Measurement Bias from Address Aliasing

Anonymous
Anonymous University
Anonymous
Anonymous, Anonymous
anonymous@anonymous.com

Anonymous
Anonymous University
Anonymous
Anonymous, Anonymous
anonymous@anonymous.com

ABSTRACT

Understanding program behavior and obtaining accurate measurements is important in performance analysis. However, recent research has shown that performance measurements can be *biased* by external factors in unpredictable ways. Seemingly irrelevant properties—such as the length of your user name—can impact program performance. In this paper, we identify an important underlying mechanism that causes this type of measurement bias on modern Intel microarchitectures. Our approach is to use a large set of hardware performance counter measurements to reveal the inner workings of the CPU, and search for correlations over a series of execution contexts. We find that *address aliasing* can explain bias from two external factors; size of environment variables, and characteristics of dynamically linked heap allocators.

This result not only makes accounting for and avoiding measurement bias much simpler, but also enables explicit optimizations for it. We demonstrate this by implementing runtime detection and correction of aliasing conditions. For heap allocators, we show that common implementations tend to give worst case alignment by default, favoring page alignment for large allocations. The performance impact of unfavorable memory layouts can be significant, with as much as $2x$ speedup in one of our examples. Finally, we show that even highly optimized BLAS libraries can be impacted by address aliasing. Our results can enable new architecture specific optimization strategies to account for this phenomenon.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; Performance attributes; D.3.4 [**Processors**]: Optimization; Run-time environments

General Terms

Measurement, Performance

SIGMETRICS 2015 Portland, Oregon USA

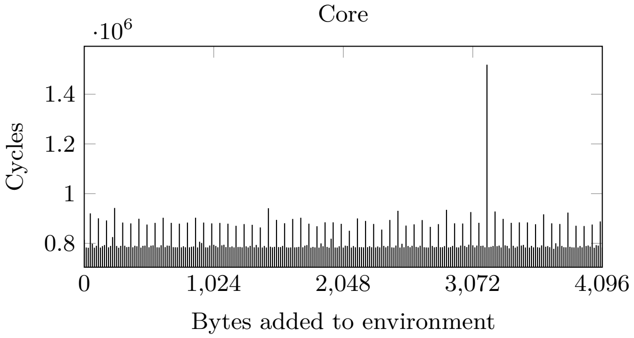

Figure 1: Performance counter data running the micro-kernel from Figure 2 under different environment sizes. Notice the clear spike at around 3200 bytes, where the number of cycles executed nearly doubles. Measuring performance in this configuration would give a biased result.

Keywords

4K address aliasing, BLAS, heap allocators, measurement bias, memory disambiguation, performance counters

1. INTRODUCTION

Accurately measuring the performance of computer programs is important in order to evaluate new algorithms, or compare different implementations. A key challenge for performance analysts and systems researchers is handling *measurement bias*. Changes to external factors of the system, such as link ordering or the contents of environment variables, has been shown to have potentially significant impacts on performance in real applications [16]. The underlying explanation is likely that this changes memory addresses of code or data, which in turn interacts with various low level hardware buffers at runtime. Because of the complexity of modern processors, it is often hard or even impossible to predict exactly how changes to for example system environment variables ultimately affect program execution.

A literature study of 88 papers by Mytkowicz et al. showed that the median reported speedup was within the margin of bias, possibly invalidating the claimed speedups [17]. To alleviate the impact of measurement bias in performance analysis, researchers have proposed techniques like causal analysis, and randomization of execution contexts [17]. Others treat the interaction complexity as a potential optimization

```
static int i, j, k;
int main() {
    int g = 0, inc = 1;
    for (; g < 65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
```

Figure 2: Microkernel first presented by Mytkowicz et al.[16], showing bias to environment size.

problem, using search over variant spaces to find optimal environments [11]. Our approach is to analyze isolated programs in detail using hardware performance counters, in an attempt to identify which underlying mechanisms are causing bias.

In Figure 1, we have reproduced a result originally presented in a paper by Mytkowicz et al.[16], showing how the number of cycles executed for a simple program (Figure 2) depends on environment variables. This example provides the basis for our analysis. We show how some instances of measurement bias, such as the above, can be explained by *address aliasing*, an artifact of a presumably Intel specific optimization on the out-of-order execution pipeline. The CPU uses a heuristic for determining whether loads are dependent on previous stores, comparing only the 12 least significant virtual address bits. Aliased memory accesses can produce false dependencies, and incur performance penalties. We look at how address aliasing can trigger bias from two external factors; size of system environment variables and configuration of dynamic heap allocation libraries.

Our results show that the cost of aliasing can be significant, exemplified by a simple program with up to $2x$ speedup based in heap address alignment alone. We also find that many heap allocation libraries tend to produce worst case behavior by default with respect to aliasing, by favoring page alignment for large allocations. With an understanding of how false dependencies in the CPU can impact performance, measurement bias caused by it can to some extent be predicted and even accounted for in software. We show how techniques like manual padding of allocations and dynamic detection of aliasing conditions can be used to improve performance. Being able to handle address aliasing can give significant speedups, and our findings show potential for further architecture-specific optimizations.

The rest of this paper is structured as follows: Our methodology and experimental setup is explained in the next section. Then section 3 introduces “4K aliasing”, providing necessary background for interpreting the later results. Section 4 presents an analysis of how the size of environment variables causes bias, and in Section 5 we go through a similar analysis for bias to heap address alignment. Finally, in Section 6 we look at alias in BLAS libraries. Related work is discussed in Section 7, before we summarize and conclude in Section 8.

Table 1: Experimental setups used.

Core	Intel® Core™2 Duo P8600 (laptop) 64-bit Ubuntu 14.04 LTS (Linux 3.13.0), GCC 4.8.2-19ubuntu1
Nehalem	Intel® Core™ i7-950 64-bit Ubuntu 12.04 LTS (Linux 3.2.0), GCC 4.8.2-19ubuntu1
Ivy Bridge	Intel® Core™ i5-3470 64-bit Ubuntu 12.04 LTS (Linux 3.8.0), GCC 4.8.2-19ubuntu1
Haswell	Intel® Core™ i7-4770K 64-bit Ubuntu 14.04 LTS (Linux 3.13.0), GCC 4.8.2-19ubuntu1

2. EXPERIMENTAL METHODOLOGY

In order to understand biased behavior it is important to be able to do precise and detailed measurements, without introducing *observer effects* [17]. This can be achieved by using performance counters, instrumentation support in hardware that can be used to *count* various events, such as cycles executed, branch misses, instructions fetched, and so on. Recent Intel architectures have several hundred available events, providing a detailed view of what happens inside the CPU. Performance counters are supported in the Linux kernel, and can be accessed via a tool called *perf*. This utility instructs the kernel to enable the processor’s Performance Monitoring Unit (PMU), before executing a specified program. Using this technique avoids modifications to the executed program, unlike tools like PAPI [14]. We use the perf-stat command in all experiments, which accepts raw PMU event codes listed in the reference manual [10].

