How to Shadow Every Byte of Memory Used by a Program

Nicholas Nethercote
National ICT Australia, Melbourne, Australia
njn@csse.unimelb.edu.au

Julian Seward
OpenWorks LLP, Cambridge, UK
julian@open-works.co.uk

Abstract

Several existing dynamic binary analysis tools use *shadow memory*—they shadow, in software, every byte of memory used by a program with another value that says something about it. Shadow memory is difficult to implement both efficiently and robustly. Nonetheless, existing shadow memory implementations have not been studied in detail. This is unfortunate, because shadow memory is powerful—for example, some of the existing tools that use it detect critical errors such as bad memory accesses, data races, and uses of uninitialised or untrusted data.

In this paper we describe the implementation of shadow memory in Memcheck, a popular memory checker built with Valgrind, a dynamic binary instrumentation framework. This implementation has several novel features that make it efficient: carefully chosen data structures and operations result in a mean slow-down factor of only 22.2 and moderate memory usage. This may sound slow, but we show it is 8.9 times faster and 8.5 times smaller on average than a naive implementation, and shadow memory operations account for only about half of Memcheck's execution time. Equally importantly, unlike some tools, Memcheck's shadow memory implementation is robust: it is used on Linux by thousands of programmers on sizeable programs such as Mozilla and OpenOffice, and is suited to almost any memory configuration.

This is the first detailed description of a robust shadow memory implementation, and the first detailed experimental evaluation of any shadow memory implementation. The ideas within are applicable to any shadow memory tool built with any instrumentation framework.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids, monitors; E.1 [*Data Structures*]

General Terms Design, Reliability, Performance, Experimentation

Keywords Shadow memory, Valgrind, Memcheck, dynamic binary instrumentation, dynamic binary analysis

1. Introduction

This paper describes how to create dynamic analysis tools that use *shadow memory*—tools that shadow every byte of memory used by a program with another value, in software—that are both efficient and robust.

VEE 2007 June 13–15, San Diego, California, USA.

1.1 What is Shadow Memory?

Programming tools such as profilers and checkers make programming easier and improve software quality. *Dynamic binary analysis* (DBA) tools are one class of such tools. They analyse a *client program* at the level of machine code as it runs. They can be built from scratch, but nowadays are usually implemented using a *dynamic binary instrumentation* (DBI) framework such as Pin [9] or Valgrind [16].

This paper focuses on a class of DBA tools that use *shadow memory*, i.e. they shadow, in software, every byte of memory used by a program with a *shadow memory value* that says something about it. We call these tools *shadow memory tools*. A shadow memory value may describe the value within a memory location (e.g. is it from a trusted source?), or it may describe the memory location itself (e.g. how many times has it been accessed?).

The *analysis code* added by the tool updates the shadow memory in response to memory accesses, and uses the shadow memory to report information to the programmer. The granularity of the shadowing can vary, but usually every used memory byte or word has a shadow memory value, and each shadow memory value may itself be one bit, a few bits, one byte, or one word, for example.

Some tools that use shadow memory also shadow every register with an extra value. Shadow registers are challenging to implement in their own right [16] but their implementation details are beyond the scope of this paper.

1.2 Shadow Memory is Useful

Shadow memory lets a tool remember something about the history of every memory location and/or value in memory. Consider the following motivating list of shadow memory tools; the descriptions are brief but demonstrate that shadow memory (a) is powerful, and (b) can be used in a wide variety of ways.

Memcheck [21, 12] is a memory checker. It remembers what allocation/deallocation operations have affected each memory location, and can thus detect accesses of unaddressable memory. It also remembers which values are undefined (uninitialised or derived from undefined values) and can therefore detect dangerous uses of undefined values. *Purify* [6] is a similar tool.

TaintCheck [17] is a security tool. It remembers which values are from untrusted (tainted) sources and which values were subsequently derived from them, and can thus detect dangerous uses of tainted values. *TaintTrace* [4] and LIFT [18] are similar tools.

Eraser [20] is a data race detector. It remembers which locks are held when each memory location is accessed, and can thus detect when a memory location is accessed without a consistent lock-set, which may imply a data race. VisualThreads [5] and Helgrind [10] are similar. DRD [19] is another race detector that uses a different race-detection algorithm.

Hobbes [3] is a run-time type checker. It remembers what operations have been performed on each value, and can thus detect any later operations that are inappropriate for a value of that type.

Annelid [13] is a bounds checker. It remembers which values are array pointers, and can thus detect bounds errors.

Redux [14] is a dataflow visualisation tool. It remembers which operation created each value, and its inputs, and records these in a dynamic dataflow graph which can be viewed at program termination.

pinSEL [11] automatically extracts system call side-effects from benchmarks so that architectural simulators do not have to emulate system calls when running those benchmarks. It shadows each memory location with a copy of itself, and does a "memory diff" between original and shadow memory values after each system call executes in order to determine how the system call affected memory.

All of these tools rely crucially on shadow memory. Eraser, DRD and pinSEL use shadow memory but not shadow registers, the others use both shadow memory and shadow registers. The shadow memory is implemented entirely in software and so these tools run on stock hardware.

1.3 Shadow Memory is Hard to Implement Well

Speed. The speed of a shadow memory implementation is important. Although programmers will use slow tools if the benefits are high enough, they prefer fast tools.

Shadow memory is inherently expensive. Large amounts of extra state must be maintained; one shadow byte per byte of live original memory is typical. Most or all loads and stores must be instrumented to keep the shadow memory state up-to-date, as must operations that affect large regions of memory, such as allocations and deallocations (on the heap, stack or via system calls such as `mmap`), reads/writes of large areas by system calls, and the loading of the program image into memory at start-up.

These requirements unavoidably increase the total amount of code that is run, increase a program's memory footprint, and degrade the locality of its memory accesses. Shadow memory tools thus typically slow down programs by a factor of 10–100, and shadow memory operations cause much more of this slow-down than other well-studied aspects of tools built with DBI frameworks such as hot trace formation [1] and code cache management [7].[1]

Robustness. The robustness of a shadow memory implementation is also important, arguably even more so than its speed. Real-world tools must cope with large, uncooperative programs, and if they are to be portable, cannot rely on particular operating system characteristics such as memory layouts.

Shadow memory is hard to implement robustly. The shadow memory must be squeezed into the address space alongside the original memory in a way that does not conflict with it and does not change the program's behaviour. This requires considerable flexibility in the shadow memory structure and layout. It also unavoidably reduces the amount of address space a program can use itself, which is an important issue on embedded platforms and even 32-bit machines. Obviously, this becomes less of an issue if shadow memory can be made smaller, so compact shadow memory is desirable.

Trade-offs. In summary, we want shadow memory to be: (a) fast (for efficiency); (b) structured flexibly (for robustness); (c) compact (for efficiency and robustness). Not surprisingly, these desires conflict and we will see that trade-offs must be made.

1.4 Contributions

In this paper we make four contributions. The first three arise because we delve more deeply into the topic of shadow memory than previous publications have.

