# Counter Inspection Toolkit: Making Sense out of Hardware Performance Event

Anthony Danalis[1], Heike Jagode[1], Hanumantharayappa[1], Sangamesh Ragate[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville, U.S.
[2] Oak Ridge National Laboratory, U.S.
[3] University of Manchester, United Kingdom

**Abstract.** Hardware counters play an essential role in understanding the behavior of performance-critical applications, and inform any effort to identify opportunities for performance optimization. However, as modern hardware is becoming increasingly complex, the number of counters that are offered by the vendors increases and, in some cases, so does their complexity.
In this paper we present a toolkit that is aimed to assist application developers, invested in performance analysis, by automatically categorizing and disambiguating performance counters. We present and discuss the set of micro-benchmarks and analyses that we developed as part of our toolkit. We explain why they work and discuss the non-obvious reasons why some of our early benchmark and analyses did not work, in an effort to share with the rest of the community the wisdom we acquired from negative results.

## 1 Introduction

Improving application performance requires that the people undertaking the effort understand what the performance bottlenecks are. A key step in the process of understanding the factors that limit the performance of an application is the examination of event counters which are recorded by the hardware during execution. Such counters can reveal the behavior of code segments with respect to the hardware. In particular, modern hardware contains performance monitoring units (PMU) that count the events that take place:

- inside CPU cores, e.g., cache misses and branch related events,
- off-core, e.g. power consumption, and bytes read/written from/to memory controller,
- on completely separate hardware, such as network cards.

Many of these events have a rather straightforward meaning and can be mapped directly to the behavior of an application. However, when developers are interested in understanding the behavior of their code in great detail, the events that are counted inside CPU cores can prove to be quite challenging for two distinct reasons. First, the number of counters has been increasing over the years. Developers interested in how branches inside their code affect performance, or how good the cache locality of their memory access patterns is, would find themselves confronted with multiple events that relate to these concepts. Second, due to the increasing complexity of modern CPUs, many of the events that provide detailed information have complex descriptions and contain multiple flags that modify what exactly is being monitored. For example when measuring requests that missed the level two (L2) cache, on an Intel Haswell-EP CPU a developer can choose to count:

– Demand Data Read requests that miss L2 cache.
– All demand requests that miss the L2 cache.
– Requests from the L2 hardware prefetchers that miss the L2 cache.
– Requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that miss the L2 cache.
– All requests that miss the L2 cache.

Clearly, choosing the exact set of events and flags needed for understanding the performance bottlenecks of an application by using such short descriptions of such complex hardware is not ideal.

Abstraction layers, such as PAPI [1], offer *derived* events that map readily to performance abstractions by offering combinations of actual *native* hardware events. However, such derived events hide details that could provide useful insights to the performance analyst.

In the rest of this paper we present the Counter Inspection Toolkit (CIT); a collection of micro-benchmarks and analyses aimed to automatically group hardware counters into logical groups based on what they are counting, and assist performance conscious application developers understand how counters relate to the behavior of code segments. We motivate the need for such a toolkit further by discussing how seemingly simple code segments can lead to non-obvious counter behavior, and discuss all the lessons learned as well as our observations on details that can affect counter values and application performance.

In summary, this paper presents a body of work that aims to help developers that care about performance and put effort in optimizing their applications, but are not hardware wizards with a perfect understanding of chip design and counter semantics.

## 2 Non-obvious code behavior

Let us consider the code shown in Figure 1 and try to reason about the number of conditional branches that will execute as a function of the parameter `size`.

We should first note that this code segment is oversimplified for demonstrative purposes, so we will assume that it has not been compiled with an aggressive optimization level that would replace the whole loop with a simple expression due to the simplicity of the operations performed by the code. By examining the code it is natural to assess that every iteration executes one conditional branch

```
temp = 0;
do{
    temp++;
    if ( ( temp % 2 ) == 0 ) {
        global_var += 2;
    }
} while ( temp < size );
```

Fig. 1: Simple code segment

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( ( result % 2 ) == 0 ) {
        global_var += 2;
    }
} while ( temp < size );
```

Fig. 2: Code segment with RNG

```
     do{
         temp++;
  <310>:  mov    eax,DWORD PTR [...]
  <316>:  add    eax,0x1
  <319>:  mov    DWORD PTR [...],eax

          if( ( temp % 2 ) == 0){
  <325>:  mov    eax,DWORD PTR [...]
  <331>:  and    eax,0x1
  <334>:  test   eax,eax
  <336>:  jne    0x400e77 <353>

              global_var += 2;
  <338>:  mov    eax,DWORD PTR [...]
  <344>:  add    eax,0x2
  <347>:  mov    DWORD PTR [...],eax

         }
     } while( temp < size );
  <353>:  mov    eax,DWORD PTR [...]
  <359>:  cmp    eax,DWORD PTR [...]
  <362>:  jl     0x400e4c <310>
```

Fig. 3: Disassembled simple code.

