLB	64, 30 cycle miss penalty
Memory	275 cycle latency
Branch Pred	16K meta chooser between gshare (8K entry) and bimodal table (8k entry); 16 Return Address Stack; 512 BTB; 10 cycle misprediction penalty

Table 1: Simulation model based on the AMD Athlon.

To better understand sources of delays in the processor pipeline, we modified SimpleScalar to classify every cycle in terms of generic delay sources. If a delay prevents useful instruction execution for that cycle, then that cycle is categorized by that delay type, otherwise that cycle is counted towards execution *ex.* A cycle is attributed to execution in this case, even if some other delay event is occurring, because the out-of-order pipeline is still doing useful work. Data-cache misses often stall data-dependent instructions, completely starving the pipeline, and are classified as *dc.* Because we want to know when data-cache misses occur, even though useful instructions are being executed, we classify cycles when this combination is the case as *dc/ex.* Front-end pipeline starving events are classified as either branch misprediction *brm*, or other front-end stalls (i.e.instruction cache miss) *fe.* Almost all of our results are classified with these five breakdowns as stacked graphs with the y-axis labeled `Normalized Execution`.

2.3. Benchmarks and Simulation Points

For our results we use programs from the SPEC INT 2000 benchmarks. These are `bzip`, `crafty`, `gzip`, `mcf`, `parser`, `twolf` and `vpr`. We do not provide results for the other SPEC benchmarks, either because (a) they did not compile with our baseline McGary compiler described above, or (b) they did not completely run or run correctly under the new x86 SimpleScalar we are developing jointing with Michigan. We simulated each program using the reference input for 100 million instructions (for baseline) at a representative simulation point chosen by SimPoint [8].

For our analysis we generate different binaries to look at the different bounds checking approaches examined in the rest of this paper. For example, the baseline binary has no bounds checking at all, and we have another binary that includes the `bound` instruction to perform bounds checking, and another for checking dangling pointers, etc. Since we have multiple versions, we need to make sure that we simulate the exact same part of the program's execution across these different binaries. To do this we use the single simulation point for the baseline binary, and we perform binary matching to find the exact same code sequence (a unique one) in the bounds checked binaries that corresponds to the simulation point. We then used this to determine when to start simulation for the binary, and did similar binary matching analysis to figure out when to stop simulation.

Since different number of instructions are simulated between the different binaries to represent the same part of execution, we report results in terms of number of cycles executed normalized to the baseline binary without any safety checking.

3. What is Meta-Data?

Sullivan and Chillarege [3] provided a detailed analysis of the failure reports from the IBM mainframe MVS operating system. They found that memory corrupting bugs are more likely to cause a high priority bug report by a ratio of three-to-one. Memory corrupting bugs often allows the program to continue for some time, potentially corrupting data and obscuring the bug's identity, instead of stopping at the point of failure. They found the top five causes of these memory corruption bugs are buffer overflow 20%, deallocated memory 19%, corrupt pointers 13%, type mismatch 12%, and 13% unknown [3]. Over half of the data-corrupting failures are directly due to memory mismanagement.

Buffer overflow attacks exploit bugs to deny service or even take over the program. As the name implies the adversary injects arbitrary data through a program's external interface e.g. network sockets, file IO, or command line arguments to overwrite program data. This causes the program to crash or execute a program of the attacker's choosing [5]. CERT data [9] from 1997 to 2003 shows that 50% or more of CERT security adversaries are due buffer overflow attacks. A 2004 study found that unpatched and Windows XP SP1 connected to the Internet would be taken over in less than four minutes [10].

Runtime safety checks using meta-data can prevent many of the above software failures. Software safety checks and maintenance activities often require some persistent knowledge of the object(s) they are operating on. *Meta-data* is a catch-all term for this persistent data, that exists outside the normal application activity. It usually is not visible to the programmer, having been automatically inserted by the compiler or some other tool. The following is a set of safety checks and memory management techniques that use meta-data to find and prevent the above top five memory corruption causes.

- Bounds Checking - Bounds checking verifies that a memory reference of an object, or array falls within the bounds of that structure. The meta-data used to perform this check is the object's low and high bounds [4].
- Dangling Pointer Checking - C and C++ require programmer managed memory. Freeing memory still referenced by the program results in a dangling pointer. If referenced, the stale pointer will incorrectly access the freed memory. We can tag the pointer and the object with a unique ID upon object allocation. If the object is freed, the object tag is cleared. Stale pointers are then identified by a tag mismatch with the object. The stored meta-data to perform this check is a tag stored with the pointer and a tag stored with the object [11].
- Garbage Collection - Garbage collectors perform automatic management of memory. Because it periodically scans through pointer references and marks used memory, it needs to temporarily store meta-data. A mark is stored in the object's meta-data to keep track of which objects have been visited [12]. Additional meta-data stored with an object can include the location of the pointers within the object, which enables the garbage collector to continue sweeping the heap.

For the rest of the paper, we will use both bounds checking and dangling pointer checking to examine where the meta-data should be stored, the efficiency of meta-data checks, and optimizations to reduce the meta-data check overhead.

4. Where to Store the Meta-Data and the Performance Overhead

As described in the prior section, software checks, such as bounds and dangling pointer checking, require additional persistent memory called meta-data to store the bounds or tag information. In this section we focus on examining the performance trade-off between storing this meta-data either along with the pointer or with the object.

4.1. Meta-Data Options

Figure 1(a) shows the two standard options for where to store the meta-data. For some checks, the *meta-data* can be, or needs to be, associated with the pointer to the object, which we call *Pointer Meta-Data (PMD).* Another option is to store the *meta-data* with the object itself, which we call *Object Meta-Data (OMD)* in Figure 1(a). For some checks, where to store the meta-data is an implementation option, whereas for other checks the information needs to be stored as either PMD or OMD. We use bounds checking and dangling pointer checks to demonstrate this.

For bounds checking, the high and low bounds are typically stored adjacent to the pointer as PMD shown in Figure 1(b). This is also called a fat-pointer [4, 13, 14]. Because the bounds information is directly associated with the pointer, obtaining the meta-data is fast and handles the problem of interior or out-of-bounds pointers due to pointer arithmetic. Interior or out-of-bounds pointers makes it difficult to associate a pointer to its object as

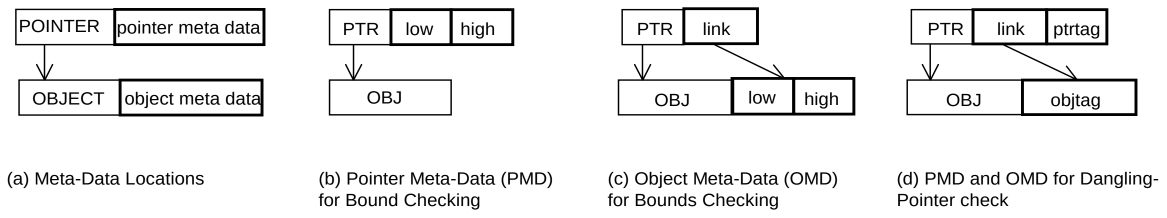

POINTER
pointer meta data
OBJECT
object meta data
(a) Meta-Data Locations
PTR
low
high
OBJ
(b) Pointer Meta-Data (PMD) for Bound Checking
PTR
link
OBJ
low
high
(c) Object Meta-Data (OMD) for Bounds Checking
PTR
link
ptrtag
OBJ
objtag
(d) PMD and OMD for Dangling-Pointer check

Figure 1: Meta-data Representations. An arrow indicates a pointer to data associated with the object. Highlighted blocks are meta-data.

the pointer no longer references the base of the object. Alternatively, we propose that the meta-data for bounds checking could be stored with the referent object as OMD shown in Figure 1(c). For this option, a *link* is stored adjacent to the pointer, which will provide the address to the location where the object meta-data is stored. The link is necessary largely due to interior and out-of-bounds pointers.