- ***First detailed description of Memcheck's shadow memory.*** Memcheck is a widely-used tool, and this is the first detailed description of its shadow memory implementation. Previous publications [21, 12, 16] have discussed in detail every significant aspect of Memcheck except its shadow memory implementation.
- ***First detailed description of any robust shadow memory implementation.*** This is also the first detailed description of any robust shadow memory implementation. This is more important than it may seem, because shadow memory is a topic where *details matter*. High-level descriptions are not sufficient; lower-level implementation details such a data representations are crucial, as they make the difference between a toy and a real-world tool. Most published descriptions of shadow memory implementations have been minimal, and the three that have been discussed in detail are not robust enough, in our opinion, for use in a widely used tool like Memcheck.
- ***First experimental evaluation of shadow memory.*** This is the first paper that has evaluated and compared multiple versions of a shadow memory implementation.

The fourth and final contribution advances the state-of-the-art in shadow memory implementations.

- ***Novel shadow memory optimisations.*** Memcheck's basic shadow memory data structure is similar to that used in several other shadow memory tools. However, Memcheck adds several novel optimisations that speed up common cases, and compress shadow memory at coarse-grained (per-64KB chunk) and fine-grained (per-byte) levels. Together they reduce Memcheck's mean slow-down factor by 4.0–13.6x and shrink its mean shadow memory size by a factor of 4.5–213.4 over a naive implementation. The reduction in shadow memory size also improves robustness because it allows programs with larger memory footprints to be run in the same amount of address space.

Although the paper is centred around the implementation of Memcheck, the ideas within are general and apply to any shadow memory tool implemented with any framework.

1.5 Paper Structure

This introduction has discussed several aspects of shadow memory: (a) what it is, (b) that it is useful and used in several existing tools, (c) that it is difficult to implement well, and (d) that it has not been studied closely. Section 2 introduces Memcheck. Section 3 describes the most basic form of Memcheck's shadow memory implementation, and Section 4 describes the optimisations that improve its performance to an acceptable level. Section 5 evaluates Memcheck's robustness, speed and memory usage. Finally, Section 6 discusses related work and Section 7 describes future work and concludes.

2. Overview of Memcheck

Memcheck is our example tool. It is built using Valgrind. It is a good example because (a) it stores two kinds of shadow memory values, and so is a challenging example, and (b) it is probably the most widely-used shadow memory tool.

2.1 Valgrind

Valgrind [16, 12, 15] is a DBI framework designed for building heavyweight tools. Tools such as Memcheck are written, in C, as plug-ins to Valgrind's *core*, which handles the details of running the client program. The core translates machine code into a platform-

[1] These aspects are more important for lightweight tools [8] where the cost of analysis is small relative to the cost of running the original code.

neutral intermediate representation before instrumentation, which means tools are naturally platform-independent.

Valgrind tools are used by thousands of programmers (the Valgrind website [22] averages more than 1000 unique visitors per day) including the developers of many large projects such as Firefox, OpenOffice, KDE, GNOME, libstdc++, MySQL, Perl, Python, PHP, Samba, RenderMan and Unreal Tournament. It is a standard package on most Linux distributions. Valgrind is licensed under the GPL and is currently available for x86/Linux, AMD64/Linux, and PPC{32,64}/{Linux,AIX} [22]; experimental ports also exist for other platforms such as x86/FreeBSD and PPC32/Mac OS X.

2.2 Memcheck

Memcheck is a memory error detector designed primarily for use with C and C++ programs. It is the main reason for Valgrind's popularity–users surveys have indicate it accounts for more than 80% of Valgrind tool use.

When a client program is run under Memcheck, Memcheck instruments almost every operation and issues messages about detected memory errors. Memcheck maintains three kinds of metadata about the running client.

- ***A bits.*** Every memory byte is shadowed with a single *A bit* ('A' is short for "addressability") which indicates if the client may legitimately access it. A 0 represents an unaddressable byte, a 1 represents an addressable byte. They are updated as memory is allocated and freed, and checked on every memory access. With the A bits Memcheck can detect uses of unaddressable memory such as heap buffer overflows and wild reads and writes.
- ***V bits.*** Every register and memory byte is shadowed with eight *V bits* ('V' for "validity") which indicate if the value bits are defined (i.e. initialised, or derived from other defined values). A 0 represents a defined bit, a 1 represents an undefined bit.[2] Every value-writing operation is shadowed with another operation that updates the corresponding shadow values. With the V bits Memcheck can detect dangerous uses of undefined values with bit-precision [21]. In this paper we are only concerned with V bits for memory, not V bits for registers.
- ***Heap blocks.*** Memcheck records the location of every live heap block in a hash table. With this information it can detect bad or repeated frees of heap blocks, and memory leaks.

Conceptually, every register byte has eight shadow bits (the V bits), and every memory byte has nine shadow bits (eight V bits and one A bit). This implies that each shadow memory byte has 512 possible states. However, a byte's V bits are only consulted if the A bit says it is addressable, so there are 257 meaningful states for each byte (one "unaddressable" state, and 256 "addressable" states with different definedness sub-states).

Representing the definedness of every bit is potentially expensive, but crucial for accuracy. Tracking definedness at the byte level—a commonly-suggested alternative—inevitably causes false positives and/or false negatives, particularly for programs that use bit-fields and bit-level operations [21]. We have heard from several users that Memcheck has identified bugs caused by the use of a single undefined bit.

Fortunately, the redundancy in the number of states and the high cost of bit-level definedness tracking can be minimised with the use of compressed shadow memory (Section 4.4).

The next two sections present Memcheck's implementation of shadow memory, i.e. how it stores and accesses the V and A bits for memory locations. We do not discuss the use of V bits in shadow registers, nor the heap block metadata, as they have been covered by previous publications.

[2] This counter-intuitive encoding makes many of Memcheck's V bit shadow operations simpler [21] for reasons that are beyond the scope of this paper.

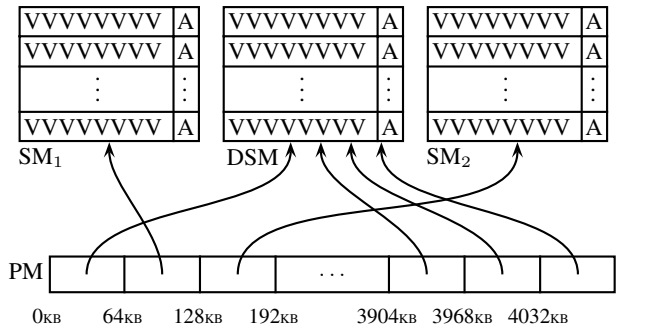

Figure 1. The two-level table. Entries `PM[1]` and `PM[2]` cover 64KB regions that have been written to and so have their own SM. The remaining PM entries still point to the `NOACCESS` DSM.

3. A Simple Implementation (M0)

This section presents a simple, robust, but slow implementation of shadow memory for Memcheck, which we call M0.

No released versions of Memcheck have used this implementation, but Memcheck has a debugging mode which falls back to it. Some or all of the optimisations described in Section 4 must be added to this implementation to obtain acceptable performance.