```
     do{
         temp++;
  <310>:  mov    eax,DWORD PTR [...]
  <316>:  add    eax,0x1
  <319>:  mov    DWORD PTR [...],eax

         pseudo_random_generator();
  <325>:  mov    eax,DWORD PTR [...]
  ...
  <585>:  mov    DWORD PTR [...],eax

         if ( ( result % 2 ) == 0 ){
  <591>:  mov    eax,DWORD PTR [...]
  <597>:  and    eax,0x1
  <600>:  test   eax,eax
  <602>:  jne    0x400f81 <619>

             global_var += 2;
  <604>:  mov    eax,DWORD PTR [...]
  <610>:  add    eax,0x2
  <613>:  mov    DWORD PTR [...],eax

         }
     } while( temp < size );
  <619>:  mov    eax,DWORD PTR [...]
  <625>:  cmp    eax,DWORD PTR [...]
  <628>:  jl     0x400e4c <310>
```

Fig. 4: Disassembled code with RNG.

for the `if` statement and another for the termination condition of the `while` statement.

Examining the assembler code, shown in Figure 3, annotated with the C code by gdb, helps enforce the assessment since the two condition branches ("`jne`" and "`jl`") can be seen in the code and no other branch instruction is present. The expectation can be verified experimentally by instrumenting the code with PAPI to count the number of conditional branch instructions that are executed by the loop, and indeed our experiments were in agreement with the theory.

Now, let us modify the previous program to include code that computes a random number. For simplicity, the random number generator (RNG) code is abstracted away in a macro that stores the random number in the variable `result` without making any calls to functions that would add branches to the execution. This variable is then used in the condition of the `if` statement as shown in Figure 2.

Examining this new C code segment one could assess that the number of conditional branches that will execute must remain the same since the control flow of the code has not been affected by the modification. Furthermore, this assessment seems to be enforced by the corresponding assembler code, shown in Figure 4, since the types and relative positions of the conditional branch instructions in the new code segment are the same as before.

However, when we performed the same experiment as before we found the count of executed conditional branches to be equal to $2.5 \times$ `size`, which at first glance was surprising. An additional clue of what is happening, which might originally seem further perplexing, comes from modifying the program again by adding, seemingly irrelevant, work. In Figure 5, we show an example where we modified the code shown in Figure 2 by adding another call to the RNG after the branch (shown highlighted in red). Performing this change to the code makes the number of executed conditional branches go back to two per iteration.

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( (result % 2) == 0 ){
        global_var += 2;
    }
    random_number( result );
} while( temp < size);
```

Fig. 5: Code segment with RNG and redundant work

Further clues can be found by counting the number of misspredicted branches, which turned out to be equal to $0.5 \times$ `size` in the examples shown in Figures 2 and 5 which use the random number in the condition of the `if` statement, and zero[4] for the example shown in Figure 1 which uses an easy to predict variable in the condition.

The final clue that helps explain this unintuitive discrepancy between these codes is the difference between the number of *retired* conditional branches and the number of *executed* conditional branches. Indeed, the number of retired conditional branches is always two per iteration in all three examples. Consulting

[4] The actual count is not zero, but rather a small number due to noise caused by code not shown in the figures, such as the calls to `PAPI_start()` and `PAPI_stop()`. However, in our experiments this number did not grow when varying the variable `size`, so for large iteration counts the fraction of misspredicted branches approaches zero.

the documentation of the hardware vendor [2], one can see that branch prediction leads to the instructions following the `if` branch to execute speculatively and the count of executed instructions includes instructions whose results were canceled due to a misprediction. On the other hand, at-retirement counting only counts events that were committed to architectural state and ignores work that was performed speculatively and later discarded. In other words, as shown in Figures 3 and 4, since the branch due to the termination condition of the loop, "`jl`", is only a few instructions after the conditional branch of the `if` statement, which is predicted, the "`jl`" instruction will execute speculatively. In the code shown in Figure 1 the speculation has no effect, because the condition of the `if` statement is very regular and thus it is always predicted correctly. However, in the code shown in Figure 2 the random variable will cause the condition to be misspredicted 50% of the time. Thus, in 50% of the iterations the instructions that will execute speculatively (and among them the conditional branch "`jl`") will later have to be canceled, but they will be counted as executed nevertheless. This accounts for the extra $0.5 \times size$ factor in the count of the executed branches for this code (in comparison to the count of retired branches and in comparison to the executed branches for the code of Figure 1.) To say it another way, the "`jne`" branch (of the `if` statement) is the one misspredicted, but the "`jl`" branch is the one with the 50% additional executions.