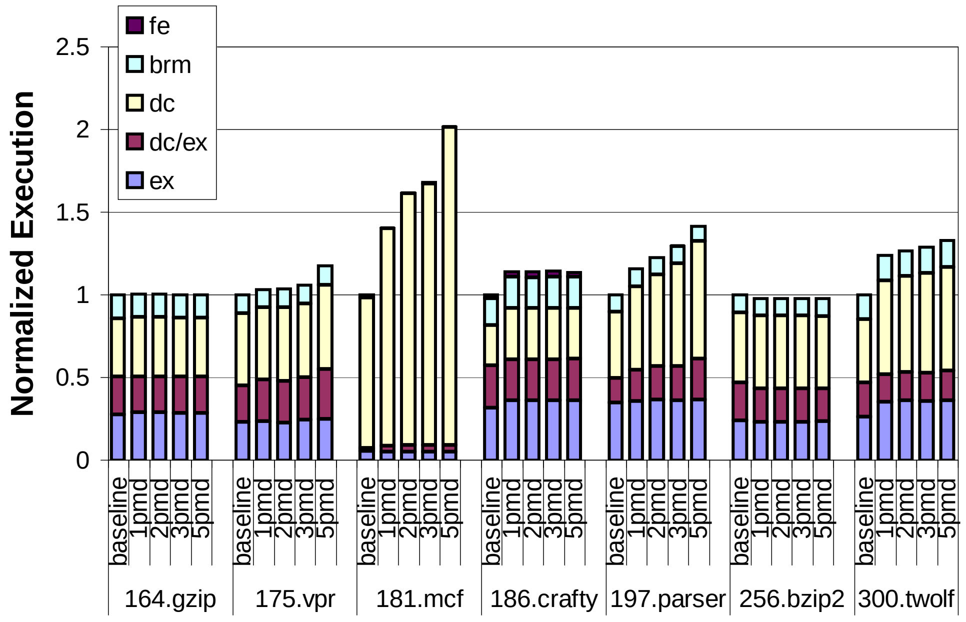
The other example we focus on in this paper is the dangling pointer checks. The meta-data for this check needs a pointer tag stored as PMD and an object tag stored as OMD. This is shown in Figure 1(d). Just as with the OMD bounds checking, a link is required as part of the PMD to find the object tag stored as part of the object meta-data [11, 15].

4.1.1. Meta-Data Overhead

Depending on where the meta-data is stored, as a PMD or OMD, the performance overhead will vary. This is because, the two representations will have different cache spatial locality. To examine this tradeoff, we ran experiments allocating different number of PMD and OMD words for all pointers and allocated objects. At each pointer reference (each time the pointer register was used in a memory operation) we access the last meta-data word. Therefore the overhead measured comes from copying and maintaining the meta-data and accessing the last meta-data word. For these results we broke the execution time into the percent of execution time (cycles) fetch was stalled (fe), the execution due to branch misprediction (brm), data cache misses (dc), overlapped data cache miss with execution (dc/ex), and execution (ex) where there were no stalls.

In Figure 2(a), we compiled the programs so that there was 1 (1pmd), 2 (2pmd), 3 (3pmd) or 5 (5pmd) extra words associated with the pointer representing the effects of having PMD of that size. The additional overhead occurs from two sources with PMD. The first overhead comes from copying the meta-data. Every pointer assignment during execution has to also copy the pointer meta-data to the new pointer. The increase due to this can be seen in `twolf` as the number of execution cycles went up. The more dominant increase in overhead comes from the increase in data cache misses (dc) from the pointers with PMD. This effect is seen for the data cache sensitive benchmarks like (`mcf`, `parser` and `twolf`).

In Figure 2(b), we experiment with varying OMD sizes. We store 2 (2omd), 3 (3omd), 6 (6omd), and 9 (9omd) extra words along with each allocated object. In addition, each



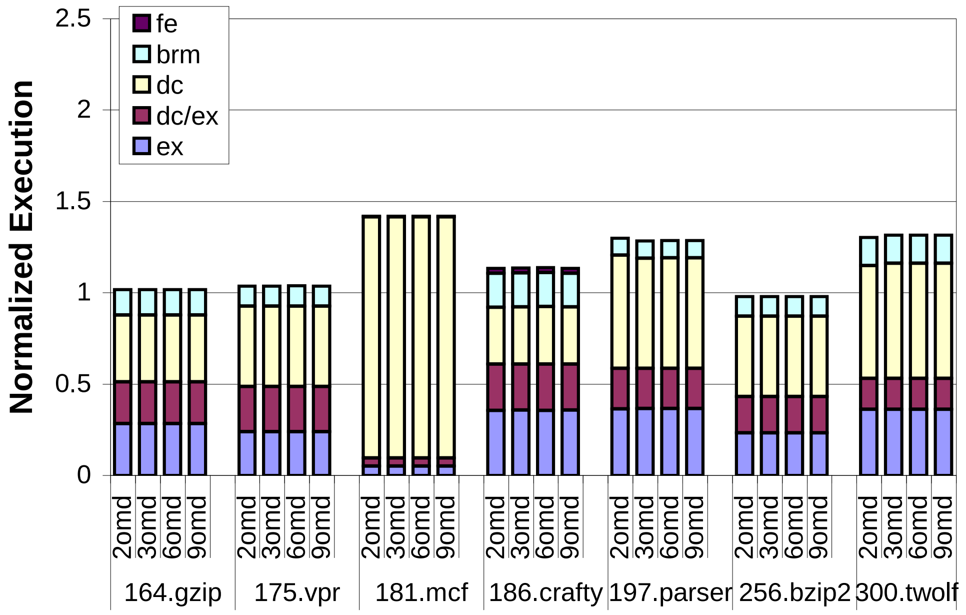


Figure 2: Performance overhead for maintaining pointer meta-data (top graph) and object meta-data (bottom graph). Results are shown for various sizes of meta-data.

pointer has 1 extra word, which provides the *link* from the pointer to the OMD as shown in Figure 1(c). Irrespective of the size of OMD, the overhead has a fixed cost of copying just the link word on every pointer assignment as opposed to copying all the meta-data in the case of PMD. The size of the pointer is also a constant two words (one word for the pointer itself and another for the link). The graph shows a nearly flat trend even as larger object meta-data sizes are allocated.

Storing meta-data with the objects scales better than storing it with the pointer, especially for programs like `mcf` and `parser` because (a) there are many more pointers stored in memory than objects, and (b) storing the meta-data with objects allows sharing of meta-data among the multiple pointers that point to the same object.

4.2. Storing Meta-Data for Bounds and Dangling Pointer Checks

We now examine the overheads of implementing bounds checking and dangling pointer checks and show how these overheads differ based on the layout used for storing meta-data.