3.1 Shadow Memory Data Structures

Memcheck's main shadow memory data structure is a two-level table somewhat like a page table. It is designed for a 32-bit (4GB) address space; Section 3.8 describes how it is modified it for 64-bit address spaces. The address space is divided into 64K chunks of 64KB each. The *primary map* (PM) is a global array with 64K entries, one for each chunk. Each entry is a pointer to a *secondary map* (SM) which holds the shadow memory (A and V bits) for a 64KB chunk. (In the code below, the types `U1`, `U4`, `U8`, `U32` and `U64` are 1, 4, 8, 32 and 64 bit unsigned integers respectively, and `Uw`, `Addr` and `SizeT` are word-sized unsigned integers.)

```
typedef struct {    // Secondary Map: covers 64KB
  U8 abits8[8192];  //  8K A bytes == 64K A bits
  U8 vbits8[65536]; // 64K V bytes
} SM;

SM* PM[65536];      // Primary Map: covers 4GB
```

There is a *distinguished secondary map* (DSM), called the `NOACCESS` DSM, which is marked as entirely "unaddressable", and is never modified. All `PM` entries initially point to it. Figure 1 illustrates the relationship between the PM and the SMs.

Although each SM contains Memcheck-specific data, the basic two-level table can be used with any tool that uses shadow memory. Indeed, Section 6 shows that most existing shadow memory tools use a data structure similar to this.

3.2 Single-byte Loads and Stores

This section defines four fundamental functions used to load and store individual shadow memory bytes. Our functionally correct but slow implementation (M0) is built on top of these four functions.

There are two main steps when loading or storing a single shadow memory byte. In the first step, Memcheck uses the high 16 bits of the address to find the relevant SM within the PM. For loads, it can use the found SM as-is.

```
SM* get_SM_for_reading(Addr a)
{
  return PM[a >> 16];   // use bits [31..16] of 'a'
}
```

For stores, Memcheck uses copy-on-write semantics—it checks if the SM found is the DSM (with `is_DSM`), and if so, allocates and initialises a new SM (with `copy_for_writing`).

```
SM* get_SM_for_writing(Addr a)
{
  SM** sm_p = &PM[a >> 16]; // bits [31..16]
  if (is_DSM(*sm_p))
    *sm_p = copy_for_writing(*sm_p); // copy-on-write
  return *sm_p;
}
```

In the second step, Memcheck uses the low 16 bits of the address to find the A and V bits within the SM. Loads are done with the following function `get_abit_and_vbits8`. Extracting the V bits from the SM is straightforward; extracting the single A bit from a group of eight A bits requires extra shifting and masking.

```
void get_abit_and_vbits8(Addr a,/*OUT*/Uw* abit,
                                /*OUT*/Uw* vbits8)
{
  SM* sm    = get_SM_for_reading(a);
  U8 abits8 = sm->abits8[(a & 0xffff) >> 3];//[15..3]
  *abit     = 0x1 & (abits8 >> (a & 0x7)); //[ 2..0]
  *vbits8   = sm->vbits8[a & 0xffff];     //[15..0]
}
```

The store case (`set_abit_and_vbits8`) is similar.

```
void set_abit_and_vbits8(Addr a, Uw abit, Uw vbits8)
{
   SM* sm       = get_SM_for_writing(a);
   Uw shift     = a & 0x7;
   Uw i         = (a & 0xffff) >> 3;
   sm->abits[i] = (sm->abits[i] & ~(1 << shift))
                | ((abit & 0x1) << shift);
   sm->vbyte[a & 0xffff] = vbits8 & 0xff;
}
```

The following sections show other shadow memory operations that are layered on top of these four fundamental functions.

3.3 Multi-byte Loads and Stores

Most memory accesses are multi-byte. Every memory load in the client program is instrumented with a call to the following function `LOADVn`, which does a shadow load of 8, 16, 32 or 64 bits. It obtains each V byte individually from shadow memory and then combines them into a single n-bit value. The V bits of each byte are only used if the A bit indicates that the byte is addressable. If any unaddressable bytes are touched, Memcheck issues an error (with `record_address_error`) and acts as if the bits are all defined.[3] This avoids possible chains of multiple error messages caused by a single program defect [21]. (The `V_BITS*` and `A_BIT*` constants hold aggregations of one or more V or A bits. For example, `V_BITS8_DEFINED` represents eight defined V bits, i.e. the V bits for a fully defined byte.)

```
U64 LOADVn(Addr a, SizeT nBits)
{
  U1  abit;  U8 vbits8;  Int i;
  U64 vbits64 = V_BITS64_UNDEFINED;
  SizeT n_bad_addrs = 0;
  for (i = 0; i < nBits / 8; i++) {
    get_abit_and_vbits8(&abit, &vbits8, a + i);
    if (abit != A_BIT_ADDRESSABLE) {
      n_bad_addrs++;              // Defined-if-
      vbits8 = V_BITS8_DEFINED;  //  unaddressable
    }
    vbits64 = (vbits64 << 8) | vbits8;
  }
  if (n_bad_addrs > 0)
    record_address_error(a, nBits);
  return vbits64;
}
```

Note that this code assumes memory values are little-endian. Memcheck also handles big-endian values, but we omit the relevant details (which are minor) for clarity.

All client stores are instrumented with a call to `STOREVn`, which is similar to `LOADVn`; it copies a given shadow memory value of 8, 16, 32 or 64 bits into shadow memory, one byte at a time. If any destination byte's A bits indicate that it is unaddressable, the relevant V bits are not copied and an error message is issued.

3.4 Range-setting Operations

In certain cases, Memcheck needs to set every shadow memory byte in a range to one of the following three states.

1. `NOACCESS` (unaddressable): for memory deallocations (on the heap, stack, or via system calls such as `munmap`).
2. `UNDEFINED` (addressable and fully undefined): for memory allocations that do not initialise the allocated memory (such as `malloc`, `brk`, and stack allocations).
3. `DEFINED` (addressable and fully defined): for memory allocations that initialise the allocated memory (such as `calloc` and `mmap`), for memory loaded at program start-up, and for when a system call writes to a block of memory (e.g. `gettimeofday` fills in two structs with data).

Valgrind provides an event-tracking system that lets a tool know, via callbacks, when these operations occur [16]. Memcheck has three range-setting callback functions: `make_mem_noaccess`, `make_mem_undefined`, and `make_mem_defined`. They each take a starting address and a length in bytes, and set the shadow memory bytes for the given memory range, one byte at a time, using `set_abit_and_vbits8`.

A similar function, `copy_range`, is used to handle `realloc` and `mremap`. It copies A and V bits from one shadow memory region to another.

3.5 Range-checking Operations

Sometimes Memcheck needs to check A and V bits over ranges of shadow memory and issue error messages if they do not match what is required. This is mostly done to check that ranges of memory passed to system calls have the properties that they should.

Memcheck has three such operations: `check_mem_is_addressable` checks that every byte in a range about to be written by a system call is addressable (and thus safe to write); `check_mem_is_defined` checks that every byte in a range about to be read by a system call is both addressable and fully defined (and thus safe to read); and `check_mem_is_defined_asciiz` checks that every byte in a string of unknown length is safe to read.

3.6 Do Not Shadow the Shadows

There is a subtle problem that can occur in this shadow memory implementation. For example, consider this memory layout:

[3] All shadow memory tools must handle this case, i.e. provide a reasonable shadow memory value for memory locations that have not been allocated and are erroneously accessed.