This explanation also covers the behavior of the code shown in Figure 5, where the `jne` (`if`) branch is also misspredicted 50% of the time, but because of the additional instructions between the `if` statement and the "`jl`" instruction, the latter instruction is not executed speculatively because it is too far away. I.e., the actual condition of the misspredicted branch, `jne`, is evaluated before the speculation can reach that far, and that whole path is discarded.

The set of micro-benchmarks and analyses we are assembling together into the Counter Inspection Toolkit—which is the focus of this paper—aim to highlight such details in hardware counters and program execution and identify connections between them.

## 3 Branch related events

One of the principal goals of this work is to automatically categorize native events based on the higher level concept they count. In other words, if two events have different counts for codes that stress different aspects of the architecture then they belong in different groups, otherwise they are grouped together. This might seem obvious, but keep in mind that in this work we are interested in low level events which have flags that modify their behavior, and in some cases multiple flags can be combined together leading to an combinatorial explosion of possibilities. Therefore, we believe that it is useful for a tool that aims to assist with performance analysis and understanding to offer developers a list with a handful of concepts that relate to branches and the set of events and corresponding flags that count each of these concepts.

For simplicity, in the rest of this paper, when we use the term "native event" we will refer to an event with flags specified, not just the base event with default flags. In the rest of this section we will discuss the effort to categorize events that count branch related execution.

## 3.1 Design Choices

One of our design choices is to keep our micro-benchmarks in C, instead of assembler. The reasoning is twofold. First, we want the benchmarks to be portable between architectures with incompatible instruction sets. Second, and perhaps more important, we want our benchmarks to be easy to read and comprehend by application developers even if they are not comfortable with assembler code. Meeting this design choice, however, can creates challenges, since the compiler might try to rearrange code blocks to optimize execution, especially when optimization flags are used. As an example consider the code shown in Figure 6.

```
do{
    temp++;
    random_number( result );
    if( (result % 2) == 0 ){
        global_var += 1;
    }else {
        global_var += 2;
    }
} while( temp < size );
```

Fig. 6: Code with direct branch.

Since the control flow of the `if-then-else` statement demands that the two blocks are mutually exclusive, one would expect that this code will be translated to assembler that includes a conditional branch and a direct one (to choose one block and skip the other.) This expectation turns out to be correct when no optimizations are performed during compilation. In Figure 7 we show the assembler code generated when the "`-O0`" flag was passed to the compiler and one can clearly identify the highlighted conditional and unconditional jumps, `jne` and `jmp` used to implement the mutual exclusion of the two blocks (as well as the additional conditional jump, `jl`, for the loop termination condition.)

However, if aggressive optimization flags are passed to the compiler then the resulting assembler code does not contain a direct branch, as can be seen in Figure 8. This kind of discrepancies between what a developer assumes that the compiler will do and what the compiler actually does has made it challenging to uphold our design decision to write our micro-benchmarks in C, but so far we have not found any impossible to overcome barrier. To address the specific

```
    do{
...
        if( (result % 2) == 0 ){
<588>:  mov    0x200a5a(%rip),%eax
<594>:  and    $0x1,%eax
<597>:  test   %eax,%eax
<599>:  jne    0x400e2e <618>

            global_var += 1;
<601>:  mov    0x200a21(%rip),%eax
<607>:  add    $0x1,%eax
<610>:  mov    %eax,0x200a18(%rip)
<616>:  jmp    0x400e3d <633>

        }else {
            global_var += 2;
<618>:  mov    0x200a10(%rip),%eax
<624>:  add    $0x2,%eax
<627>:  mov    %eax,0x200a07(%rip)

        }
    } while( temp < size );
<633>:  mov    0x200a31(%rip),%eax
<639>:  cmp    -0x14(%rbp),%eax
<642>:  jl     0x400cf7 <307>
```

Fig. 7: Assembler with `-O0` flag.

```
    do{
...
        if( (result % 2) == 0 ){
<424>:  mov    0x200a82(%rip),%eax
<430>:  test   $0x1,%al
<432>:  je     0x400c08 <136>

            global_var += 1;
<136>:  mov    0x200b76(%rip),%eax
<142>:  add    $0x1,%eax
<145>:  mov    %eax,0x200b6d(%rip)

        }else {
            global_var += 2;
<438>:  mov    0x200a48(%rip),%eax
<444>:  add    $0x2,%eax
<447>:  mov    %eax,0x200a3f(%rip)

        }
    } while( temp < size );
<151>:  mov    0x200b97(%rip),%eax
<157>:  cmp    %ebx,%eax
<159>:  jge    0x400d53 <467>
<453>:  mov    0x200a69(%rip),%eax
<459>:  cmp    %ebx,%eax
<461>:  jl     0x400c25 <165>
```

Fig. 8: Assembler with `-O3` flag.

problem discussed above, we wrote a micro-benchmark (shown in Figure 10(f)) that contains a `goto` statement and has a control flow graph that cannot be simplified by the compiler.