Reliability of PMU measurements have been evaluated by Weaver and Mckee [20]. The potential issues they find are either handled explicitly or are irrelevant to our setup.

A small Python script is used to iterate over an exhaustive set of all known counters, which amounts to about 200 on our architecture. Only a small set of events are collected at a time, to ensure events are actually counted continuously. We do not use multiplexing between a limited set of counter registers, as it will introduce significant variations [13]. Controlled variations in environment size is performed by setting a dummy environment variable to n number of zero characters, starting from a minimal environment.[1] Interesting events are identified by computing linear correlation to cycle count, measuring all counters over a series of execution contexts. Results are also averaged over multiple runs to reduce potential random error, using built-in perf rerun functionality.

Because bias can be hardware dependent, and to keep the scope of this work manageable, we initially chose to focus our analysis on the 4th generation Intel “Haswell” microarchitecture. We later reproduced similar results for the “Core”, “Nehalem”, and “Ivy Bridge” microarchitectures as well. Each setup is described in Table 1, and referenced throughout the paper.

[1] Because perf-stat itself adds a few variables, the environment will never be completely empty.

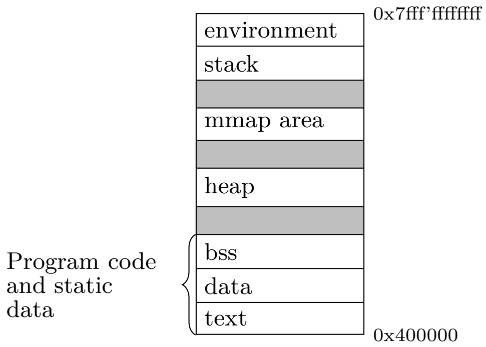

Figure 3: Memory execution context, assuming a 64-bit process running on a Linux system. Initial addresses of stack, heap and memory mapped files are often randomized for security reasons. The stack is also offset by environment variables and program arguments. Addresses of code and statically allocated data are allocated at compile time by the linker, and can be determined by inspecting the executable.

Best practices for controlling the execution context were applied, ensuring that we are not affected by any unwanted bias [16]. Most importantly, this means keeping the memory address space under control. For security reasons, addresses of the stack, heap and dynamic libraries are often randomized at load time, a technique known as *Address Space Layout Randomization* [18, 3]. By disabling ASLR, we are able to execute the same program multiple times with identical virtual address spaces. All experiments are done on a dedicated machine under minimal load, with *frequency scaling* disabled to keep the CPU's clock speed fixed. Finally, we disable *Hyper-threading* (HT), which reduces the possibility for resource contention between threads. Some PMU events also require HT to be turned off, or produce inaccurate results with HT enabled.

3. 4K ADDRESS ALIASING

Modern processors are *superscalar*, and achieve parallelism by issuing multiple instructions simultaneously and out of order. One of the issues that can limit throughput is dependencies between a load and previous stores. To increase parallelism, modern architectures use a technique called *memory disambiguation* to execute memory operations out of order [5]. More often than not, loads can safely be issued before a previous store has completed and written its value to L1 cache. Loads are therefore issued *speculatively*, based on a prediction on whether it will conflict with a previous store that is still not retired. The prediction is later verified, replaying any instructions that were wrongly assumed to have no dependencies. Similarly, if the load and store locations are the same, the value can be *forwarded* from the store before it retires.

While optimizations such as these are good on the average case, there are corner cases. In particular, an event known as "4K aliasing" can occur when the memory addresses of a store followed by a load differ by a multiple of 4096 bytes. A store to address 0x601020 followed by a load to address 0x821020 is an aliasing pair, because the 12-bit address suffix of 0x020 is the same in both. Despite being independent, in these cases the memory order subsystem generates *false* dependencies, causing the load to be reissued. The number of times this happens can be counted by the following event:

LD_BLOCKS_PARTIAL.ADDRESS_ALIAS. *"Counts the number of loads that have partial address match with preceding stores, causing the load to be reissued."* [9, B.3.4.4]

This event is listed in the manual for microarchitectures going back to "Nehalem", including "Ivy Bridge" and "Haswell" CPUs [10]. Older architectures such as Core do not have this particular event listed, but 4K aliasing is covered by a more general event:

LOAD_BLOCKS.OVERLAP_STORE. *"Loads that partially overlap an earlier store, or 4-Kbyte aliased with a previous store."* [10, Table 19-17]

As address aliasing depends on the memory addresses of loads and stores, any environmental factor that affects memory layout has the potential to induce aliasing conditions. In the following sections, we show how performance penalties from address aliasing can be the root cause of measurement bias.

4. BIAS FROM ENVIRONMENT SIZE

System environment variables contain things like the name of the logged in user, home directory, shell state, and various other settings. As a source of bias, it is not the environment variables themselves that are important, but rather the effect their total *size* has on the position and alignment of the stack. As indicated in Figure 3, environment variables and program arguments are allocated in the stack section of virtual memory, close to the upper address 0x7fff'ffffffff[2] and before the first call frame. Changing environment variables will therefore offset the location of the stack, and consequently all stack allocated variables. After this offset, the stack is normally realigned to a 16 byte boundary, enforced by the compiler. Within a span of 4096 bytes there are thus 256 possible initial stack addresses, each representing a different execution context with respect to address aliasing. Note that there is no clear relationship between environment size and stack location with ASLR enabled. However, there will still be as many execution contexts with respect to aliasing (considering the stack only), making any occurrences of measurement bias indeed random.

4.1 Microkernel Analysis

To illustrate how address aliasing can cause bias, we revisit the example first presented in "Producing Wrong Data Without Doing Anything Obviously Wrong!" by Mytkowicz et al. [16], a small C program reproduced here in Figure 2. This example is interesting for several reasons; the bias effects are significant and easily reproducible, while the code itself is simple and straightforward to analyze. Still, no satisfactory explanation as to what actually causes bias was given in the original paper. The following outlines our data and analysis of this program conducted on the "Haswell"

[2] Modern processors do not actually use the full 64-bit space, only the low order 47 bits are used for addressing memory.