SM_X	(a 72KB SM, which covers 64KB of address space)
Y	(4KB of client data)
SM_Z	(a 72KB SM, which covers 64KB of address space)

Any client accesses to Y causes shadow memory accesses to a secondary map SM_Y (it does not matter where SM_Y is located). But because SM_Y covers 64KB of address space, even in the best case it must cover at least 60KB's worth of address space from SM_X and/or SM_Z. In other words, because of the intermingling of SMs and client data, some parts of the SMs end up uselessly covering parts of the address space which are occupied by other SMs. (In Memcheck, those ranges would be marked as unaddressable by the client program.) This wastes space. If such intermingling is frequent, it can lead to a steep increase in the number of SMs needed. This problem affected early versions of Memcheck.

There are two ways to avoid this problem. If SMs are all nKB and they are guaranteed to be nKB-aligned, there will be no overlapping. But this can be too restrictive; for example, in Section 4.4 we introduce an SM that covers 64KB but is only 16KB in size. Memcheck uses a simpler approach: ensure that SMs are kept far away from the client's original data whenever possible.

3.7 Possible Corruption of Shadow Memory by the Client

It is possible for a buggy client to do wild writes that overwrite Memcheck's shadow memory (or any of its other data structures). When Valgrind only ran on x86/Linux, it used the x86 segment registers to prevent this. However, this non-portable feature was removed when Valgrind was ported to other architectures, and we know of no other way to prevent such wild writes without large slow-downs.

This problem rarely occurs in practice because Valgrind and Memcheck's data tends to be far away from client data, which minimises the chance of a wild write causing corruption. (This is another good reason why client data should not be closely intermingled with Valgrind and Memcheck data.) Also, Memcheck will always warn about any such wild writes by the client before they happen, because Valgrind and Memcheck data is marked as unaddressable via the `NOACCESS` DSM. (Other shadow memory tools will not be so lucky.) This is a good example of a trade-off: some robustness is sacrificed, albeit in rare cases, in favour of performance and portability.

3.8 Handling 64-Bit Machines

64-bit address spaces are much larger than 32-bit address spaces. The obvious extension is to use a three- or four-level table, but this would make every shadow memory access slower. Instead we extend the size of the primary map to 2^{19} bits (covering 32GB), and we add a slow, sparse auxiliary table for secondary maps higher than 32GB, and the Valgrind core avoids allocating above this 32GB point when possible. A below-32GB check has to be done for every shadow memory operation, but for the fast cases it can be combined for zero cost in a single mask-and-test operation with the alignment check used in the optimised shadow loads and stores (described in Section 4.1 below).

This approach works well under Linux because Valgrind has enough control over the address space layout that it can allocate most memory under the 32GB limit. Unfortunately, we have found that this is not true for PPC64/AIX. Therefore, Memcheck uses some "semi-fast" cases, similar to those described in Section 4, for certain accesses above the 32GB limit, e.g. those that are aligned and fully defined. This avoids the large slow-down that the slow cases cause, but is still approximately half the speed of the 32-bit version.

In the future, as 64-bit architectures become more common and memory footprints grow, this issue will become increasingly important. It is unclear how this shadow memory scheme can best be scaled to 64-bit address spaces, so this remains an open research question for the future.

3.9 Handling Multi-threaded Programs

Threads pose a particular challenge for shadow memory tools. The reason is that loads and stores become non-atomic: each load or store translates into the original load/store plus a shadow load/store.

There are two potential problems with this. First, asynchronous signals may be delivered between these two operations. To avoid this problem, Valgrind only delivers asynchronous signals to the client at particular safe points (between the code blocks that Valgrind's JIT compiler uses) [16].

Second, on a uni-processor machine, a thread switch might occur between these two operations. On a multi-processor machine, concurrent memory accesses to the same memory location may complete in a different order to their corresponding shadow memory accesses. It is unclear how to best deal with this, as a fine-grained locking approach would likely be slow.

To sidestep this second problem, Valgrind uses a thread locking mechanism; on a thread-switch the kernel still chooses which thread is to run, but Valgrind dictates when thread-switches occur and prevents more than one thread from running at a time. This works well on uni-processor machines—Valgrind can ensure that thread-switches never occur between a load/store and a shadow load/store, and Memcheck can simply ignore threading issues. On multi-processor machines it can also run multi-threaded programs safely, but only serially. As multi-processor machines become more popular, this shortcoming will become more critical. Whether the current approach will remain the optimal one in the future is an open research question.

4. A Better Implementation

This section presents four optimisations for M0 which when applied successively give us four new versions which we call M1–M4. These optimisations use standard principles: make common cases fast, and reduce storage requirements by exploiting redundancy in data. Together they reduce Memcheck's mean slow-down factor by 4.0–13.6x, and reduce its mean shadow memory size by 4.5–213.4x, making it fast enough for widespread use, and compact enough to handle large programs.

4.1 Faster Loads and Stores (M1)

Multi-byte loads and stores are very common, and we can do better than use `LOADVn` and `STOREVn` with them. For example, the following function does fast 32-bit shadow loads. If the load is aligned (which guarantees that all four bytes are covered by the same SM) and all four bytes are addressable, it obtains the 32 V bits from the SM in a single operation. Otherwise, it falls back to the slow, general case.

```
U32 LOADV32_fast(Addr a) {
  SM* sm;  U4 abits4;  U8 abits8;
  if (!IS_32BIT_ALIGNED(a))
    return (U32)LOADVn(a, 32);
  sm     = get_SM_for_reading(a);
  abits8 = sm->abits8[(a & 0xffff) >> 3];
  abits4 = (abits8 >> (a & 0x4)) & 0xf;
  return ( A_BITS4_ADDRESSABLE == abits4
         ? ((U32*)(sm->vbits8))[(a & 0xffff) >> 2]
         : (U32)LOADVn(a, 32) );
}
```

This function's fast path does one alignment test, two SM lookups, and one addressability check. This is much faster than `LOADVn` which does, for 32-bit loads, eight SM lookups, four addressability checks, and combines the four V bytes together. With

this function present, the slow case is run only every few thousand or even million loads.

`STOREV32_fast` is similar, but with the additional fast-path condition that `sm` must not be the DSM. The functions for fast 1, 2 and 8 byte shadow loads and stores are similar, except that the 1-byte case does not need the alignment check. These functions handle all the common multi-byte loads and store cases.

Section 5 shows that this optimisation reduces Memcheck's mean slow-down factor by 3.73x.

4.2 Faster Range-setting (M2)

Range-setting operations are also very common. There are three improvements that can be made to them. Section 5 shows that together these optimisations reduce Memcheck's mean slow-down factor by 1.62x, and reduce its mean shadow memory size by 1.97, but sometimes drastically more (e.g. more than 30x).

Vectorising *`set_range`***.** First, `set_range` from Section 3.4 can be vectorised so that it sets one byte at a time until the current address is N-aligned, then sets N bytes at a time, and finally sets any left-over bytes one at a time. N must be a power of two to ensure that all N bytes belong to the same SM. Memcheck uses $N = 8$.

This improvement is generally helpful, and also occasionally affects performance greatly by mitigating some nasty performance cases. Some operations that are very fast natively take far longer when run under Memcheck. For example, a large stack allocation takes a single instruction natively, but requires setting a large region of shadow memory under Memcheck. We have seen one example program that did a lot of 8KB stack allocations, and with the improved `set_range` it ran more than three times faster. This is a good reminder that it is important to not only make the common cases fast, but also make the uncommon cases not too slow.