## 3.2 Controlling Branch Misprediction

The codes we showed in Figures 2, 5 and 6 all contain `if` statements with conditions that compare the last bit of a random variable against zero. When a program does that, then the expected rate of branch misprediction is 50%. Because of this, as we discussed in Section 2, the count of executed conditional branches for the code in Figure 2 was equal to $2 \times 0.5 + 3 \times 0.5 = 2.5$ per iteration. Going a step further, we can control the rate of branch misprediction by changing the condition to the one shown in Figure 9. This allows us to control the rate of misprediction by assigning different values to the variable `K`.

Increasing the value of `K` leads to more branches evaluating to `false` than `true` (assuming a reasonable random number generator and an iteration count that is not trivially small.) Therefore, we can expect the branch prediction unit to tend to predict `false` more often than `true`. A naive approximation would be to assume that as soon as $\texttt{K} > 2$ the branch prediction unit will always predict `false`. If that were the case then the count of executed conditional branches

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( (result % K) == 0 ){
        global_var += 2;
    }
} while( temp < size );
```

Fig. 9: Code with variable misprediction rate

for the code in Figure 9 would be equal to $2.0 \times \frac{\mathtt{K}-1.0}{\mathtt{K}} + 3.0 \times \frac{1}{\mathtt{K}}$ per iteration, since only one out of `K` iterations would be mispredicted (and thus only one out of `K` iterations would speculatively execute an additional conditional jump.) In Figure 10 we plot this curve and the experimentally measured count of executed conditional branches for this code.

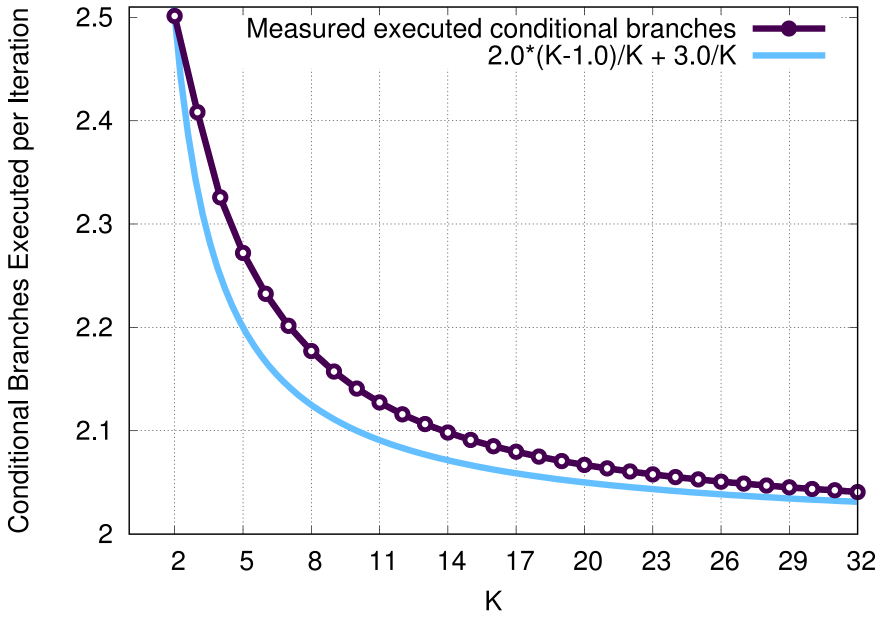

Fig. 10: Controling the misprediction rate

As can be seen in the graph, when $\mathtt{K} = 2$ the code degenerates to the original micro-benchmark where the misprediction rate was 50% and thus the conditional branches that execute are 2.5. Also, as the value of `K` grows (and thus the condition rarely evaluates to `true`), the measured value converges to the naive prediction, but for intermediate values the measured value is slightly above the curve. This suggests that the branch prediction unit tries to identify patterns in the random values instead of merely falling back to always predicting `false`, just because `false` is more frequent. Interestingly, this non-naive behavior of the branch prediction unit leads to more mispredictions than the naive approach of always predicting `false` would have led to.

## 3.3 Event Categories

As we mentioned earlier, one of the goals of this work is to automatically categorize native events based on the hardware features they measure. The main categories for branch related instructions are the following five:

**CE:** **Conditional Branches Executed.** This type of event counts the number of times a branch instruction that depends on a condition is executed by the hardware. Such instructions are generated from high level language statements that affect the control flow, such as `IF`, or loops – where the conditional branch is used for the termination of the loop. Examples from the x86 instruction set are `je/jne` for "jump if (not) equal", `jge` for "jump if greater or equal", `jl` for "jump if lesser", and so on. Note that the instruction is counted as executed even if it executed speculatively, based on the misprediction of a previous branch (as discussed in section 2). Also, a branch does not have to be taken to be considered executed. For example the branch `if( 0 == 1 )` will never be taken but the hardware will still execute it to decide not to take it (assuming the compiler did not optimize it away.)

**CR:** **Conditional Branches Retired.** This type of event is similar to the one above, but the execution of the branch has to be committed to architectural state for it to be counted as retired. This means that it has to either not be part of speculative execution, or the speculation has to be proven correct.

**T:** **Conditional Branches Taken.