4.2.1. Bounds Checking

Bounds checking uses the low and high boundary information associated with each memory object to determine if an out-of-bounds pointer reference has occurred. This is done for each source code pointer dereference or array reference. The x86 instruction set has an explicit instruction `bound` for performing bounds checking as shown in Figure 3(a) and (b). The code example assumes that the pointer is stored in register `ptr_reg` and the base address for the two words storing the high and low bounds is the second parameter. Figure 3(a) assumes the meta-data is stored as PMD as in Figure 1(b). The other option would be to store the bounds as OMD as in Figure 1(c), and Figure 3(b) shows the code for this. In this case, the link pointer is loaded, and then passed to the `bound` instruction.

The differences between storing the bounds meta-data as OMD vs PMD are:

- Sharing of Meta-Data - Storing the meta-data with the object will allow the meta-data to be shared across several pointers to the same object.
- Number of Pointers vs Number of Objects - Related to the above point is that some programs have many more pointers than objects. For example, programs like `mcf` and `parser` where each object has N pointers. For these programs, storing the bounds as PMD requires significantly more storage (and data cache usage) than storing them with the object. Storing meta-data with the object enables sharing them between pointers pointing to the same object.
- Reducing the PMD to 1 Word - Moving the meta-data to the object reduces the PMD from 2 words down to 1 word, and this is the link word to the object meta-data.
- Overhead of Extra Link Load - The OMD approach has the additional overhead of loading the link register. Note, that the link register overhead for the OMD case can actually be fairly small. This is because the link register can be hoisted to occur at the same time as the pointer load. If these both overlap, then the cost of the link load can be minimal.

```
bound ptr_reg, [base_reg+4]         mov   [base_reg+4], link_reg
                                    ...
                                    ...
                                    bound ptr_reg, [link_reg]



(a) PMD x86 Bound Instruction      (b) OMD x86 Bound Instruction
```

```
mov [base_reg+4], link_reg
mov [link_reg], objtag_reg
mov [base_reg+8], ptrtag_reg
cmp objtag_reg, ptrtag_reg
jeq done
trap

(c) Dangling Pointer Check
```

Figure 3: x86 implementation of the bounds instruction storing the meta-data with the pointer (a), and storing the meta-data with the object (b). (c) shows the pseudo code for performing the dangling pointer check where the link register and pointer tag are stored as pointer meta-data and the object tag is stored as object meta-data.

```
---------------------------
bound ptr_reg, [base_reg+4]
---------------------------
load [base_reg+4], low_reg
cmplt_trap ptr_reg, low_reg
load [base_reg+8], high_reg
cmpgt_trap ptr_reg, high_reg
```

Figure 4: The baseline micro-op expansion of the x86 Bound Instruction.

For C, several researchers use the PMD representation for bounds checking [4, 13, 14]. Others [16, 9] use a table lookup on the pointer address to determine the bounds. The table lookup scheme has the advantage in that it is not necessary to change the memory layout of the data objects. The meta data required to do a bounds check is obtained by doing a table lookup on the bounds meta-data table. Since C allows interior pointers, a fast hash lookup on the object address cannot be done, and instead we have to use tree search which would incur significant performance overhead. We therefore, concentrate on the PMD and OMD representation for our analysis.

4.2.2. Dangling Pointer Checks

Dangling pointer check determines if a referenced object has been freed and potentially reallocated, but incorrectly accessed afterward with the old pointer. It does this by associating a tag with the pointer and a second tag with the object, with the property that they must match. At object creation, a unique tag id is assigned to both the pointer, and object tags. When the object is freed, the object tag field is cleared. A pointer dereference to the object performs a tag check. If they mismatch then the pointer must point to an object that's been either freed or reallocated. The x86 pseudo code for implementing a dangling pointer check is shown in Figure 3(c). The meta-data for the dangling pointer needs to be stored as in Figure 1(d), where there is a link and pointer tag stored as pointer meta-data, and the object tag is stored as object meta-data.

4.3. Meta-Data Checking Overhead

When using bounds checking or dangling pointer checking, the checks occur at pointer dereferences, which can create large run-time overhead. Figure 5 shows the overhead for using the bounds checking instruction in Figure 3(a), where it is translated into the micro-op sequence in Figure 4(a) when executed in the pipeline. The second bar in Figure 5 shows the results for storing the bounds as PMD as in Figure 3(a). The first bar shows the results for storing the bounds as OMD as in Figure 3(b). The overhead of bounds checking is 81% on average when the bounds are stored in PMD but is 48.4% when the bounds are stored in OMD. The overhead comes from increased number of instructions from having to copy the pointer meta-data, the additional micro-ops to perform the check, and the increase number of cache misses.

As part of this study, we also want to examine the effect of performing multiple safety meta-checks on a pointer at the same time. In addition, looking farther into the future having multiple forms of meta-data stored with an object can potentially even aid hardware optimizations.

To examine the effect of performing multiple safety checks, we also provide results in Figure 5 for performing both bounds checking *and* dangling pointer checks for pointers at the same time. This is equivalent to executing the code in Figure 3(a) and (c) at the pointer dereference when the bounds are stored as PMD, or executing the code in Figure 3(b) and (c) at the pointer dereference when the bounds are stored as OMD.

To perform the combined check for PMD, the pointer-meta data is now 4 words wide since it contains the high and low bounds, a link to the object meta-data, and the dangling pointer tag. Then the object meta-data contains just the dangling object tag. To perform

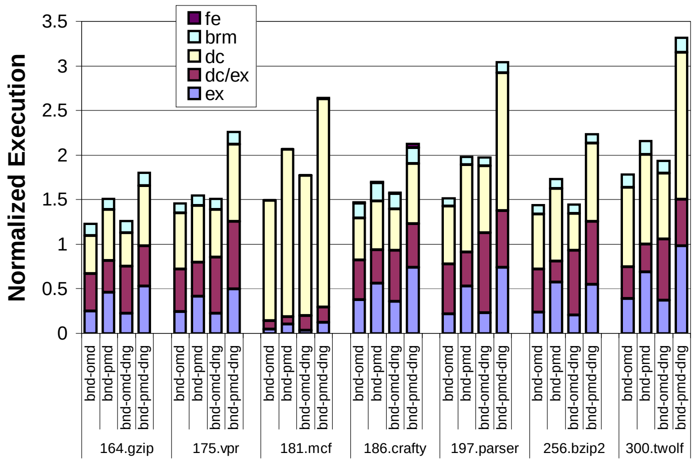


Figure 5: Bounds and combined Dangling Pointer and Bounds check overhead.

the combined check for OMD, the pointer-meta data is only 2 words wide since it contains only a link to the object meta-data, and the dangling pointer tag. Then the object meta-data contains 3 words, which includes the low and high bounds and the dangling object tag.

The fourth bar in Figure 5 shows results for bounds plus dangling checks where the bounds information is associated with PMD. The third bar shows results for doing both the checks, but for these bounds information is associated with OMD. The performance overhead increases greatly due to the wider pointer-meta data as we saw in our earlier results in Figure 2.

5. Meta Data Checker

The performance overhead of meta-data checks needed for bounds checking and dangling pointer checks shown in the previous section is still too high for these safety checks to be incorporated into released software. In this section we examine *Meta-Data Checking* (MDC) architecture extensions to reduce the overhead of meta-data checks. The architecture extensions include, extending the x86 ISA with a new instruction, called the *meta-check* instruction and the necessary hardware support to implement and use it.