Replacing whole SMs. Second, when setting a 64KB range covered by a single SM to "unaddressable", instead of laboriously marking every byte Memcheck can instead replace the existing SM with the `NOACCESS` DSM, achieving the same effect in a single operation (this can be viewed as vectorisation on a much larger scale). The replaced SM can then be deallocated. This greatly speeds up large deallocations, such as those done with `munmap` when unloading a shared object, and reduces shadow memory size as well.

Additional DSMs. The third improvement is more subtle, and was added to Memcheck more than three years after Memcheck was first released. It involves the introduction of additional `DEFINED` and `UNDEFINED` DSMs to complement the `NOACCESS` DSM. This turns out to be a big win: large code segments and read-only data regions can be covered by the `DEFINED` DSM, and because code segments are rarely written to, Memcheck avoids allocating many SMs. This saves memory and also improves speed because fewer SMs need to be initialised.

4.3 Faster Stack Pointer Updates (M3)

Stack pointer updates are very common, and the increment/decrement sizes are often small, statically known constants such as 4, 8, 12, 16 or 32 bytes. Memcheck uses specialised range-setting functions for these sizes which are faster than the variable-length range-setting functions. These functions first check that the stack pointer is aligned—it almost always is—and then operate like unrolled versions of the vectorised `set_range`. These operations are so common that this optimisation reduces Memcheck's mean slow-down factor by 1.28x, as Section 5 shows.

4.4 Compressed V Bits (M4)

This section describes a more elaborate optimisation—a low-level compression technique that on average further reduces the mean size of shadow memory by 4.29x and the mean slow-down factor by 1.16x, as Section 5 shows.

It is an optimisation that may seem obvious in hindsight. However, we are aware of no other shadow memory tool that uses compression like this, and Memcheck had been publically available for more than four years before we conceived and implemented it.

Indeed, before this optimisation was implemented, the Valgrind distribution included a cut-down version of Memcheck (called Addrcheck) which tracked A bits but not V bits (thus recording 1 bit of shadow information per byte, rather than 9), for use when Memcheck's memory overhead was too great. After adding this optimisation to Memcheck, we were able to kill off Addrcheck because the difference in memory usage (1 bit per byte versus slightly more than 2 bits) was so much smaller.

4.4.1 The Basic Idea

Memcheck's tracking of definedness at the level of individual bits is useful [21], but it is expensive considering that partially defined bytes (PDBs) are rarely involved in more than 0.1% of memory accesses, and are not present at all in many programs.

This situation can be improved. Instead of maintaining 8 V bits and 1 A bit for every used memory byte, we can instead use only two *VA bits* per memory byte. With two bits we can mark each memory byte with one of four states: `NOACCESS`, `DEFINED`, `UNDEFINED` or `PARTDEFINED`. The first three states are the familiar ones from Section 3.4. `PARTDEFINED` represents PDBs, which have their full eight V bits stored in a sparse *secondary V bit table*.

Shadow registers still have eight V bits per byte, so only shadow loads and stores are affected. Shadow loads uncompress the two VA bits for each memory byte into the eight V bits for each register byte, and shadow stores do the opposite. Loads and stores involving PDBs are much slower because they involve the secondary V bits table, which is an AVL tree. `copy_range` is also changed to copy entries from the secondary V bits table for any bytes that have the `PARTDEFINED` state.

This approach makes shadow memory much smaller. And although the PDB cases are slower, this approach is faster overall. This may be partly due to better cache behaviour but it is mostly because many shadow operations are simpler in the common case, as the next section shows.

4.4.2 The Details

Secondary maps now have the following structure.

```
typedef struct {
  U8 vabits8[16384];    // 64K two-bit values
} SM;
```

The first two of the four fundamental functions introduced in Section 3.2, `get_SM_for_reading` and `get_SM_for_writing`, are unchanged because the primary map is unchanged. The third fundamental function, `get_abit_and_vbits8` is replaced by the following function which loads the two VA bits for a memory byte.

```
U8 get_vabits2(Addr a)
{
   SM*   sm   = get_SM_for_reading(a);
   U8 vabits8 = sm->vabits8[(a & 0xffff) >> 2];
   vabits8 >>= ((a & 3) << 1);// shift 2 bits down
   return 0x3 & vabits8;      // mask out the rest
}
```

It is used by the following function which uncompresses the obtained VA bits into eight V bits, suitable for placing in a shadow register, and returns a boolean indicating if it was unaddressable. If the byte is a PDB, `get_sec_vbits8` is used to look up the V bits in the secondary V bits table.

```
Bool get_vbits8(Addr a, U8* vbits8)
{
   U8 vabits2 = get_vabits2(a);
   if        ( VA_BITS2_DEFINED   == vabits2 ) {
     *vbits8 = V_BITS8_DEFINED;
   } else if ( VA_BITS2_UNDEFINED == vabits2 ) {
     *vbits8 = V_BITS8_UNDEFINED;
   } else if ( VA_BITS2_NOACCESS  == vabits2 ) {
     *vbits8 = V_BITS8_DEFINED; // Defined-if-
     return False;              //  unaddressable
   } else {
     *vbits8 = get_sec_vbits8(a);
   }
   return True;
}
```

This function is in turn used by a new version of `LOADVn` which is very similar to the original one from Section 3.3.

The fourth fundamental function, `set_abit_and_vbits8`, is replaced by a new function `set_vabits2` similar to `get_vabits2`. A function `set_vabits8` (similar to `get_vabits8`) is built on top of it; it is used by the new `STOREVn`.

The fast-case shadow load function `LOADV32_fast` from Section 4.1 now has the following form.

```
U32 LOADV32_fast(Addr a) {
  SM* sm;  U8 vabits8;
  if (!IS_32BIT_ALIGNED(a))
    return (U32)LOADVn(a, 32);
  sm      = get_SM_for_reading(a);
  vabits8 = sm->vabits8[(a & 0xffff) >> 2];
  if      (VA_BITS8_DEFINED == vabits8)
    return V_BITS32_DEFINED;
  else if (VA_BITS8_UNDEFINED == vabits8)
    return V_BITS32_UNDEFINED;
  else
    return (U32)LOADVn(a, 32);
}
```

The alignment check and primary map look-up is the same as before. The secondary map look-up differs; the cases where the four bytes are entirely defined or entirely undefined are handled here, and the decompression of the two VA bits into eight V bits is straightforward. The remaining cases are handled by the slow `LOADVn` function, similar to before. In particular, if any of the bytes loaded are PDBs—i.e. they have the `PARTDEFINED` state—`LOADVn` looks up the secondary V bits table.

This `LOADV32_fast` is faster than the previous version in the fast case because (a) it gets the VA bits with one SM access instead of two; (b) it does not have to do any shifting and masking to extract the A bits; and (c) the number of conditional branches is usually unchanged—most loads are from `DEFINED` memory, so the second test (`VA_BITS8_DEFINED == vabits8`) usually succeeds. The other fast multi-byte load and store functions have similar benefits.