** This type of event counts only the branches where the condition evaluated to `true` and the branch was actually taken. For this type of event the differentiation between `executed` and `retired` is micro-architecture specific, as not all CPUs, even from the same vendor, offer both versions.

**D:** **Direct Branches Executed.** This type of event counts the number of times an unconditional branch instruction is executed by the hardware. Such instructions are commonly generated from compilers to support the control flow of `IF-THEN-ELSE` statements, or to translate high level language statements such as `goto`.

**M:** **Branches Misspredicted.** This type of event counts the number of times the branch prediction hardware made a wrong prediction. For example, if it predicts that a branch will be taken, because it predicts that the condition will evaluate to `true`, but it is not taken because the condition turns out to be `false`.

In order to categorize the native events of an architecture in these five groups we wrote a set of micro-benchmarks that all contain branch instructions, some of which are shown in Figure 11. As can be seen from the figure the benchmarks share many similarities, but they also diverge from one another. As a result, an event that falls in any of the five categories (CE, CR, T, D, M) should give a set of values, when measured for the different benchmarks, that is not the same across all benchmarks. Furthermore, depending on the category of the event, the set of values for the different benchmarks will be different.

In the following section we will discuss the different approaches we tried in our effort to automate the classification of different events based on the output of these benchmarks and the lessons we learned.

### 3.4 Analysis of benchmark results

The basic methodology behind all the different analysis techniques we have tried relies on varying the iteration count, which is controlled by the variable `size`, and running each benchmark multiple times – for each native event – so that we get a curve from each benchmark for each event.

Our first attempt to associate events with categories was by using the Pearson correlation coefficient. The idea was that a given benchmark stresses a particular category of events, therefore each benchmark would produce a growing curve for the events that belong to this category and a flat curve for all others. Let us take the code shown in Figure 10(a) as an example, and let us call it bench1, for simplicity. Now, consider that for every possible native event on a system, we make multiple runs of bench1, every time setting a different value to the variable `size`. In the resulting data sets we should witness a correlation between the control variable `size` and the measured variable, only for events that measure conditional branches (CE and CR) and taken branches (T) – since these are the only categories of branch events bench1 is expected to trigger. While the correlation coefficient does distinguish the relevant events from the majority of the irrelevant ones, it proved to be a very crude tool unfit for automatic categorization. There are two reasons for this failure. First, we witnessed a few false positives due to noise, and multiple false positives due to events that are completely unrelated to branches (i.e., number of executed instructions), but are legitimately correlated with the iteration count. Second, this technique has a fundamental flaw in that most of our micro-benchmarks trigger events from more than one branch categories at the same time and this technique is unable to differentiate between them.

A better solution is to use the data from each benchmark to calculate a slope for each event using Least Squares Fitting. Each of our benchmarks is expected to trigger a known number of events in each category, per iteration, as shown in the captions in Figure 11. Taking again bench1 as an example and running it multiple times (while varying `size`) for each native event will generate a unique data set for that event. Consider now that we have such a data set by running bench1 and measuring event $E_i$ (e.g. "`BR_INST_EXEC:TAKEN_COND`"). Fitting this data set using least squares will generate a measured slope, $\beta_m$.