5.1. Motivation for Meta Checker Instruction

The special meta-check instruction (explained later) is designed to meet following goals which strive to reduce the performance overhead and at the same time provide enough flexibility to support a variety of checks that need meta-data.

- Reduce Additional Instructions in Binary to Perform the Check - As shown in Figure 3 the dangling pointer check executes about five x86 instructions for each check (around 7 micro-ops expanding out the address generation). This can result in register spill and consume fetch bandwidth, which can adversely affect the performance. A generic meta-data instruction can be used to concisely represent this check, so that when pointer dereferencing instruction is executed a sequence of micro-ops to perform additional checks will be automatically generated.

- Flexible Meta Data Representation and Efficient Cache Usage - As we noted in the prior section, object meta-data layout is efficient in terms of performance but for some checks like dangling pointer checks we also need pointer meta-data. So having the flexibility to associate the meta-data as either PMD or OMD (wherever appropriate) would be important for adding customized instructions for efficiently executing safety checks.

5.2. Overview of Meta Data Check Architecture Extensions

We propose extending the ISA with a special instruction called the *meta-check* instruction to perform the memory safety checks. The meta-checks are bound to a virtual register, which at compile time is determined to hold a pointer. The virtual register for a meta-check is explicitly represented in the meta-check instruction. When that register is used (dereferenced) by a load or store memory operation, the meta-check micro-op instructions are inserted into the execution stream to perform the check. These meta-check micro-ops are inserted before the memory operation. Thus, check operations need not be explicitly specified for each pointer dereferencing memory operation, reducing register spill and pressure on fetch bandwidth.

A sequence of meta-check instructions is used perform a bounds check and/or a dangling pointer check. One can view each meta-check as an assertion or a rule that a pointer value in the register must obey. The meta-check instruction is coded with few values - here we will briefly explain the important fields. One field specifies the type of the check operation (eg: less than, greater than, equal to etc – operations using which the compiler can perform required safety checks) and another field specifies the virtual register that needs to be associated with that check. The check operation also needs the meta-data to compare against the pointer value in the register. Hence, each meta-check instruction also specifies the sources for meta-data, which can be a PMD or an OMD or another virtual register.

When a meta-check instruction is executed, `MDCT` Meta-Data Check Table is updated. MDCT is a finite sized buffer to hold the information needed to later perform the meta-data checks associated with a given virtual register. Each meta-check instruction is assigned an entry in the MDCT. For a given virtual register, the compiler can use multiple meta-check instructions to associate more than one check with the virtual register. While executing a memory operation, during the register renaming stage, the MDCT is accessed to determine the checks that need to be performed for the register used by the memory operation. Then required micro-op instructions are inserted into the pipeline which will automatically load the required meta-data into the physical registers and execute the check operations.

The micro-op expansion for the meta-check instruction could be supported by techniques such as DISE [17]. By generating the micro-ops to perform the safety check when the virtual register holding the pointer is used, we avoid the need to explicitly insert those checks in the binary. As a result, we also avoid register spill as we don't have to use virtual registers in the binary to hold the meta-check's temporary values. The format and the implementation of the meta-check instruction is flexible enough to support different types of meta-data layouts, and also potentially many different types of checks.

The remainder of this section will describe the format of the meta-check instruction, few sample checks that can be performed with the meta-check instruction, and the hardware extensions required to support this instruction.

5.3. Meta-Check Instruction

A meta-check instruction binds a check operation to a virtual register, which will be executed whenever a memory operation uses the register. The format of the meta-check instruction we modeled is shown below.

```
meta-check  ptr_reg, slot, offset(ptr_base),  meta-operand-1, meta-operand-2, cond
```

At a high level, `ptr_reg` is the register containing the pointer over which safety check need to be performed. `offset(ptr_base)` specifies the address where we can find the pointer meta-data. The meta-operand field in the instruction is a bit mask that specifies which field in the meta-data needs to be used for the check (there are two masks for two operands). `cond` is the check operation to be executed. Here is a more detailed definition of the fields for the instruction:

- `ptr_reg` - is the virtual register that contains the pointer value that the compiler wants to associate the check with. It is assumed that the register will contain the pointer before executing the meta-check instruction.
- `slot` - The compiler should be allowed to bind more than one check with a particular virtual register (bounds check requires two meta-check operations, dangling pointer check requires one, and to do both we need three meta-checks to be associated with the pointer register). To keep track of the checks associated with a virtual register, we use a table called MDCT (explained later in Section 5.5.). To bound the size of the MDCT, we must limit the number of meta-checks associated with each virtual register. For this study, we use a limit of four. The slot bits specify which of the four possible meta-checks is being defined by the instruction for the specified virtual ptr_reg.
- `offset(ptr_base)` - ptr_base is the register containing the address where the pointer (loaded into ptr_reg) is located in memory. An offset from this ptr_base address, is where we can find the pointer meta-data in memory. The pointer meta-data can contain all the necessary meta-data or can contain a link to the object meta-data (refer Figure 1). On executing the meta-check instruction, a physical register will be allocated and ptr_base plus offset will be saved to into it (this value is referred to as `MD_base`, which is the effective address required to access the pointer meta-data) .
- `meta-operand-1 and meta-operand-2` - There can be multiple fields in the pointer meta-data or object meta-data. For example, for bounds checking, we need two fields (one for low bounds and another for high bounds) in the meta-data. These specify where to find the two source operands required to perform the check operation. These could take one of the following type:

  ```
  O(OMD_Mask)|P(PMD_Mask)|ptr|const
  ```

These formats mean:

  - `N-bit PMD_Mask` - It is possible that a pointer has many meta-data associated with it. This field indicates which pointer meta-data word(s) should be used as operand(s) for the meta-data check specified by the instruction. This could be implemented as an offset instead of a Mask.
  - `N-bit OMD_Mask` - Similar to the above PMD_Mask, the OMD_Mask indicates which object meta-data word(s) should be used in this meta-data check. This could be implemented as an offset instead of a Mask.
  - ptr - this specifies to use the value in the pointer register that triggered the check as an operand for the meta-check (this is the ptr_reg that the instruction associates the check with. It is already specified in the meta-check encoding but this field here specifies whether to use it as an operand for the meta-check or not).
  - const - A small N-bit constant.

- `cond` - this determines the type of check to perform using the meta-data. The supported traditional types of checks could be: EQ, NEQ, GT, GTE, LT, and LTE.

The meta-check instruction binds a check operation (comparison of two meta-operands using the condition specified) to the specified virtual register ptr_reg. The two meta-operands could be both meta-data, or the comparison could be between one meta-data and the value in `ptr_reg`. They also could be both from the PMD or both from the OMD. The order in which the expression is evaluated is from left to right in terms of the `ptr_reg` and meta-data words specified in the PMD and OMD being compared.

The execution of meta-check instruction results in two updates. First, the meta-check instruction allocates a physical register and saves the meta-data base address value called the `MD_base` in the physical register. The `MD_base` is computed from the ptr_base by adding to it a fixed one pointer-word offset- (`offset(ptr_base)`). This `MD_base` pointer contains the address of the 1st word of the pointer meta-data. From base pointer we obtain all PMD addresses by adding the PMD_mask offset. If OMD is being used, 1st word of the pointer meta-data will be the link pointer to object meta-data. From the link pointer we can obtain all the OMD by adding OMD_mask offset.