Finally, the new versions of `set_range` (from 4.2) and the stack operations (from 4.3) both benefit from the faster SM accesses.

4.4.3 The Secondary V Bits Table

The secondary V bits table is an AVL tree that holds the full V bits for PDBs in memory. It has three subtleties that require care.

- ***Stale nodes.*** When a PDB is overwritten with a non-PDB, we could remove its entry from the table, but checking for overwritten PDBs on every store would be slow and remove much of the benefit of compressed V bits. Instead we let these entries become *stale*. This does not affect correctness—the stale values are never read—but we need to garbage collect (GC) the table when it fills up to prevent space leaks. Memcheck initially limits the table to 1024 nodes, but doubles that limit after any GC in which more than half the nodes survive. This scales well for programs with many PDBs.
- ***Line Sizes.*** We can store the V bits for multiple consecutive memory bytes in a single table node, i.e. have a larger *line size*. A node (line) is then stale only if every byte in it is stale. Bigger lines are better if PDBs are clustered, because fewer lines will be needed, saving space and lookup time. But if PDBs are sparsely distributed, bigger lines will just take up more space. (The issues are similar to those affecting cache line sizes.) Memcheck uses a line size of 16 bytes, which provides a good balance.
- ***Eviction policies.*** A GC should not immediately evict all stale lines from the table, because lines may become non-stale soon, in which case unnecessary work will have been done. Memcheck uses an aging mechanism: during a GC it only evicts lines that have not been touched for three GCs.

This policy ensures that the secondary V bits table lookups have negligible performance impact in all but the most pathological cases. With the large common-case time and space savings compressed V bits are a clear overall improvement.

4.4.4 Another Trade-off

Compressed V bits show another important trade-off, this time between precision and performance. Memcheck does not detect writes to read-only memory before they occur. It would require an extra read-only state, which would be common, and so five states would be needed. Five states would not fit neatly into two bits, so the implementation would be much slower. (A read-only state was omitted from the pre-compressed V bits representation for similar reasons.) Besides, the additional benefit would be small because such errors are rare and they usually cause segmentation faults which Memcheck can pinpoint immediately afterwards anyway.

5. Evaluation

In this section we evaluate the robustness, speed and memory usage of Memcheck's shadow memory implementation.

5.1 Robustness

Memcheck's key shadow memory robustness feature is its division of shadow memory into smallish (SM-sized) chunks which can be laid out very flexibly. The only restriction is that the base Memcheck executable (which is 4MB and contains some Memcheck code and some Valgrind code) is statically linked and so must be loaded at a pre-specified address. The address chosen is one that is rarely used but never reserved by the kernel. On x86/Linux it is `0x38000000`; no problems have been reported with this address but a user could change it if necessary by changing a configuration file and recompiling Valgrind. This restriction is an implementation detail of Valgrind itself, however, and not inherent in Memcheck's shadow memory scheme.

Robustness is not easy to quantify, and we can provide only anecdotal evidence for Memcheck's robustness—we cite its number of users, and the range of software and systems it has been used on. Section 2.2 described how many users Memcheck has. We have heard from users that Memcheck has been used successfully on programs containing up to 25 million lines of code, on 32-bit and 64-bit platforms, both big-endian and little-endian, on several flavours of Unix. Despite this broad exposure, we are aware of no user problems relating to shadow memory layout while the current scheme has been in place.

In earlier versions, Memcheck used the two-level table, but put all shadow memory into a large contiguous region towards the high end of the address space. This region could then never be used by a client. During this period a number of problems relating to shadow memory layout were encountered by users. Some programs wouldn't work without access to this part of the address space. It was also incompatible with kernels with uncommon address space configurations (such as the top 2GB being kernel-only, instead of the more common 1GB), and with kernels configured to disallow "over-committing", i.e. the mapping of more virtual memory than the machine has physical memory and swap space.

These cases, while not typical, were common enough that we rewrote Valgrind's address space management code to give Memcheck its current flexibility. This flexibility solved the Linux problems, and is becoming increasingly important as Valgrind and Memcheck are ported to OSes that have more restrictive address space layouts than Linux. For example, Mac OS X places the main stack, shared libraries, C library code and Objective-C runtime code in the upper half of address space in a manner that is very difficult to avoid. Similarly, AIX places the main stack, thread stacks, shared libraries and mapped-in network cards at various locations in the address space which Valgrind cannot control. Also, many embedded systems have similarly restrictive address space layouts.

This change also means that Valgrind can now run itself, which it could not do before this change.[4]

5.2 Performance

We performed experiments on 25 of the 26 SPEC CPU2000 benchmarks (we could not run `galgel` as gfortran failed to compile it). Eight of the benchmarks invoke their program more than once; for these (marked with a '*' in Table 1) we ran all of them but only report the results for the longest-running invocation. We ran them in 32-bit mode on a 2.4 GHz Intel Core 2 Duo with 1GB RAM and a 4MB L2 cache running SUSE Linux 10.2, kernel 2.6.18.2. To ensure a fair comparison, we implemented all variants using a single version (a pre-3.2.0 version) of Memcheck as the starting point.

Smaller inputs. The left-hand side of Table 1 shows the slow-down factors of the five versions (M0–M4) of Memcheck from Sections 3 and 4.1–4.4 (all with leak-checking off, because it runs at program termination and is largely orthogonal to the concerns of this paper) on the SPEC "test" inputs.[5] The slow-down factors for perlbmk and fma3d are omitted; the native run-times were so short that their slow-down numbers for M4 were both over 200. The middle portion of the table shows the peak size of shadow memory—the peak combined size of the primary map, DSMs, non-distinguished SMs, and the secondary V bit table (for M4)—for M0–M4.

The four optimisations all improve speed, reducing the mean slow-downs by 3.73x, 1.62x, 1.28x and 1.16x, for a combined speed-up factor of 8.9. The optimisations also reduce the mean memory consumption by a factor of 8.5—extra DSMs by 1.97x (although occasionally drastically more for programs with a lot of code and/or read-only data such as mcf and applu), and compressed V bits by 4.29x. This last figure shows that compressed V bits are highly effective.

Larger inputs. The right-hand side of Table 1 shows the same statistics for fully optimised Memcheck (M4) on the SPEC "reference" inputs. These inputs are so large that the experiments took several days to run. To get an idea of the proportion of the slow-down caused by shadow memory, we also give the figures for two other tools: (a) Nulgrind (NL), the no-instrumentation tool, which shows the base slow-down due to Valgrind; and (b) Memcheck-lite (M5), a version of Memcheck with its register-level V bit propagation and checking turned off, in which almost all of the tool overhead is due to shadow memory operations.

Nulgrind's mean slow-down factor is 4.6. This is high, but the no-instrumentation case is mostly uninteresting because the added instrumentation code dominates execution time, and Valgrind is not optimised for this case [16]. The mean slow-down of 22.2 for Memcheck on the "ref" inputs is respectable given the amount of analysis it is doing. (The improvement over the mean slow-down of 23.4 for the "test" inputs shows how instrumentation costs are usually amortised in longer-running programs). Memcheck-lite's mean slow-down is 16.0x. By subtracting Nulgrind's slow-down factor from Memcheck-lite's slow-down factor, we can estimate that approximately half of Memcheck's overhead is related to shadow memory accesses.