```
do{

    if ( temp < (size/2) ){
        global_var2 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(a): {2, 2, 1.5, 0, 0}

```
do{
    global_var2 += 2;
    if ( temp < global_var2 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(b): {2, 2, 1, 0, 0}

```
do{

    global_var2 += 2;
    if ( temp > global_var2 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(c): {2, 2, 2, 0, 0}

```
do{
    random_number( result );
    global_var2 += 2;
    if ( (result % 2) == 0 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(d): {2, 2, 1.5, 0, 0.5}

```
do{
    random_number( result );
    global_var2 += 2;
    if ( (result % 2) == 0 ){
        global_var1 += 2;
    }

    temp++;

}while( temp < size );
```

(e): {2.5, 2, 1.5, 0, 0.5}

```
do{
    global_var2 += 2;
    if ( temp < global_var2 ){
        global_var1 += 2;
        goto zz;
    }
    random_number( result );
zz: temp++;
    random_number( result );
}while( temp < size );
```

(f): {2, 2, 1, 1, 0}

```
do{
    global_var2 += 2;
    temp++;
}while( temp < size );
```

(g): {1, 1, 1, 0, 0}

Fig. 11: Benchmark kernels and their expected values for the branch event categories (CE, CR, T, D, M).

Then this slope can be compared with the expected slope, $\beta_e$, for each event category, so for bench1 we would compare $\beta_m$ against the values 2, 2 and 1.5 – as shown in the caption of Figure 10(a). If $\beta_m$ matches one of these values, then the event $E_i$ belongs to the corresponding category. Using the example of “BR_INST_EXEC:TAKEN_COND”, we expect the measured slope to match the value 1.5 which reveals that this event belongs to the category “**T** (conditional branches taken).”

## 4 Cache related events

The level at which a code is reusing the caches of a CPU usually has a substantial effect on performance. To help assess how well cache reuse is achieved by an application, hardware vendors offer multiple events that count different behaviors of the cache hierarchy. Unfortunately, the complexity of modern cache subsystems has led to multiple such events, some times with non-obvious names and functionality.

To assist developers in choosing which event to use, and understanding what each event measures, we used micro-benchmarks that stress the cache subsystem. The key idea underlying our codes is to control the way memory is accessed, as well as the amount of memory that is accessed and observe how the measured events change.

We use a technique known as pointer chaining (or pointer chasing), which is common in the benchmark literature [3, 9, 12]. The basic idea is to use an array of integers, each long enough to hold a pointer (`uint64_t`). Then, each element of the array is made to point to another element of the array following a random pattern. This creates a “pointer chain”. After this setup phase, the program can start a “pointer chase” where the first element of the array is accessed and the value it contains becomes the next element to be accessed and so on.

The setup of the array can happen off-line, so even an expensive pseudo-random number generator (RNG) can be used, such as the function `random()`, commonly found in POSIX and BSD systems, which employs a non-linear, additive feedback and has a period of $\approx 16 \cdot (2^{31} - 1)$. Using such an RNG the generated pattern becomes exceedingly difficult for the prefetching hardware to guess.

Figures 12 and 13 show the results of running our memory access benchmark with a variable array size while measuring the value of different events. As can be seen, the values of the different events show sharp transitions from 0% to 100% at the boundaries of the different caches—in this example, L1=32KB, L2=256KB and L3=32MB. These sharp transitions can function as signatures that enable us to categorize events based on whether they measure a *hit* or a *miss*, and which cache level they relate to.