Table 2: Events with significant correlation to cycle count.

Performance counter	Median	Spike
ld_blocks_partial.address_alias	137	327,871
cpu_clk_unhalted.thread_p	692,686	825,311
uops_executed_port.port_0	240,248	106,477
uops_executed_port.port_1	249,016	100,913
uops_executed_port.port_2	336,768	345,932
uops_executed_port.port_3	358,848	356,919
uops_executed_port.port_4	280,766	276,036
uops_executed_port.port_5	251,631	121,517
uops_executed_port.port_6	234,337	178,586
uops_executed_port.port_7	137,428	130,325
resource_stalls.any	274,373	406,364
resource_stalls.rs	272,371	135,567
cycle_activity.cycles_ldm_pending	590,879	718,973
cycle_activity.stalls_l2_pending	465,205	490,004

configuration. Similar experiments were conducted on the other experimental setups listed in Table 1 as well, all arriving at the same conclusion.

Performance counter measurements of cycle counts over 512 different environment sizes are shown in Figure 4. Every 16 byte increment of environment size is measured, covering two 4K periods of initial stack addresses. A finer sampling is not necessary, because the stack is by default aligned to 16 bytes. The microkernel is compiled with GCC using no optimization; any optimization would likely disregard most of the function as redundant code, reducing it to return zero immediately.

There are clearly two worst cases, indicated by significant spikes at the 3184 and 7280 byte offsets. We sampled an extensive set of performance counters in addition to cycle count for each execution context. To analyze the bias effects at these points, we calculate the *median* value of each counter over all contexts, and compare that to the two extreme cases. Going through the list of events manually, we selected the set that had the highest correlation with cycle count[3]. Table 2 shows numerical counter values for the first spike, compared to the median values. Event counts on the second spike are virtually identical to the first, and omitted for brevity.

We see the most extreme change from median to worst case in the number of alias events. If we plot the graph for address aliasing, we see that it is near zero everywhere and spikes at exactly the points we observe bias. The results show a high number of resource stalls and pending memory loads when the spikes occur, which is consistent with address aliasing issues. On the other end we get a much lower number of reservation station (RS) stalls in the aliasing case, with a reduction from around 272,000 to 136,000. The reservation station buffers micro-ops for scheduling to the execution units, and a stall event means that there are no free slots available [10, Table 19-2]. Fewer stalls in the aliasing case could indicate less contention on the reservation station. This probably has to do with the overall *decrease* in the number of micro-ops executed per port, as all but one of the in total 8 execution ports have fewer operations going in parallel. Note that the number of micro-ops *retired* overall does not change. The memory disambiguation system automatically assumes dependency between a load and a previous store that are aliased, which can limit the potential for issuing many instructions simultaneously and out of order [9, Page 2-20]. In our case, lower occupation of execution ports, and less pressure on the reservation station, can be explained by the CPU not being able to issue more operations while waiting for an aliased memory access to be resolved.

The cycles_ldm_pending and stalls_l2_pending are interesting, as they are related to the memory and cache system. However, this program use only 2 cache-lines, except for the initial startup. Using the L1 hit and miss counts also revealed that the L1 load is 99.3% and store hit rate is 100%. These events do not indicate a cache hit rate issue, but are probably due to having to wait for aliased writes to be retired and committed to cache.

Performance counter data clearly points to address aliasing as a plausible explanation, thus the next step is to see exactly which memory accesses are aliasing. For that we need to know the addresses of each variable at runtime. The program contains five variables; `g` and `inc` which are stack allocated, and `i`, `j` and `k` which are statically allocated. Memory layout of static data is decided at compile time, and we can find the location of these variables by looking at the symbol table in the ELF executable. We find the addresses to be `&i` = 0x60103c, `&j` = 0x601040, and `&k` = 0x601044.[4]

Observing addresses of stack allocated data at runtime is more challenging, as we have to make sure to not introduce any observer effects that alters the addresses as we are observing them. Assembly code was injected to calculate and output the addresses of `g` and `inc`. We made no change to the stack allocation instructions, and the code offset did not affect the addresses of static variables. The same experiment was run again, showing that the modified program had the exact same bias to environment size. From the output at each spike, we found the addresses to be `&g` = 0x7fffffffe038, `&inc` = 0x7fffffffe03c, and `&g` = 0x7fffffffd038, `&inc` = 0x7fffffffd03c, respectively Notice the common suffix 0x03c between `inc` and `i`, which is the aliasing pair.

Because the stack is aligned to 16 bytes, or 4 *words*, there are a couple of different scenarios that could have been observed here. Static variables are fixed and cover 12 contiguous bytes (3 words), in our case the addresses end in 0x0, 0x4 and 0xc, leaving the 0x8 slot free. The two automatic variables `g` and `inc` occupies 8 contiguous bytes on the stack (2 words), in our case the addresses will always fit in the 0x8 and 0xc slots. In this scenario, `g` will never alias with any of the static variables – as it always covers the 0x8 slot not occupied by either of `i`, `j` or `k`. A less fortunate scenario with respect to the number of alias events occurs when there can be collisions with both stack allocated variables, which can be achieved for example by reserving an extra 8 bytes to

[3] Some correlating performance events are omitted; for example bus-cycles, which will naturally vary with total cycle count.