Second, the required information for executing the check operation is stored in the MDCT, so that later on, when a memory operation access the virtual register the check operation can be automatically inserted into the pipeline (MDCT and other extensions to the pipeline to execute the meta-check instruction will be described later in Section 5.5.).

The reason for going with the above fairly generic meta-check instruction description is to not make an assumption about where data is located in the PMD and OMD for the type of checks that might want to be performed. The only assumption is that when there is a link, the first word of the PMD is the link to the OMD.

5.4. Using the Meta-Check Instruction

To better understand the meta-check instruction, lets look at using the meta-check instruction for performing bounds checking and dangling pointer checks. In the example below,

(1) corresponds to the dangling pointer check in Figure 1(d), (2) corresponds to the PMD bounds checking in Figure 1(b), (3) corresponds to the OMD layout of bounds checking in Figure 1(c), and (4) corresponds to performing both OMD bounds checking and the dangling pointer check on the same pointer. In this last case, the object tag is the third word of the object meta-data.

```
-------------------------------------------------------------------------------
(1) Dangling Pointer Check: Compare second PMD field with first OMD field

    meta-check  ptr_reg, 00, off(ptr_base),  P(0100), O(1000), NEQ
-------------------------------------------------------------------------------
(2) Bounds Check using Pointer Meta Data (PMD)

    meta-check  ptr_reg, 00, off(ptr_base),  P(1000), ptr, GT // Check Lower Bound
    meta-check  ptr_reg, 01, off(ptr_base),  P(0100), ptr, LT // Check Upper Bound
-------------------------------------------------------------------------------
(3) Bounds Check using Object Meta Data (OMD)

    meta-check  ptr_reg, 00, off(ptr_base),  O(1000), ptr, GT // Check Lower Bound
    meta-check  ptr_reg, 01, off(ptr_base),  O(0100), ptr, LT // Check Upper Bound
-------------------------------------------------------------------------------
(4) Combining Bounds Check using OMD and Dangling Pointer Check

    meta-check  ptr_reg, 00, off(ptr_base),  O(1000), ptr, GT // Check Lower Bound
    meta-check  ptr_reg, 01, off(ptr_base),  O(0100), ptr, LT // Check Upper Bound
    meta-check  ptr_reg, 10, off(ptr_base),  O(0010), P(0100), NEQ
-------------------------------------------------------------------------------
```

Figure 6: Example meta-check instructions for dangling pointer and bounds checking. `P` stands for meta-check data stored as the PMD, and `O` stands for meta-check data stored as the OMD.

In Figure 6(1), the first meta-check instruction is for specifying a dangling pointer check. As explained in Section 4.2.2., to perform a dangling pointer check, the tag stored in the pointer PMD is compared against the tag stored in the OMD. Because it uses OMD, the first word after the pointer (in the PMD) is the pointer to the OMD. The second word in the PMD is the pointer tag and the first word in OMD is the object tag. These are specified by the bit masks `PMD_mask` and `OMD_mask`. These source operands for the NEQ check operation will cause a trap if they are not equal. Note, the example shows just 4-bits for the masks, but the masks can be longer based on how many bits are available in the instruction encoding. In addition, to allow access to larger meta-data structures, an offset into the meta-data could be used instead of a mask.

Figure 6(2) shows using the meta-check instructions for bounds checking using the layout where the bounds information is stored in the PMD, as shown in the Figure 1(b). One check instruction is for comparing the `ptr_reg` address, when it is used in a later instruction, with the lower bound stored in the first word of PMD and the other one compares the `ptr_reg` address with the higher bound stored in the second word of PMD.

After the meta-check is registered for a given `ptr_reg`, any instruction that uses that register (before it is redefined) for an address calculation has the corresponding checks inserted into the instruction stream. The architecture to support this is described later. The checks are inserted directly after the address generation and before any remaining operations for that instruction. If a virtual register has associated with it multiple meta-checks, as in bounds checking, the architecture inserts the micro-op checks based on their instruction slot number. In addition, all of the meta-checks assigned to the same virtual register must specify the same base register, since the value of the base register is only stored once in the architecture.

To give an example of how the checks are inserted automatically into the instruction stream, assume we insert the meta-check instructions in Figure 6 (2) into the binary after a load of a pointer to virtual register `r1`. Then before `r1` is redefined, we see a use of it in the instruction `sub offset(r1), immediate`. Below is the micro-op sequence generated for the x86 subtract instruction along with the two meta-checks for bounds checking and their meta-data access loads that are inserted right after the address generation.

```
// Original x86 instruction
sub offset(r1), immediate

// micro-op expended of subtract
1.  agen  tmpAddrReg = r1 + offset    // address generation
2.  agen  lowaddr  = P(1000)+ MD_base // meta-check: compute low bound address
3.  load  low  = M[lowaddr]           // meta-check: load low bound from PMD base
4.  cmp_gt_trap low, tmpAddrReg       // meta-check: compare low bound
5.  agen  highaddr = P(0100)+ MD_base  // meta-check: compute high bound address
6.  load  high = M[highaddr]          // meta-check: load high bound from PMD base
7.  cmp_lt_trap high, tmpAddrReg      // meta-check: compare high bound
8.  load  tmpReg = M[tmpAddrReg]      // load real data
9.  sub   tmpReg = temReg - immediate // perform the subtract
10. store M[tmpAddrReg] = tmpReg      // store the result
```

One advantage of doing the above, is that if a trap occurs, it will be caught before the store commits and the PC that will be marked as having the exception is the store. This allows an exception handler or debugger to know exactly the instruction that violated the safety check. In comparison, when a `bound` instruction is used, the PC of the bound instruction would be marked as having the exception.

5.5. Hardware Support for Meta-Check Instruction

Meta-check instructions are buffered in the Meta-Data Check Table as noted above. Currently we assume its capacity to be four entries for each virtual register (which we believe is sufficient to perform a variety of checks like the ones discussed in this paper). Therefore, for x86, we need thirty-two entries in MDCT (eight times four). When a meta-check instruction is decoded, it populates an entry in the MDCT table. If it is the first meta-check assignment for the virtual register, a physical register is allocated. The base pointer (MD_base) to the PMD meta-data is computed (from the ptr_reg and offset specified in the meta-check instruction) and stored in the physical register. The mapping between the physical register holding MD_base and the virtual register to which the meta-check is associated in stored in the *Meta-Data Base Register Map* (MDBRM) shown in Table 3. MDBRM has an entry for each virtual register (eight entries in the case of x86) keeping track of the base address of the pointer meta-data. Both the MDCT and MDBRM can be directly written and read so as to enable context switching.

On executing a memory operation, the MDCT is consulted to determine if check operations are bound to the virtual register that is used for effective address computation (we will refer to this register as pointer-register). If so, check instructions corresponding to that pointer-register are micro-op expanded and inserted into the pipeline before executing the memory operation itself. The MDCT, shown in the Table 2, contains the following fields. The first field holds the virtual register that will hold the pointer we want to check, the second is the slot identifier, the next two fields hold the first and the second meta-check operand bits, and the last field holds the condition to evaluate the check expression. The table is direct-mapped indexed first by the virtual register and then by the slot number. Similar to the register rename map, the MDCT keeps track of only the most recent definition for each virtual register.