Another interesting behavior that we can probe with our micro-benchmarks is prefetching. One can assume that when a miss occurs, the cache fetches more consecutive lines than the one requested speculatively, expecting spatial locality in future memory accesses. If this is the case, then altering the minimum distance between memory accesses (i.e., changing the size of our minimum accessible unit) should result in a different hit rate. In Figure 14 we show three curves that corresponding to three different minimum unit sizes, on an architecture where the actual L2 line size is 64B. Indeed, even when the buffer size exceeds the

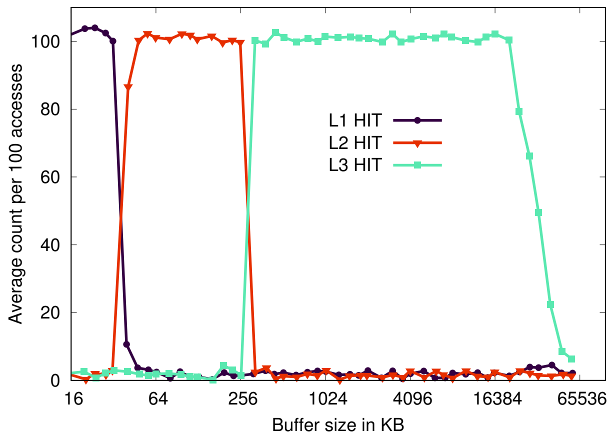

Fig. 12: Cache HIT related events.

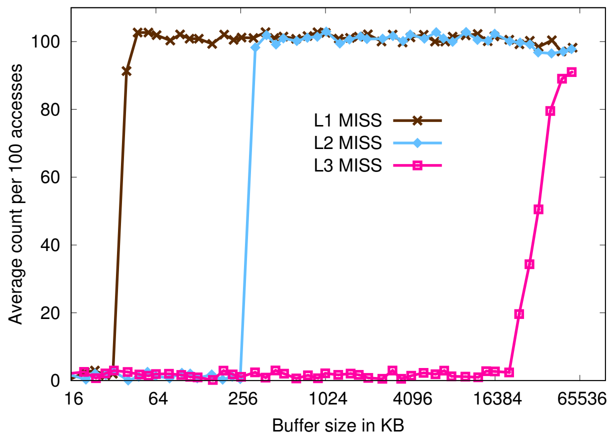

Fig. 13: Cache MISS related events.

size of the L2 (256KB), for small unit sizes the hit rate remains surprisingly high. However, when the unit size increases, which means that the additional consecutive lines are never accessed, the hit rate sharply drops to a very small value when the buffer size exceeds the size of the L2 cache.

Figure 15 shows a different experiment that also tests aspects of prefetching. Here, we kept the minimum unit size constant (128B) across runs, but we altered the way we created the pointer chains. Namely, regardless of the buffer size, the buffer is segmented into logical blocks and each block has each own chain. Therefore, all the elements in the first block are accessed first, then all the

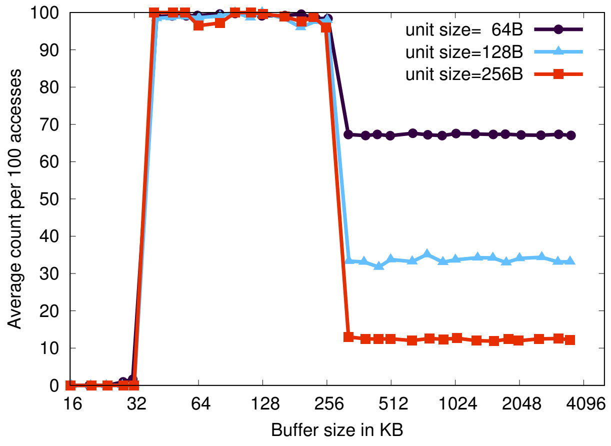

Fig. 14: Effect of unit size on L2 hit ratio.

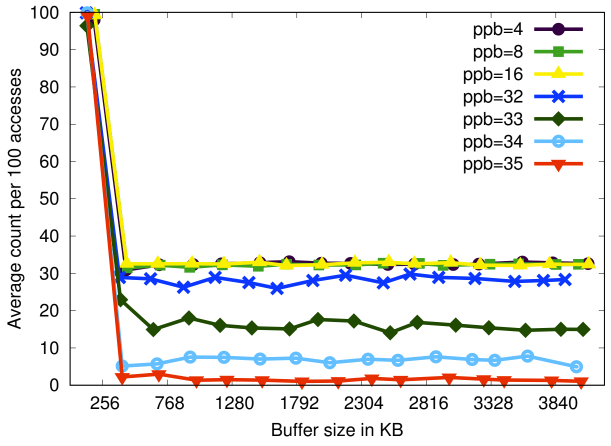

Fig. 15: Effect of block size on L2 hit ratio.

elements in the second block, and so on. The rationale behind this design is to restrict the number of operating system pages in each block, so that the number of TLB misses are minimized during pointer chasing (since TLB misses are often more expensive than cache misses, and pollute the L3 cache, and thus affect the behavior of the memory hierarchy.) As the size of the logical blocks grows it increases the pressure on the cache prefetcher since more and more pages need to be monitored. Indeed, as can be seen in Figure 15, when the number of pages per block (ppb) is 16 or less, the hit rate remains high even past the size of the L2 cache ($\approx 30\%$). However, as the number of pages per block grows beyond 32, even for values barely above 32, the cache hit rate drops quickly, all the way to almost zero. This behavior, can then be used to categorize events that relate to cache prefetching.

## 4.1 Assisting developers with code optimization

As we mentioned earlier, one of the main driving forces behind this work is to assist performance conscious application developers in understanding the behavior of the hardware so they can optimize their codes. As a result, we are interested in behaviors that are demonstrated by micro-benchmarks and can be used to make design decisions in larger applications.

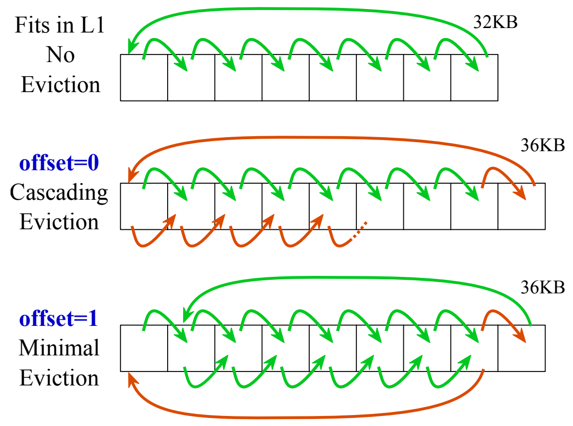

Fig. 16: Access patterns with and without offset.

Consider a system with an L1 cache of size 32KB and an application that accesses a buffer of size 32KB (or less), such that smaller blocks, e.g., 4KB, are accessed one after the other sequentially—instead of the application accessing elements spread out in the whole 32KB buffer. Consider also that after the code accesses the last block, it goes back to the beginning of the buffer and accesses all the blocks again, and this loop continues for many iterations. This behavior is shown schematically in the first line of Figure 16. Since the buffer fully fits in the L1 cache there will be good cache reuse and therefore low average memory access time. This case is shown graphically in the first line of Figure 16. However, if we grow the buffer to $32+4 = 36$KB something interesting happens. When the application now accesses the last block, which does not fit in the L1 cache along with all previous ones, 4KB from the previously accessed data has to be evicted. Since the very first block was the least recently used (LRU) data, and since LRU is a popular replacement policy, the cache will evict the whole first 4KB block. As a consequence, when the code comes around to access the first block again, that block will not be in the L1 cache. Even worse, these new accesses to the first block will evict the second block, which is now the least recently used one. As the code continues, each block will evict the next, and every memory access will lead to a miss in the L1 cache, so it will be served at the latency of the L2 cache. This behavior is shown in the second line of Figure 16.