[4] ELF symbol tables can be read using `readelf -s`

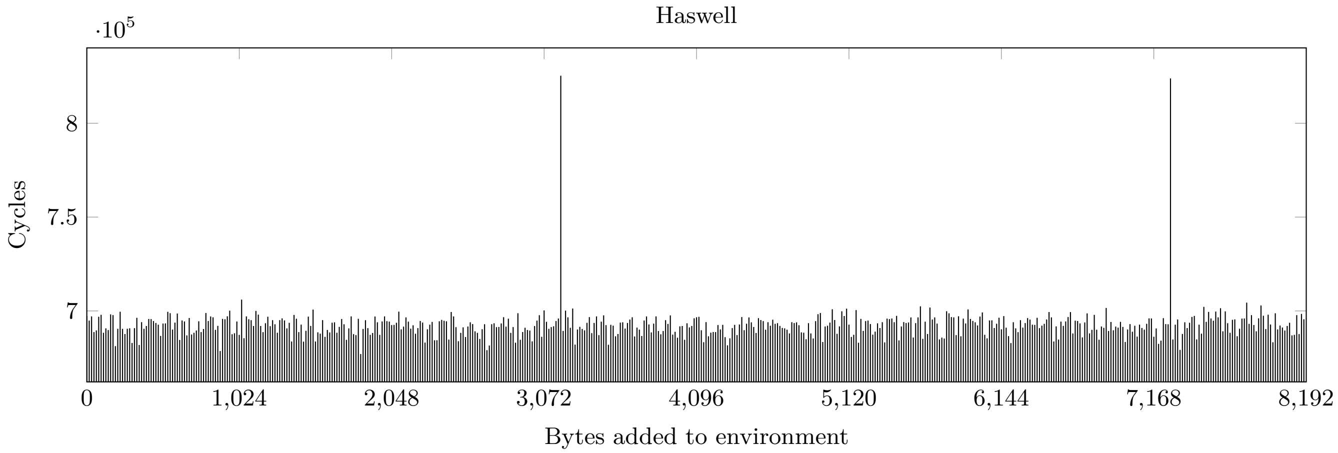

Figure 4: Bias from environment size for microkernel. Measured average of 10 cycle count samples for 512 different environments. Spikes show aliasing case, occurring once for each 4K period.

```
#define ALIAS(a, b) \
    ((((long)&a) & 0xfff) == (((long)&b) & 0xfff))
static int i, j, k;
int main() {
    int g = 0, inc = 1;
    if (ALIAS(inc, i) || ALIAS(g, i))
        return main();
    for (; g < 65536; g++) {
        i += inc;
        j += inc;
        k += inc;
    }
    return 0;
}
```

Figure 5: Modified microkernel that can dynamically detect aliasing case, and avoid it by pushing another stack frame.

offset `i`, `j` into the 0x8, 0xc slots. While this will give significantly more alias counts, we found that it had little effect on the total number of cycles executed.

In conclusion, we identified that address aliasing is the root cause of measurement bias from environment size for this program. Worst case occurs for precisely one out of 256 possible initial stack addresses in every 4K segment, where resource stalls are generated because of false dependencies between the stack and static data. The program is *biased* towards environment sizes that avoid this specific stack alignment, which in principle could be triggered just by changing user name. This type of bias can occur in a number of memory configurations. Here, the aliasing was caused by interactions between the stack and static variables, but we can imagine similar conflicts between the stack and the heap, depending also on how dynamic memory allocators behave. Because the stack is used for local variables, complex code is likely to contain many potential conflicts with other areas of memory.

4.2 Avoiding Aliasing

Addresses of automatic variables can not be determined statically, because the location of the stack at runtime is generally unknown. In addition to being offset by environment variables, the initial stack address can also be perturbed by for example program arguments or address randomization. Although one can not easily know if a collision is going to happen for a given environment a priori, it is possible to change the program to account for possible alias conditions. A proof of concept of how alias-free code could be generated in this particular case is shown in Figure 5. If the addresses do alias, a branch to an alternative but semantically equivalent code path is performed. Calling the function recursively will effectively allocate a new set of variables a bit further down the stack, and the alias condition is avoided.

Even though this particular solution is impractical, it shows that compilers and programmers *could* take measures to account for aliasing.

4.3 Other Architectures

While we show only the analysis for "Haswell" here, the same approach gave similar results for all the CPUs we tested. Figure 6 shows additional cycle count plots for "Nehalem" and "Ivy Bridge", revealing the same alias behavior.[5] The experiment was also done for the older "Core" setup, shown in Figure 1. On this architecture, the *load_-blocks.overlap_store* event is used to indicate address aliasing. Notably, this event was also found to be significant in the original paper, where the authors used a similar setup [16]. We were able to verify that partial address overlap was indeed occuring at the same time this event is spiking, and draw the conclusion that address aliasing must be the underlying explanation. Still, this only explains part of what is going on in Figure 1, as there seem to be yet another bias effect at every fourth environment increment. These smaller spikes correlate with the performance event *load_blocks.std*, which is described as "loads blocked by a preceding store

[5] For these results we also pinned the process to a single core during measurements using `taskset`, which helps remove some of the random noise.

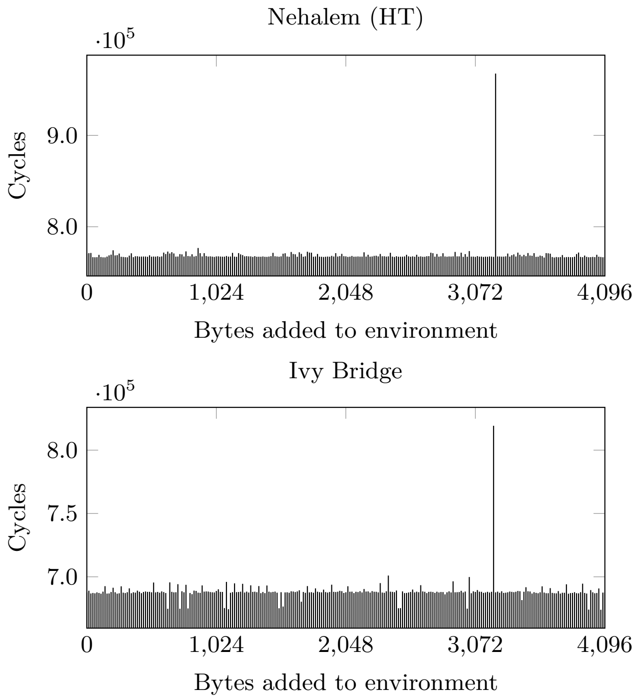

Figure 6: Microkernel experiment on the "Nehalem" and "Ivy Bridge" microarchitectures.

with unknown data" [10, Table 19-17]. Our assumption is that this is a different phenomenon, not related to 4K aliasing. We were only able to observe the effect on this particular architecture, and did not investigate the root cause further.

Results for the "Nehalem" and "Ivy Bridge" architectures are more clear-cut, with large spikes only at the aliasing environment configuration. It is clear that the address alias issue exist for a wide range of Intel microarchitectures.

4.3.1 Note on Hyper-Threading

As described in Section 2, Hyper-Threading was disabled by default in order to avoid any potential interference while collecting data. However, we found that configuration of Hyper-Threading either way did not change the alias behavior. As an example of this, we show the results for "Nehalem" with HT enabled in Figure 6.