Table 3 shows the physical register holding the base address of the pointer stored in a virtual register. When the first meta-check instruction is encountered for a virtual register, a physical register is allocated to hold the base address of the pointer. All meta-checks for a virtual register definition have the same pointer base address, so they all use this physical register, which is used to get access to the PMD (the first field in PMD contains the link to the OMD, if OMD is used) for the micro-op expanded checks.

Pointer Virtual reg	Slot	1st Operand reg	2nd Operand reg	Operation
r1	0	O(1000)	ptr	GT
r1	1	O(0100)	ptr	LT
r1	2	O(0010)	P(0100)	NEQ
r1	3			

Table 2: Meta-Data Check Table (MDCT).

Pointer Virtual reg	Physical Register containing MD_base
r1	p20
r2	p2
r3	-

Table 3: Meta-Data Register Map (MDBRM).

We now describe what happens when a meta-check is fetched, and when the pointer register we are watching is used for an address generation.

Expanding a Meta-Check Sequence- Take for example the three meta-check instructions in Figure 6(4). After executing those three meta-check instructions, the state of the MDCT table will be as shown in Table 2 and MDBRM table in Table 3. The virtual register $r1$ is the pointer register to be checked (the register into which the pointer would have been loaded). On executing the first meta-check instruction, a physical register $p20$ is allocated to address of the start of the PMD. This is the `MD_base`, and it is shared among all the meta-checks for the same virtual register $r1$.

Then the micro-code engine automatically inserts into the instruction stream instructions to perform the check comparisons in the table when there is a use of $r1$ for a memory reference. The micro-ops first load the meta-data using the value stored in the MDBRM along with the offsets specified in the meta-check operands stored in the MDCT.

If the micro-op expansion uses the OMD data, it first inserts an instruction to load the link register to the OMD. In this example it allocates $p8$ as the link register. Next expansion generates the load operations for the meta-data, using as the base register: for PMD meta-data, which is register $p20$, and for OMD meta-data is accessed via the link register $p8$. For the example in Figure 6(4), the following micro-ops would be inserted to perform the bounds and dangling pointer check. As described earlier for the store example, these checks will be inserted in the instruction stream between the address generation and the rest of the instruction's execution.

```
load  p8  =    [p20]
agen  p25 =    O(1000) + p8
load  p2  =    [p25]
cmp_gt_trap    p2,  p10
agen  p26 =    O(0100) + p8
load  p4  =    [p26]
cmp_lt_trap    p4,  p10
agen  p27 =    O(0010) + p8
load  p8  =    [p27]
agen  p28 =    P(0100) + p20
load  p5  =    [p28]
cmp_neq_trap   p16, p5
```

Freeing MDCT and MDBRM Table Entries and their Physical Registers - When a virtual register is redefined by an instruction, the MDCT and MDBRM entries corresponding to that register are removed, since the virtual register has been redefined.

But note that the physical base register allocated to those entries is not freed until the instruction that is redefining the virtual register commits. When a new register definition occurs, if there are hits in the MDCT, we (1) remove the entries from the MDCT, and (2) remove the base pointer register mapping from MDBRM. When this new instruction commits, we know that we can then free the base pointer physical register. This is similar to the conventional algorithm used to manage freeing physical registers in current architectures.

Even though multiple definitions of a virtual register can be alive at a time, the MDCT and MDBRM table needs to only hold the check instructions and base meta-data mapping corresponding to the latest definitions of the virtual registers. This is because decoding and renaming are done in-order, and the tables are used to just generate the micro operations in-order during the decode stage.

Branch Mispredictions, Context Switches and Exceptions - Branch mispeculations are handled in modern architectures by checkpointing the register rename table. To support our extensions, the physical register mapping of the MDCT and MDBRM is checkpointed as with any other renamed register set. Upon recovering from a misprediction, the check-point map is restored.

Context switching imposes additional burdens, as the MDCT and MDBRM state must be saved to software memory (kernel stack). We narrowly expose the MDCT/MDBRM architecture to enable efficient saving and restoring of state. For the MDCT, the saved state is the original meta-check opcode encoding. When we store an entry from the MDCT to memory, it recovers the original meta-check representation, which is stored in the MDCT. We also save and restore the *value* of the base pointer register from the MDBRM for that meta-check. Upon restoring the meta-check instruction along with the base value, we restore the meta-check into the MDCT table and allocate a new register in the MDBRM. As there are up to 32 meta-check instruction entries and eight base pointer entries, the context switcher checks if the register is used for meta-check instructions, spilling them only if necessary. We keep track of the use status in a bit vector indexed by virtual register number. Upon restore we walk through the bit vector, and reload the corresponding previously used MDCT table entries and the base register.

5.6. Link and Pointer Meta Data Compression

We observed that there are two pieces of information that typically can be compressed in the PMD. The upper bits of the link are usually the same as the pointer value. One can potentially provide a special version of the above meta-check instruction, so that the first two data items in the PMD are compressed into one word. The link would use 20-bits, and this leaves 12-bits to be used for something else. We examine using this combination for compressing the dangling PMD pointer tag and the link together. In doing this, the PMD in Figure 1(d) becomes only one word instead of two words. We examine the effects of this optimization in the results section.

6. Results for Bounds and Dangling Pointer Checks

In this section we will discuss the benefit of Meta-Data Checking (MDC) architecture. First we will discuss the results for doing just the bounds checking and then discuss results for doing both bounds and dangling pointer checks.

6.1. Performance of Bounds Checking

In the Section 4 we discussed the overheads of bounds checking implementations. There we did not assume any architectural support but instead implemented bounds checking using existing x86 assembly instructions. Those results are shown again in the Figure 7(a). The result labeled as *bnd-pmd* shows the overhead of bounds checking using PMD layout (shown in the Figure 1(b)) and the one labeled as *bnd-omd* shows the bounds checking overhead when we use OMD layout (shown in the Figure 1(c)). In addition, Figure 7(a) shows the overhead of bounds checking when we implement it using the meta-check instruction described in the previous section. This result is labeled as *bnd-omd-MDC*. For all the results we again break the execution time between fetch stall (fe), branch misprediction (brm), data cache misses (dc), overlapped data cache miss with execution (dc/ex), and execution (ex) where there were no stalls.

We see that the average overhead is 81% when the bounds are stored with the PMD but we incur only 48% overhead when the bounds are stored with the OMD. This improvement can be attributed to the improvement in data cache miss rates as we now share the bounds information for an object across all the pointers to that object.

Using the MDC architecture the average overhead of bounds checking is reduced significantly to 21%. These savings can be attributed to the reduction in time spent in execution (represented by `ex` and `dc/ex` in the Figure). Time spent due to `ex` and `dc/ex` is consistently reduced across all the benchmarks. Especially for programs like `bzip`, the performance improvement is significantly reduced from 43.7% to 8.3%.

For programs like `mcf`, we do not see appreciable gains. The reason is that `mcf` is memory bounded and a greater proportion of the execution time is spent servicing cache misses. The MDC architecture, though it optimizes the number of instructions fetched and executed, the overhead due to increased memory footprint to store the meta-data information still remains. But, note that the stalls due to data cache misses is significantly reduced in OMD layout (*bnd-omd*) as compared to PMD layout (*bnd-pmd*).