Other tools. Section 6 mentions some published performance results for other shadow memory tools. We do not perform any direct comparisons with other tools because they (a) are built with Valgrind and use basically the same implementation (but less optimised) as Memcheck (Annelid, Helgrind, TaintCheck, Redux); or (b) are proprietary, not publically available, and/or implemented on different platforms (Purify, Eraser, VisualThreads, Hobbes, pinSEL); or (c) use shadow value data structures sufficiently different to be not worth comparing (DRD—see Section 6); or (d) are only capable of running a fraction of the SPEC 2000 benchmark suite (TaintTrace, LIFT).

Nonetheless, as our second and third contributions stated, our detailed description and evaluation of Memcheck's shadow memory implementation exceeds anything else in the literature.

6. Related Work

In this section we compare Memcheck's shadow memory implementation to those of other shadow memory tools, all of which were introduced in Section 1.

Other Valgrind tools. Four of the tools (other than Memcheck) mentioned in Section 1 were built with Valgrind: Annelid, Helgrind, TaintCheck and Redux. Like Memcheck, they all use the two-level shadow memory data structure. Unlike Memcheck, they do not use all of Section 4's optimisations because they are more experimental, so their performance is not as critical.

Hobbes, TaintTrace and LIFT. Hobbes [3] and TaintTrace [4] use a simple implementation of shadow memory that we call "half-and-half". They put client memory in the bottom 1.5GB of address space, shadow memory in the next 1.5GB, and assume the top 1GB is reserved for the kernel (this is all for 32-bit machines). Shadow memory accesses become so simple—each memory byte's shadow byte is found at a 1.5GB offset—that they can be inlined rather than requiring a C call, which makes them very fast. LIFT [18] is similar, but shadow memory is 1/8th the size of client memory because each memory byte has a 1-bit shadow, and so it uses a scaled offset.

Hobbes' reported slow-downs for SPECint programs were in the range 30–187x. However, other parts of Hobbes were inefficient and so this is a poor comparison point. TaintTrace is implemented with DynamoRIO [1], and its reported average slow-down is only 5.5x for six of the SPECint benchmarks. LIFT [18] is built with StarDBT, and has a mean slow-down factor of 3.5x for a similar subset of the SPEC CPU2000 integer benchmarks. There are two main reasons why they are much faster than Memcheck [16]: (a) they are doing simpler analyses, and (b) they use some instrumentation techniques that are faster but do not handle as wide a range of

[4] With one proviso: the "inner" Valgrind must be configured so it (i.e. the static executable) is loaded at a different address to the "outer" Valgrind.

[5] These versions are so slow that larger inputs would have taken weeks to complete.

	"Test" inputs										"Ref" inputs			
	Slow-down Factor						Peak Sh Mem Size (KB)				Slow-down Factor			ShMem
Prog.	M0	M1	M2	M3	M4	Tx	M0,M1	M2,M3	M4	Mx	NL	M4	M5	M4,M5
bzip2*	162.9	45.0	23.7	20.8	19.2	8.5	27,040	21,424	4,880	5.5	3.6	17.1	12.9	47,888
crafty	252.0	110.5	51.3	38.1	35.1	7.2	5,584	3,856	864	6.5	7.0	35.9	26.2	864
eon*	380.7	201.8	133.8	59.8	55.1	6.9	5,296	2,416	656	8.1	8.4	51.9	51.1	704
gap	243.4	113.1	48.4	35.7	30.2	8.1	78,304	43,312	7,664	10.2	4.1	26.7	17.4	49,728
gcc*	234.9	112.5	51.0	39.9	37.0	6.3	14,944	12,064	2,873	5.2	5.2	33.3	24.5	25,523
gzip*	173.9	40.4	26.3	20.3	15.7	11.1	12,496	10,768	2,496	5.0	3.0	13.7	10.6	48,576
mcf	184.9	63.5	32.0	19.5	16.1	11.5	109,264	3,136	512	213.4	2.1	7.1	5.4	528
parser	216.7	72.6	45.0	24.1	18.4	11.8	38,128	8,896	2,144	17.8	3.9	17.9	13.9	7,200
perlbmk*	(omitted; run-time too short)						3,496	1,264	464	7.5	4.8	25.3	18.9	40,512
twolf	156.7	47.7	32.8	28.3	25.4	6.2	4,432	2,704	704	6.3	3.2	15.8	11.5	5,584
vortex*	238.9	119.1	64.4	49.0	43.7	5.5	36,904	34,672	7,632	4.8	6.9	41.2	30.8	20,784
vpr*	172.2	52.6	30.9	24.2	21.4	8.0	3,568	2,056	512	7.0	4.3	20.3	14.1	1,552
ammp	113.6	39.9	36.1	32.7	28.2	4.0	28,192	22,504	5,072	5.6	3.6	32.7	27.0	5,088
applu	222.2	38.8	27.5	27.2	25.1	8.9	221,728	13,072	3,008	73.7	5.4	19.3	11.9	47,728
apsi	223.1	26.3	20.6	19.1	16.9	13.2	223,168	221,872	49,392	4.5	3.7	16.2	11.1	49,600
art*	207.1	45.0	44.1	43.0	25.9	8.0	7,024	5,152	1,200	5.9	5.1	24.4	21.6	1,568
equake	205.1	59.4	25.5	21.0	17.9	11.5	29,632	28,696	6,320	4.7	4.3	17.1	13.3	25,472
facerec	135.1	27.4	20.8	16.7	14.2	9.5	33,880	29,056	6,480	5.2	4.9	18.4	11.8	6,848
fma3d	(omitted; run-time too short)						5,872	2,704	736	8.0	4.3	25.4	18.2	28,592
lucas	331.8	67.8	37.6	27.2	24.8	13.4	3,928	2,056	576	6.8	4.1	23.3	14.6	37,056
mesa	202.2	109.3	45.1	30.9	29.2	6.9	27,256	11,200	2,560	10.6	5.9	58.8	33.4	2,704
mgrid	205.5	20.0	20.0	20.0	17.5	11.7	67,000	65,632	14,672	4.6	4.1	16.8	11.2	14,720
sixtrack	268.8	34.8	27.8	22.3	20.2	13.3	74,560	39,352	8,464	8.8	6.4	19.8	15.2	9,648
swim	183.8	25.8	16.7	16.0	13.5	13.6	222,736	87,232	19,456	11.4	3.7	10.8	7.1	49,296
wupwise	279.1	63.2	35.6	30.0	25.6	10.9	205,960	204,520	45,520	4.5	7.8	26.9	19.1	45,536
geo. mean	209.6	56.2	34.7	27.2	23.4	8.9	254,65	12,928	3,013	8.5	4.6	22.2	16.0	11,144
rel. imp.		3.73	1.62	1.28	1.16			1.97	4.29				1.38	

Table 1. Performance of six Memcheck variants (M0–M5) and Nulgrind (NL). Column 1 gives the program name; integer programs are listed before floating-point programs. Columns 2–6 give the slow-down factors for M0–M4 (with "test" inputs), and column 7 (Tx) gives the overall speed improvement from M0 to M4. Columns 8–10 give the shadow memory sizes for M0–M4, and column 11 (Mx) gives the overall shadow memory reduction from M0 to M4. Columns 12–14 give the slow-down factors for Nulgrind, M4 and M5 (with "ref" inputs). Column 15 gives the shadow memory size for M4 and M5. The second-last row gives geometric means of each column. The last row gives the relative improvements in the means for M1–M4.

programs, and half-and-half shadow memory is one of these techniques.