5. BIAS FROM HEAP ALLOCATION

Address aliasing can be caused by conflicting pairs of load and store operations to any part of memory. In the previous sections we saw collisions between static data and the stack, observing bias from external conditions that affected addresses of automatic variables. Most dynamic memory is allocated on the *heap*, which is managed by an *allocator*. A heap allocator is responsible for managing dynamic memory, and ultimately assigns the actual addresses of heap allocated variables at runtime. Heap allocation routines such as `malloc` and `free` are typically dynamically linked, for example as part of glibc. If this assignment is independent of the initial stack offset all heap memory accesses can be biased.

Table 3: Addresses returned by different heap allocators when allocating pairs of equally sized buffers.

	5,120 B	1,048,576 B
glibc (ptmalloc)	0x602010	0x2aaaaaaf6010
	0x603420	0x2aaaab096010
tcmalloc	0xe0e000	0xe0e000
	0xe10800	0xf0e000
jemalloc	0x2aaaabc0e000	0x2aaaabc06000
	0x2aaaabc10000	0x2aaaabd06000
Hoard	0x2aaaaab00070	0x2aaaaab00070
	0x2aaaaab02070	0x2aaaabf40070

The particular library used therefore constitutes an important part of the execution context, as linking to a different library, or a library with some alternative configuration, can impact heap addresses at runtime.

5.1 Most Allocators Alias by Default

Acquiring dynamic memory at runtime is usually done by calling `malloc`, which takes a number of bytes to allocate as input, and returns a pointer to that area. Depending on the particular request and allocator used, the returned value will either point to the "regular" heap, or to a memory mapped area (see Figure 3).

- The *heap* is marked by a break point representing the end of uninitialized data in virtual memory, and more space is requested by the `brk` or `sbrk` system calls.

- The `mmap` system call is used to map file descriptors to virtual memory. *Anonymous* mappings, i.e. buffers not backed by a file, can be used for general purpose allocations.

The allocator included as part of glibc uses both mechanisms, choosing which based on the *size* of the request [12]. Small sizes generally live in the heap, while long lived and large allocations tend to be memory mapped.

The heap section starts at a relatively low address right above static code and data. Memory mapped chunks are placed towards the upper end of the virtual address space, closer to the stack. Whether a request is served by the heap or by memory mapping is therefore easy to determine just by looking at the pointer values returned: Addresses in the regular heap can look something like 0x16e30a0 or 0x1723020, while pointers returned by `mmap` are numerically much larger, for example 0x7f0318a8f010 or 0x7f03105d2010. This distinction is unimportant for application developers, as everything is conceptually the same "heap". However, `mmap` has an interesting property in that allocations will always be page aligned. The page size is 4096 bytes, meaning two pointers returned by `mmap` will *always* alias.[6] This behavior is often the worst case for functions that operate on two or more independent buffers.

[6] glibc's version of malloc adds 16 bytes of metadata at the beginning, therefore every memory mapped address ends with 0x010.

Table 3 illustrates how using a different allocator can affect potential aliasing between heap pointers.[7][8] We observe the addresses of two equally sized `char` buffers allocated with `malloc` for different size parameters. Equal three digit address suffix indicate an aliasing pair. In addition to glibc, where the heap allocator is called *ptmalloc*, we look at the following alternatives:

1. Thread-Caching Malloc (tcmalloc) by Google [7].
2. jemalloc, originally developed for FreeBSD [6].
3. Hoard [2].

All of the above focus heavily on performance in multi-threaded environments. Heap allocation is intrinsically inefficient in that all threads share the same address space, leading to a high potential lock contention on memory accesses. Here, we only look at behavior for a single thread, and whether the addresses returned alias or not.

We see that glibc and tcmalloc utilize the normal heap area for smaller allocations, returning numerically low addresses. Interestingly, jemalloc and Hoard appear to never use the heap, but allocate to memory mapped areas even for smaller requests. Conversely, tcmalloc seem manage only the heap. We also find an example where one allocator yields aliasing buffers while another does not. Allocating 2×5120 bytes returns aliasing pointers for jemalloc and Hoard, but not with glibc or tcmalloc. Given that these results are deterministic (with ASLR disabled), it is not hard to construct a program with significant bias towards one or the other allocator. But even with ASLR, pointers returned by `mmap` must still be page aligned. This means that addresses returned by allocators using this mechanism directly will *always* alias, giving a deterministic execution context considering only the address suffix. From our limited analysis, this seems to be the case for glibc, jemalloc and Hoard.

5.2 Aligned Sequential Access

Many functions operate in a “sliding window” fashion; reading and writing to different buffers in some loop construction. This type of access pattern is potentially vulnerable to 4K aliasing, where the worst case will be when the read and write pointer addresses are aliased, continuously generating false conflicts. As an example of this, consider a simple implementation of convolution shown in Figure 7. The performance of this program greatly depends on the address alignment of each buffer, favoring memory addresses that are not closely aligned on the last 12-bits.

We use an input size of $n = 2^{20}$ (4 MiB in memory for each array), which results in glibc’s heap allocator always choosing `mmap` to serve requests. By default, even with address randomization enabled, both `input` and `output` will have start addresses with the same address suffix of 0x010. To analyze performance for different addresses, we manually insert padding to offset the start address of one buffer.

```
static float k[5] = {0.1, 0.25, 0.3, 0.25, 0.1};

void conv(int n, const float *input, float *output)
{
    int i, j;
    for (i = 2; i < n - 2; ++i)
    {
        output[i] = 0;
        for (j = 0; j < 5; ++j)
            output[i] += input[i-2+j] * k[j];
    }
}
```

Figure 7: Basic implementation of convolution with a fixed kernel, ignoring endpoints for simplicity. This program is highly sensitive to aliasing between input and output arrays.

This is accomplished by requesting a bit more memory, and use pointer arithmetic to offset one of the function arguments. Controlling the offset parameter, we can create environments where the address suffixes are any desired number of `sizeof(float)` bytes apart. Allocating and managing these buffers at startup takes a non-negligible amount of work. The overhead can be masked by repeatedly invoking the convolution kernel after allocating and initializing all the inputs. Repeated invocation will also even out performance by warming up the cache.

```
for (i = 0; i < k; ++i)
    conv(n, input, output + offset);
```

An estimate of the actual cost a single invocation can be calculated by averaging over a number of repeated function calls, after subtracting the overhead of invoking it the first time.

$$t_{\text{estimate}} = \frac{t_k - t_1}{k - 1}$$

Our results are with $k = 11$, effectively using the average over 10 loop iterations to estimate the cost of a single invocation. In addition, performance counter measurements are averaged over 10 samples using the repeat mechanism of perf.