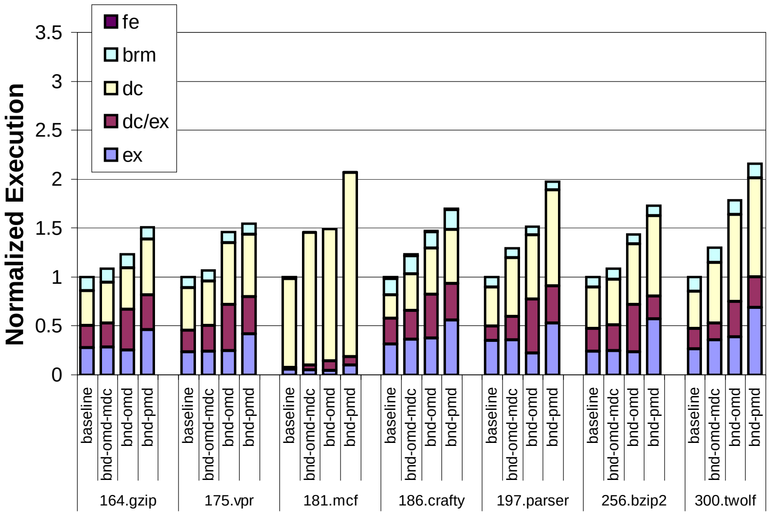
To summarize, our meta-data layout coupled with meta-check instruction reduce the average overhead of bounds checking to 21% slowdown which is a significant reduction when compared to 81% incurred by current software implementations when providing complete bounds checking.

6.2. Performance of Dangling Pointer Check

Figure 7(b) shows the overhead for performing dangling pointer checks on top of bounds checks. The two results from Figure 2(b), *bnd-pmd-dng* and *bnd-omd-dng* are reproduced here for comparison. The bounds check and the dangling pointer check are implemented for these two results using only x86 instructions.

Before discussing the results, here is a quick summary on how the meta-data is laid out. For *bnd-pmd-dng*, the bounds information is associated with PMD. There will be four PMD words: two for bounds, one for link address and another for pointer tag needed for the dangling check. In addition there will be one OMD word to hold the object tag needed for dangling pointer check. For the *bnd-omd-dng* results, bounds information is associated with OMD, which means there will be just two PMD words (one for link address and another for pointer tag) but three OMD words (high, low bounds and one more word for object tag).

The overhead of these implementations are pretty steep. The overhead for *bnd-pmd-dng* configuration is 148% which we expected as it uses four PMD words. Especially, since the dangling pointer check needs a link address it is definitely better to store bounds in OMD. When we do that, we see a significant reduction in the average overhead to 63.9% (corresponding to *bnd-omd-dng*).



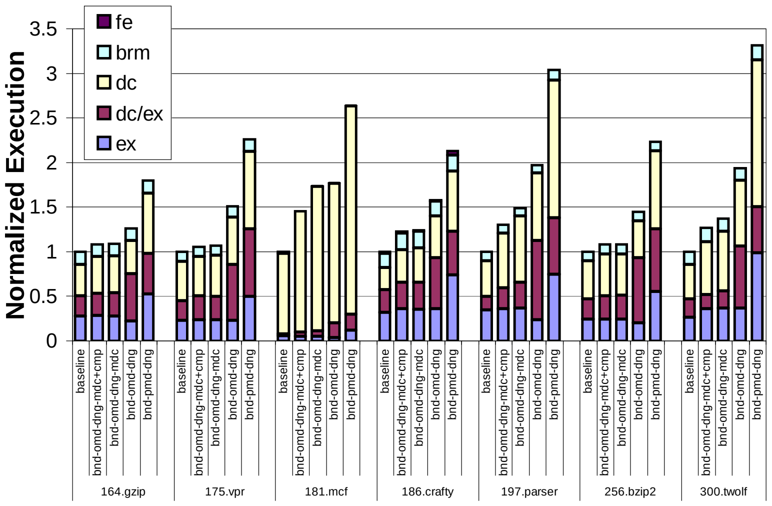


Figure 7: Normalized execution time for bounds checking (top graph) and dangling pointer check (bottom graph) using the meta-check instruction.

bnd-omd-dng-MDC in the Figure 2(b) corresponds to the implementation that assumes the MDC architecture. The average overhead reduces to 29.8% from 63.9% when we apply MDC architecture optimizations. We achieve this reduction in performance overhead by reducing the number of instructions inserted into the binary to perform the check. This can be noted by comparing the reduction in `ex` and `dc/ex` components.

Finally, as described in the Section 5.6., we can compress the link address and the pointer tag into one PMD word. The result corresponding to this optimization is labeled as *bnd-omd-dng-MDC+Comp*. This compression reduces the increase in memory footprint and as a result yields better cache performance. On average, the overhead reduces to 21.2%, which is only a slight increase in overhead for adding dangling pointer checks on top of bounds checking. This shows that our approach scales well and that as long as we can avoid increasing the PMD size we can keep the performance degradation within tolerable limits.

7. Related Work

In this paper we focus on providing comprehensive bounds checking and dangling check for every pointer reference. Complete bounds checking is required to guarantee security. In this section we will discuss the recent architectural and software proposals to assist bounds checking.

7.1. Hardware Support for Bounds Checking

Recently there has been significant interest in providing hardware support to assist debugging. Zhou et.al., proposed iWatcher [18] to monitor accesses to memory locations. The memory location that needs to be monitored and the monitoring function that needs to be executed when a monitored memory location is accessed are specified through a system call. A bit is associated with each word in the L1 and L2 caches, so that the hardware knows which locations need to be monitored (information will be lost when a block is evicted). A software table is used to map the addresses of monitored locations and the monitoring function corresponding to them. When there is an access to a monitored location, the software table is searched to access the monitoring function which is then executed. HeapMon [19] is another related work which proposed to use status word for each word in the heap to dynamically detect uninitialized or unallocated memory locations. Witchel et.al. proposed Mondrian Memory protection [20] to provide fine grained protection down to a word using hardware support, mediated by kernel.

The above proposals did not discuss and evaluate the performance overhead of checks like bounds checking and dangling pointer checks, which require us to keep track of meta-data information for pointers. In this paper, we discuss where to store the meta-data information and propose ISA extensions that allows us to access and use meta-data efficiently.

Lam and Chiueh [21] examine optimizing bounds checking by cleverly exploiting a feature in the x86 architecture that is used to protect segments of memory. One segment is used for each object and before dereferencing an object the segment registers are initialized with the base (lower bound) and the limit (specifies upper bound) of the segment corresponding to the object. When the object is dereferenced, the x86 architecture will verify if the pointer is within the bounds of the segment. Using the segment registers in this way allows for lower and upper bounds checking for objects up to 1MB, but not larger than that. Due to the overhead of setting up the segment registers with the bounds, they propose to limit the use of their technique to verify only the array references inside loops. In comparison, we propose a general approach for meta-check instructions that allow various additional checks to be done for pointers (eg: dangling pointer check). We apply our technique for protecting *all* the pointer references (not just the arrays) and hence it is useful for pointer intensive applications, and our approach works for objects (e.g., arrays) larger than 1MB.

Shao et al [22] examined having hardware instruction for bounds checking similar to the x86 bounds instruction. They propose using a special bounds check instruction to reduce the overheads of the software bounds check. One contribution of our work is that they only look at using pointer meta-data, whereas we examine both object and pointer meta-data.

The instruction they propose is almost identical to the x86 bound instruction, whereas we have proposed a flexible ISA architecture to handle other software checks such as dangling pointers. Also, the special bounds check instruction they use needs to load the bounds from memory for each reference of the object. In comparison, in our implementation bounds are held in registers which reduces the bounds check performance overhead.