Unfortunately, although half-and-half is simple and fast, its less flexible layout means it fails for some programs under Linux, and is incompatible with OSes with more restrictive memory layouts such as Mac OS X and AIX, as Section 5.1 explained.

For these reasons, for 32-bit machines, half-and-half is unsuitable for Memcheck and related Valgrind tools, for which robustness is as important or more important than performance. For 64-bit machines the situation is less clear, but we suspect similar problems would arise with half-and-half in that setting. In comparison, the two-level table approach provides acceptable performance and excellent robustness. This is an example of a crucial design trade-off.

The Hobbes, TaintTrace and LIFT papers are notable for being the only other publications we know of that describe a shadow memory implementation in detail beyond a couple of sentences. Also, all three tools could be changed to use a two-level shadow memory implementation.

Other tools. The original version of Eraser [20] used the half-and-half approach. The commercial version uses an approach more like Memcheck's—each memory page has a shadow page, a shadow page table does the real-to-shadow page mapping, and an array is used as a mapping cache (shadow TLB) [2]—but there is no publication describing it.

Purify [6] uses "a bit table that holds a two-bit state code for each byte in the heap, stack, data and bss sections". The two-bit state code is like Memcheck's compressed VA bits but without the `PARTDEFINED` value for handling PDBs. We know of no published information about the bit table's structure.

VisualThreads [5], another data-race detector, uses a two-level table like Memcheck, but with much larger secondary maps (16MB vs. 64KB). Judging from the cited paper, the primary map is a structure with a non-constant lookup time such as a tree. This is in contrast to Memcheck's first-level lookup which is constant-time. Larger secondary maps cause more memory to be wasted in the cases where secondary maps are only partially used, and DSMs are likely to be less effective. The paper also says: "This table lookup was added for improved robustness necessary in a product, even at the cost of some additional execution overhead." We suspect this cryptic statement corroborates our claim that a flexible layout is required for robustness, as opposed to the half-and-half scheme.

pinSEL [11] uses a two-level table, with smaller secondary maps than Memcheck (4KB vs. 16KB). Its primary map is a hash table. The reported slow-down for pinSEL is in the range 10–163x, with an average of 93x.

DRD [19] structures shadow memory differently. It needs to record all the memory bytes accessed during a *segment* (a time-slice). For each segment it uses a bit-map, where each bit represents a memory byte. Each bit-map is structured like our two-level table, but instead with nine levels. This makes lookups slower, but results in very little wasted space in the sparsely populated segment bit-

maps, which is important as there can be many segments live at one time. The measured slow-down factors ranged from 10–247.

Other DBI frameworks. Although this paper described a tool implemented using Valgrind, the techniques described here would be suitable for use with shadow memory tools built with other DBI frameworks such as Pin [9] and DynamoRIO [1].

OS page tables. Memcheck's two-level shadow memory table looks somewhat like an operating system (OS) page table. The obvious similarity is that page tables divide address space up into smallish chunks, as Memcheck's table does.

However, there are many differences. OS page tables point to pages of original values rather than shadow values, so there are no questions about shadow value representation, such as whether compression is suitable. Also, shadow value tools do not have to deal with issues that OSes do, such as making decisions about which pages should be swapped out, nor track which files are mapped to which pages. Finally, the performance issues are completely different because page tables benefit from hardware TLBs. Could a shadow value tool somehow utilise a hardware TLB to speed it up? We do not see how it could, since all existing shadow value tools we know of are user-mode programs.

7. Future Work and Conclusion

A number of powerful DBA tools share one crucial characteristic: the use of shadow memory. We have shown how to implement shadow memory in a manner that is highly robust and acceptably fast. We began with a simple but slow implementation in Memcheck, and improved it by (a) speeding up common cases such as loads, stores, range-setting and stack pointer updates, and (b) reducing the size of shadow memory using both high-level and low-level compression. The resulting implementation is fairly fast, very compact and robust, and used by thousands of programmers daily. The results show the importance of low-level representation details and operations in good shadow memory implementations.

We think there are three main areas of future work in shadow memory. First, the performance issues thrown up by 64-bit address spaces and multi-processor machines need to be addressed. Second, the performance of shadow memory tools could still be improved, perhaps with better representations, or by finding ways to omit unimportant shadow memory operations. Third, new tools that use shadow memory in new ways could be created. For example, a profiling tool that tracks how values flow through memory and how often they are copied might help programmers reduce the memory bandwidth requirements of their programs; shadow memory would be an important part of such a tool.

Shadow memory tools are powerful. We look forward to seeing them become better, faster, and more widely-used.

Acknowledgments

Thanks to Greg Parker for his Mac OS X expertise, Jeremy Fitzhardinge for the multiple DSMs idea and implementation, Donna Robinson for encouragement, and Mike Bond, Kim Hazelwood, Kathryn McKinley, Jeremy Singer and the anonymous reviewers for helpful comments on earlier versions of this paper.

References

[1] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, pages 265–276, San Francisco, California, USA, March 2003.

[2] M. Burrows. Personal communication, February 2006.

[3] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In *Proceedings of CC 2003*, pages 90–105, Warsaw, Poland, April 2003.

[4] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of ISCC 2006*, pages 749–754, Cagliari, Sardinia, Italy, June 2006.

[5] J. J. Harrow, Jr. Runtime checking of multithreaded applications with Visual Threads. In *Proceedings of SPIN 2000*, pages 331–342, Stanford, California, USA, August 2000.

[6] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, San Francisco, California, USA, January 1992.

[7] K. Hazelwood. *Code Cache Management in Dynamic Optimization Systems*. PhD thesis, Harvard University, Cambridge, Mass., USA, May 2004.

[8] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, San Francisco, California, USA, August 2002.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI 2005*, pages 191–200, Chicago, Illinois, USA, June 2005.

[10] A. Mühlenfeld and F. Wotawa. Fault detection in multi-threaded C++ server applications. In *Informal Proceedings of TV06*, pages 191–200, Seattle, Washington, USA, August 2006.

[11] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operation system effects to guide application-level architecture simulation. In *Proceedings of SIGMetrics/Performance 2006*, pages 216–227, St. Malo, France, June 2006.

[12] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, United Kingdom, November 2004.

[13] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of SPACE 2004*, Venice, Italy, January 2004.

[14] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[15] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[16] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI 2007*, San Diego, California, USA, June 2007.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS '05*, San Diego, California, USA, February 2005.

[18] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-oeverhead practical information flow tracking system for detecting security attacks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (Micro'06)*, Orlando, Florida, USA, December 2006.

[19] M. Ronsse, B. Stougie, J. Maebe, F. Cornelis, and K. De Bosschere. An efficient data race detector backend for DIOTA. In *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13, pages 39–46. Elsevier, February 2004.

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[21] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, April 2005.

[22] The Valgrind Developers. Valgrind. `http://www.valgrind.org/`.