Figure 8 illustrates how the convolution kernel behaves for increasing offsets between the heap addresses (modulo 4096), clearly indicating a relationship between address aliasing events and cycles executed. The effects are most distinct on optimization level 2 and 3, where the ratio of cycles to alias events is most significant. Numbers on the x axis represent the amount of offset, measured in number of `sizeof(float)` bytes. An offset of zero is the default behavior for this program when using `malloc` and moderately large inputs. Both for optimization level 2 and 3, the default alignment is close to worst case performance. The differences in cycles executed is significant, with about $1.7x$ speedup for O2 and as much as $2x$ speedup for O3 for increasing relative offset. This phenomenon is only observed for address separations close to zero, thus we only show the first 20 data points. If extended to cover the full width of possible offsets within a 4K segment, we see that the performance is uniform everywhere else.

[7] Switching allocator was done by setting the LD_-PRELOAD environment variable.

[8] Memory mapped addresses starting with the 0x2aa... prefix is an artifact of using *make* to generate results. Executing the test program from *bash* directly result in 0x7fff... prefixes. This difference is not important to our discussion.

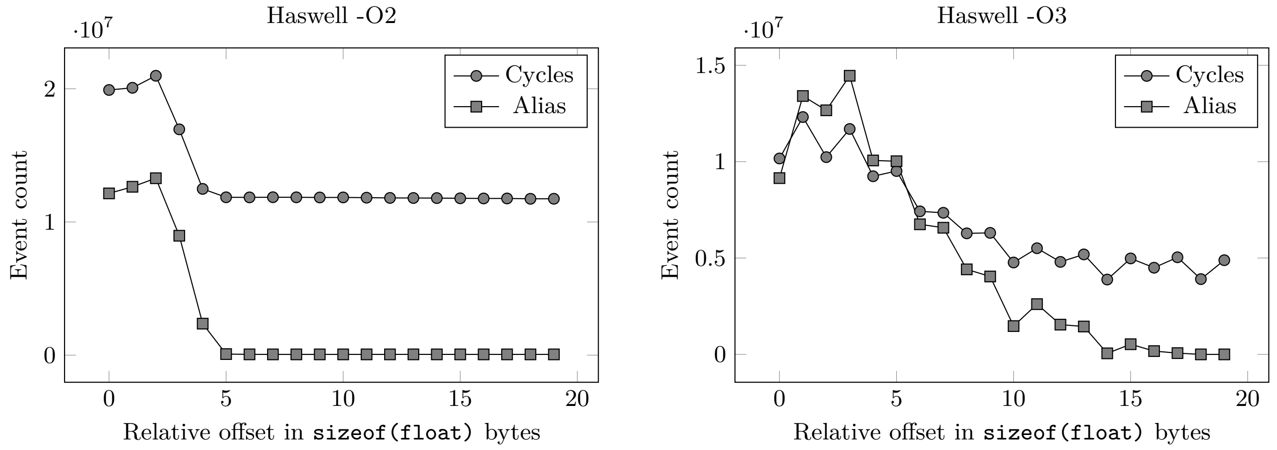

Figure 8: Cycle- and alias counts for different offsets between input and output arrays in convolution kernel from Figure 7. Offset 0 means equal 12-bit address suffix, which is default behavior for `mmap` allocations. Showing results for optimization levels O2 and O3 compiled with GCC, input size $n = 2^{20}$.

More detailed performance counter data is presented in Table 4, where we show the subset that correlates best with the cycle count. The numbers are for the level 2 optimization case, which we choose to study in more detail. However, the performance data looks fairly similar between the two data sets, with a couple of events that appear to stand out:

- A high number of resource stalls for the default alignment, which is reduced substantially together with increasing offsets.
- A high number of cycles with memory loads pending, indicating that the pipeline is stalled waiting for load operations to resolve.
- Changes to the number of micro-ops executed for certain ports.

Interestingly, it looks like small address offsets incur a massive increase in operations issued on port 0 for the O2 case. On "Haswell", this port handles various ALU operations, and branching together with port 6 [9, Figure 2.1]. As there is also variations in the number of branch instructions executed, it seems like these counters together show that certain branches are being re-issued. Unlike the microkernel in section 4.1, the micro-ops executed and RS stall counts drop together with aliasing events. Speculation is possible in this kernel, as there are many independent store addresses. However, false dependencies will limit the amount of speculation, making the CPU incorrectly discard executed instructions. This can explain the inverted behavior compared to the microkernel, where the store addresses were fixed between loop iterations.

It is worth noting that most cache related metrics do *not* stand out in this experiment. For cycles_ldm_pending and stalls_l2_pending, we see similar behavior to the microkernel analysis in section 4.1. These events can be explained by pipeline stalls waiting for aliasing store operations to be retired. The L1 hit rate also remains stable across all offsets, and only a negligible number of memory accesses actually miss L1. On the other hand, we see a fairly strong correlation between cycle count and outstanding offcore requests. These events count the number of outstanding loads to memory outside the processor core each cycle. Since we do not see any significant number of L1 misses, the correlation to outstanding loads is probably a result of stalling and more cycles executed.

Overall, it seems reasonable to conclude that the resource stalling is causing the slowdown, ultimately generated by false dependencies from address aliasing. We do not attempt to pinpoint exactly which accesses generates these conflicts, but at a high level the CPU falsely assumes dependencies between `input[i]` and `output[i]`. By manually adjusting the address alignment of one of these buffers, cycle count can be reduced by as much as 50%. This is a speedup on top of already aggressive compiler optimization.

5.3 Ways to Deal with Heap Address Aliasing

Performance penalties caused by address aliasing can be significant, creating bias towards certain memory layouts. However, with an understanding of the underlying mechanism that causes bias, the effects can be predicted and to some extent eliminated in software. The most relevant scenario to consider is probably code that can hit the aliasing `mmap` scenario, where performance impact can be consistently bad over all environments. We identify some mitigation or optimization techniques that can be used:

5.3.1 Mark buffers with `restrict`

In our convolution kernel implementation, the compiler has to account for the fact that input and output pointers might alias, or that the buffers partially overlap. This limits the extent generated code can keep data in registers without updating the values from memory, as a write to one buffer potentially could invalidate a cached value from the other. The C99 keyword *restrict* can be used to explicitly tell the compiler that accesses through a pointer does not alias with any other, allowing for more efficient code generation with fewer memory accesses.