In DISE [17], Corliss et.al. proposed a programming interface to the dynamic instruction macro-expansion found in modern processors. A sequence of functions (essentially micro-ops) are associated with an instruction and are dynamically injected into the pipeline when that instruction is executed. They applied their technique for achieving memory fault isolation, which ensures that the modules sharing the same address space are accessing within the data or code segment that they can legally access. In their follow up work, Corliss et.al. [23] used the DISE mechanism to efficiently implement watchpoints that will be useful for implementing interactive debuggers. DISE can be used to associate and execute additional checks like bounds checking and dangling pointer checks with instructions that dereference pointers. But previous work has not analyzed the performance overhead of using DISE like scheme for bounds checking and dangling pointer checks, which is analyzed in this paper. Also, to perform such checks we need mechanisms to track and access the meta-data efficiently, which are not addressed in the earlier works but are discussed in this paper.

7.2. Software Based Solutions

Austin et.al. [11] implemented bounds checking and dangling pointer checking by doing a source to source translation. For doing bounds checking, they tracked meta-data with pointers (PMD). To implement dangling pointer checks, they had a *capability table*, which holds capabilities of the objects (similar to object tags we used). Whenever an object is created, an unique capability is generated and inserted into the table and also stored along with the pointer to the object. When the pointer is referenced, they make sure that the capability table contains the capability stored along with the pointer. Searching through the capability table and using PMD for bounds checking could be expensive and they report an execution overhead in the range of 130% to 540%.

Patil and Fischer [15] provided bounds and dangling pointers checks using a second (shadow) processor running on a separate co-processor to accelerate checking. The original program runs ahead while a sliced checker process follows the main thread, synchronizing at system calls with a combined run-time overhead of 10%. Their solution involves source to source translation to create a completely different shadow process which needs to be executed concurrently on a different co-processor. The two processes need to be kept in synchronization to ensure that they are executing along the same path in the program. When compared to this approach, ours is very lightweight and requires less hardware resources.

We recently proposed compiler optimizations for reducing the performance overhead of bounds checks [24]. One of the optimizations involved pruning bounds checks for read operations as they are not vulnerable to write buffer overflow attacks. The bounds checked version of the binaries that we use in this work use compiler optimizations proposed in [24], but only those that provide complete bounds checking. In this work, we further reduce the bounds check overhead using hardware support. The hardware optimizations proposed in this work are complimentary to the compiler optimizations proposed in [24]. We also consider the performance trade-offs at the micro-architectural level for placing the meta data along with the pointer versus the object. In addition, the optimizations proposed here are more generic, in that they are applicable for reducing the overhead of other safety checks such as dangling pointer checks.

8. Conclusion

Automatic run-time pointer checking can detect memory bugs, provide security, and help software developers find memory bugs efficiently. As programs get ever larger, and the cost of bugs in dollars and security adversaries becomes painfully expensive, these techniques become increasingly important.

Computer architecture needs to play a role in lowering the overhead of these software checks. The meta-data checks we examine in this paper are bounds checking and dangling pointer software checks. We provided a detailed analysis of the trade-offs for where to store the meta-data, with the pointer or with the object. The results show that storing the meta-data with the object instead of the pointer provides better results, especially for programs like `mcf` and `parser` where there are many more pointers stored in memory than objects (each object has several pointers). In addition, as many more different checks are done on a pointer, storing the required meta-data with the object scales better in terms of performance. Incorporating both bounds and dangling pointer checks using this approach results in an average slowdown of 63.9%.

This slowdown is still too large for the checks to be used in released software. We therefore propose an ISA and architecture extension using the meta-check instruction. The meta-check loads the bounds and stores them into physical registers, and associates with a pointer register a set of micro-ops to be inserted to perform the dynamic check whenever that register is used to generate an address. This resulted in an average slowdown of 21.2%.

Acknowledgements

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant CNS-0509546 and grants from Microsoft and Intel.

References

[1] J. Gray, "Distributed computing economics," Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[2] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (roc): Motivation, definition, techniques and case studies," Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkely, March 2002.

[3] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability," in *21st International Symposium on Fault Tolerant Computing*, (Montreal), 1991.

[4] G. McGary, "Bounds Checking in C and C++ using Bounded Pointers," 2000. http://gcc.gnu.org/projects/bp/main.html.

[5] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the 10th Network and Distributed System Security Symposium*, pp. 149–162, February 2003.

[6] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Z. S. Midkiff, and J. Torrellas, "Accmon: Automatically detecting memory-related bugs via program counter-based invariants," in *37st International Symposium on Microarchitecture*, Nov. 2004.

[7] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[8] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder, "Automatically characterizing large-scale program behavior.," in *Proceedings of the International Conference on 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[9] O. Ruwase and M. Lam, "A practical dyanmic buffer overflow detector," in *11th Annual Network and Distributed Security Symposium (NDSS 2004)*, (San Diego, California), pp. 159–169, February 2004.

[10] Avantgarde, "Time to live on the network." http://www.avantgarde.com/xxxxttln.pdf.

[11] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Symposium on Programming Language Design and Implementation*, pp. 290–301, June 1994.

[12] D. Stutz, T. Neward, and G. Shilling, *Shared Source CLI Essentials*, ch. Managing Memory Within the Execution Engine. Sebastopol, CA: O'Reilly, 2003.

[13] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," 2002.

[14] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Symposium on Principles of Programming Languages*, pp. 128–139, 2002.

[15] M. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Software - Practice and Experience*, vol. 27, Jan. 1997.

[16] R. Jones and P. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Automated and Algorithmic Debugging*, pp. 13–26, 1997.

[17] M. Corliss, E. Lewis, and A. Roth, "Dise: A programmable macro engine for customizing applications," in *30th Annual International Symposium on Computer Architecture(ISCA-30)*, (San Diego, CA), June 2003.

[18] P. Zhou, F. Qing, W. Liu, Y. Zhou, and J. Torrellas, “iwatcher: Efficient architecture support for software debugging.,” in *31st annual International Symposium on Computer Architecture (ISCA'04)*, June 2004.

[19] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic, “Heapmon: a low overhead, automatic, and programmable memory bug detector,” in *Proceedings of the First IBM PAC2 Conference*, Oct. 2003.

[20] E. Witchel, J. Cates, and K. Asanović, “Mondrian memory protection,” in *Proceedings of ASPLOS-X*, Oct 2002.

[21] L. Lam and T. Chiueh, “Checking Array Bound Violation Using Segmentation Hardware,” *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp. 388–397, 2005.

[22] Z. Shao, C. Xue, Q. Zhuge, E. Sha, B. Xiao, H. Hom, and H. Kowloon, “Efficient Array & Pointer Bound Checking Against Buffer Overflow Attacks via Hardware/Software,” *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)-Volume I-Volume 01*, pp. 780–785, 2005.

[23] M. Corliss, E. Lewis, and A. Roth, “Low-overhead debugging via flexible dynamic instrumentation,” in *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture (HPCA-05)*, (San Francisco, CA), Feb. 2005.

[24] W. Chuang, S. Narayanasamy, and B. Calder, “Bounds checking with taint-based analysis,” in *International Conference on High Performance Embedded Architectures & Compilers*, 2007.