Table 4: Relevant performance counters and correlation (r) with cycle count for optimization O2. Estimated cost accounting for constant overhead.

Performance counter	r	0	2	4	8
ld_blocks_partial.address_alias	1.00	12,145,292	13,284,521	2,365,416	55,708
cpu_clk_unhalted.thread_p	1.00	19,911,654	20,979,715	12,483,429	11,852,757
offcore_requests_outstanding.demand_data_rd	0.97	260,554,661	259,000,404	153,807,128	131,547,189
offcore_requests_outstanding.all_data_rd	0.91	142,993,699	101,601,131	86,517,306	72,907,030
br_inst_exec.cond.notaken	0.96	1,048,515	1,049,248	877,345	810,971
br_inst_exec.all_branches	0.96	6,294,410	6,292,251	6,129,940	6,033,809
uops_executed_port.port_0	1.00	17,627,202	18,503,555	7,049,536	6,569,198
uops_executed_port.port_6	0.99	10,781,884	11,014,683	9,728,002	9,287,286
resource_stalls.any	1.00	8,312,221	9,161,506	919,717	282,612
resource_stalls.rs	1.00	8,285,878	9,005,145	917,485	281,160
cycle_activity.cycles_ldm_pending	1.00	19,810,652	20,700,085	12,465,608	11,825,649
cycle_activity.stalls_l2_pending	0.98	5,456,418	5,597,398	248,265	304,801

```
void conv(int n, const float * restrict input,
    float * restrict output);
```

With this updated function prototype, the compiler is able to move a store instruction out of the inner loop, reducing the number of stores executed by more than 80 % on optimization level O2. Alias events is reduced by about 10 million for the default alignment, with a corresponding improvement in cycle count.

5.3.2 Use a special purpose allocator

The Intel optimization manual mentions this in *User/Source Coding Rule 8*, suggesting that special purpose allocators could be used to avoid aliasing [9]. However, to our knowledge there are no commonly used allocators that specifically try to mitigate aliasing in heap allocated memory. We show that heap allocators are prone to generate pairwise aliasing buffers, the worst case for functions such as the convolution example.

A potential solution could be to apply some heuristic to randomize addresses more, and in particular not always return the same 12 bit suffix for large allocations. The allocator could also accept hints to which buffers are going to be accessed in parallel, for example as an optional argument to a custom version of `malloc`.

5.3.3 Manually adjust address offsets

In some cases it might make sense to explicitly control the memory addresses used, for example forcing some fixed relative offset between input and output pointers. This can be achieved by exploiting the fact the `mmap` is in fact guaranteed to be placed at a page boundary, and use that directly instead of `malloc`. The following approach can be used to make an anonymous memory mapping with offset `d` bytes away from page alignment.

```
mmap(NULL, (n + d), PROT_READ | PROT_WRITE,
    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0) + d;
```

The same pointer difference must of course by subtracted before unmapping the memory.

6. ALIASING IN REAL APPLICATIONS

To investigate how real world applications can be impacted by aliasing, we look at low level numerical applications. Basic Linear Algebra Subroutines (BLAS) is the de facto standard API for high performance linear algebra routines, with several heavily optimized implementations available [4]. The functionality is divided into three categories:

Level 1 Scalar and vector operations, such as dot product and vector addition.

Level 2 Matrix-vector operations, such as gemv for general matrix-vector multiplication.

Level 3 Matrix-matrix operations, including the widely applied gemm routine for general matrix multiplication.

Procedures operating on vectors, from level 1 or 2, intuitively appears the most likely to have potential for aliasing.

We look at three different libraries available through the Ubuntu package manager; *libblas*, *libatlas*, and *libopenblas* [21, 19]. We ran our bias analysis on the level 2 procedure `cblas_dgemv`, which computes the matrix-vector product $\boldsymbol{y} = \alpha \mathrm{op}\left(A\right) \boldsymbol{x} + \beta \boldsymbol{y}$ for `double` sized numbers. In this equation, α and β are constants, and $\mathrm{op}\left(A\right)$ is an optional transpose or complex conjugate of the matrix. We set $\alpha = 1$, $\beta = 0$ and $\mathrm{op}\left(A\right) = A$ to reduce the formula to $\boldsymbol{y} = A\boldsymbol{x}$. Denote the matrix size as $M \times N$, making $\boldsymbol{x}$ of length N and $\boldsymbol{y}$ of length M. In code, the function is invoked with the following parameters:

```
cblas_dgemv(CblasColMajor, CblasNoTrans, M, N,
    alpha, A, M, x, 1, beta, y, 1);
```

Each of the structures `A`, `x` and `y` represent different locations in memory, and their addresses is part of the execution context. The `mmap` method outlined in the previous section is used to control the location of each of these buffers on the heap. From our experiments, we found that the memory address offset of vector $\boldsymbol{y}$ compared to matrix A is the most important with respect to the number of alias events. For the results presented here, we set $A = I$ (identity matrix) and $\boldsymbol{x} = [0, 1, 2, ..., N-1]$, with parameters $M = N = 1024$. Address suffix of A and $\boldsymbol{x}$ is fixed to 0x000, while performance events are sampled for increasing address offset of $\boldsymbol{y}$, starting at a baseline of 0x000. The experiment was run on the "Haswell" setup, with Hyper-Threading enabled. In an attempt to isolate the cost of a single `cblas_dgemv` invocation, we use the estimation approach described in Section 5 to subtract the constant overhead from heap allocation and initialization.

Cycles executed and alias events for each library is shown in Figure 9. Each library behaves differently, but in all cases we identify a significant amount of alias. The diagrams can be understood the same way as for the convolution example; partial address overlap occur when the address suffix delta of A and $\boldsymbol{y}$ is close to zero. Libblas generates the most distinct pattern, with a relatively clear spike for address deltas between 0x10 and 0x60. Correlation between cycles executed and address aliasing events were calculated to 95% for libblas, 81% for libatlas and 45% for libopenblas.

We do not go into a more detailed analysis of the raw performance data here, but note that we see many of the previously discussed events stand out; such as execution port occupancy, resource stalls, branch instructions executed, offcore requests, and ldm pending. Especially for the libblas case, results look quite similar to what we saw for the convolution kernel.

Our results show that address aliasing can indeed occur in real, performance critical applications. The amount of *bias* this creates with respect to number of cycles executed is not as clear in all cases, but at least for libblas we are confident in claiming that aliasing is indeed causing a performance degradation.

7. RELATED WORK

Measurement bias and observer effect in performance analysis has been studied in detail by Mytkowicz et al. [17, 15]. The authors introduce environment variables and link ordering as examples of external bias triggers, showing how standardized SPECint benchmarks suffer from measurement bias when trying to evaluate the effectiveness of O3 over O2. Strategies for detecting bias through causal analysis and randomization of experimental setups is introduced. The microkernel we analyzed is taken from a followup paper by the same authors, where it is used as an example of how environment size can cause bias [16].

Many different approaches have been explored in order to take performance counter data and processor intrinsics into account for optimization. Knights et al. introduce "blind" optimization, treating the underlying hardware as a black box and use search over *variant spaces* [11]. Their space of variants consist of different position and alignment for each function and global variable, in principle exploring the different configurations possible by altering link order. Others have used machine learning to classify performance counter data, recognizing patterns or *pathologies* in measurements to identify optimization opportunities [22]. The authors of MAO present a framework for extending compilers to provide very low level microarchitectural assembly optimization, using rules, pattern matching, and random insertion of nop-instructions to discover optimal assembly outputs [8].

8. CONCLUSIONS

We have shown how address aliasing affects program performance under different memory contexts, and how it can explain certain cases of measurement bias. The effect is caused by how speculative and out-of-order memory operations are handled by the CPU, which only considers the last 12 address bits to resolve conflicts between a load and previous store operations. This phenomenon is likely present on most consumer hardware based on Intel platforms today, ranging from "Core" to "Haswell" microarchitectures.

In general, any change to how data is laid out in memory layout can potentially introduce bias effects from address aliasing. Analyzing an example with bias to environment size, we determined that collisions between automatic variables and static data resulted in aliasing for certain stack positions. Aliasing conditions were triggered because variations in environment size offset the virtual addresses of stack allocated variables. Dynamically allocated data is controlled by heap allocators, which we introduce as another source of bias. Comparing four different libraries, we show that most implementations tend to favor page alignment for larger requests. This means that structures such as vectors and matrices are likely to alias by default, which can be worst case scenarios for many algorithms. We analyze an example with up to $2x$ variation in cycle count between different heap alignments, showing that address aliasing can have a significant performance impact. We are able to identify instances of alias issues in matrix-vector multiplication routines for different BLAS libraries, proving that this phenomenon also appear in real, performance critical code.

We show how techniques like padding of variables and alternative alias-free code paths can be used to avoid aliasing at runtime. For heap address aliasing especially, we find that manual intervention can be required in order to achieve optimal performance. Our results should inspire more clever optimizations and heuristics taking address aliasing effects into account, as the potential performance improvements can be substantial.

9. ACKNOWLEDGMENTS

This work is ... under the ..., . under the supervision of ..., looking at sources of measurement bias ... [1].

10. REFERENCES

[1] Anonymous. Anonymous. 2013.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[4] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math.*

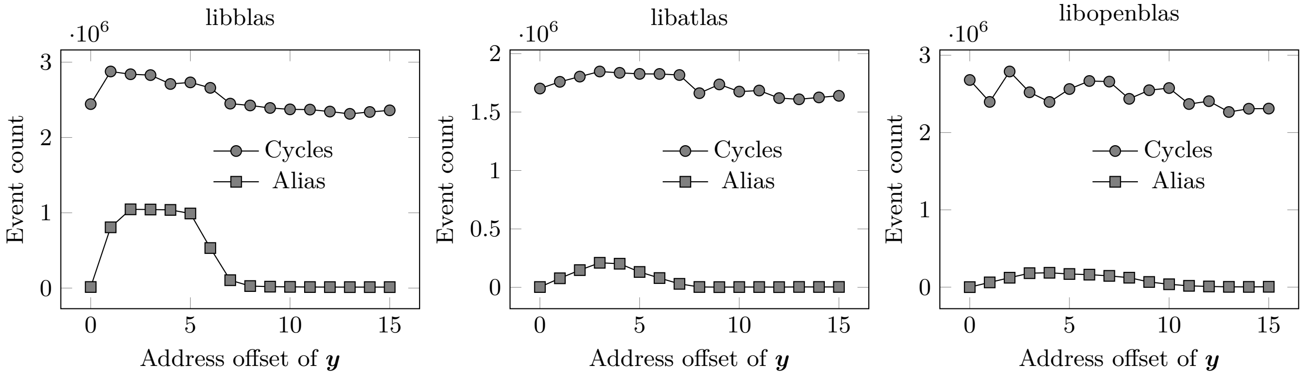

Figure 9: Performance of `cblas_dgemv` for different BLAS packages, measuring cycles executed and address alias events for varying relative address offset between matrix A and vector $\boldsymbol{y}$. Each sample point represents an increment of 16 bytes, or 0x10.

Softw., 28(2):135–151, 2002.

[5] J. Doweck. White paper: Inside Intel® Core™ microarchitecture and smart memory access, 2006.

[6] J. Evans. A scalable concurrent 'malloc(3)' implementation for freebsd. April 2006.

[7] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, February 2007.

[8] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, March 2014.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, February 2014.

[11] D. Knights, T. Mytkowicz, P. F. Sweeney, M. C. Mozer, and A. Diwan. Blind optimization for exploiting hardware features. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 251–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] S. Loosemore, R. M. Stallman, R. McGrath, A. Oram, and U. Drepper. The gnu c library reference manual, 2014.

[13] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. In *13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), 27-29 September 2005, Atlanta, GA, USA*, pages 23–34. IEEE Computer Society, 2005.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. We have it easy, but do we have it right? In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–7, 2008.

[16] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[17] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, and A. Diwan. Observer effect and measurement bias in performance analysis. Technical Report CU-CS-1042-08, University of Colorado, June 2008.

[18] Pax. Address space layout randomization, March 2003.

[19] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013. ACM.

[20] V. M. Weaver and S. A. Mckee. Can hardware performance counters be trusted? In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC*, pages 141–150, Sept 2008.

[21] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[22] W. Yoo, K. Larson, L. Baugh, S. Kim, and R. H. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 283–294, New York, NY, USA, 2012. ACM.