However, if the application was written by someone aware of this behavior, much better locality could have been achieved. Specifically, if after the last 4KB block is accessed the code skips the first block and accesses all others, leaving the first block for last, then this cascading series of evictions would be interrupted and most of the blocks would be served from the L1 leading to a much lower average access time, as shown in the last line of Figure 16. Hereafter, we will use the term *offset*=1 to describe this approach. Growing the buffer by another 4KB block would defeat this technique, but using *offset*=2 would still work, as would larger offsets for larger buffers. The efficiency of this technique is demonstrated in the performance graph shown in Figure 17, where we show the result of using these access patterns on a machine with L1 latency $\approx 1ns$ and L2 latency $\approx 4ns$.

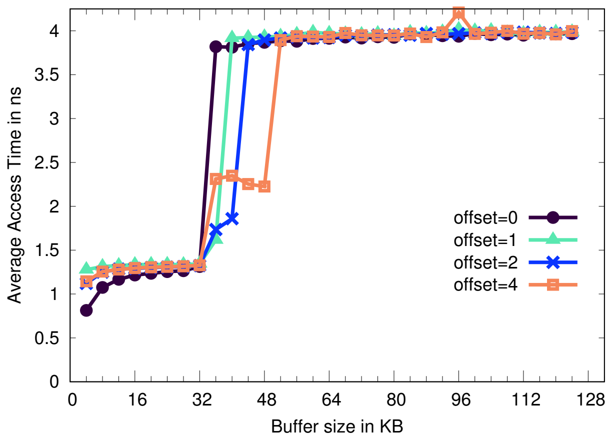

Fig. 17: Cache replacement and access pattern.

## 5 Categorizing events automatically

As we discussed in Section 3.4 the count of branch events grows linearly as we increase the iteration count of our benchmarks. Cache events however, tend to follow step functions that jump abruptly between extreme values. Nevertheless, there are ways to automatically categorize events from both groups, by turning the information we generate through our benchmarks into signatures. In Table 1 we show the expected values for the branch event categories we described in

Section 3.3 for all the benchmarks we showed in Figure 11. As it is easy to see from the table, no two rows are identical. Therefore, if for every event that we test we use the results of all benchmarks together, then we obtain a signature that is unique for each event category.

To make the concept of the signature clearer, consider, as an example, that we perform a test where we run these seven benchmarks and in every run we measure the native event "`BR_INST_EXEC:ALL_COND`." Each benchmark has to be run multiple times, varying the iteration count. Subsequently, we process the measurements taken from each benchmark to obtain a slope. If the slope values we obtain from the different benchmarks are "$2, 2, 2, 2, 2.5, 2, 1$" then by consulting this table we can uniquely identify this native event as belonging to the category "**CE** (conditional branches executed)".

| | bench1 | bench2 | bench3 | bench4 | bench5 | bench6 | bench7 |
|---|---|---|---|---|---|---|---|
| **CE** | 2 | 2 | 2 | 2 | 2.5 | 2 | 1 |
| **CR** | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| **T** | 1.5 | 1 | 2 | 1.5 | 1.5 | 1 | 1 |
| **D** | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **M** | 0 | 0 | 0 | 0.5 | 0.5 | 0 | 0 |

Table 1: Expected values for different branch event categories across multiple benchmarks.

In reality, our measurement will contain noise, so the values acquired from measurements are unlikely to exactly match the values in this table. To address the noise and suppress irrelevant native events whose measurements happen to have a slope similar to an expected one, we also incorporated the correlation coefficient ($r^2$) of the fitting into our "slope goodness function", which is presented below.

$$goodness = \mathrm{e}^{-2\cdot(\beta_m \cdot r^2 - \beta_e)^2} \tag{1}$$

We chose this formula because it has the shape of the normal curve, which is forgiving for small variations, but quickly becomes punishing for larger ones. As a result, when the measured slope, $\beta_m$, is close to the expected slope $\beta_e$ and the correlation coefficient, $r^2$, is very close to one (indicating a good fit), then this formula will produce a number very close to one. However, if the measured slope diverges from the expected slope, or if the correlation coefficient is low, then the formula will produce a number close to zero[5].

[5] Other, more sophisticated goodness functions, such as Pearson's $\chi^2$ test [13], could be used to assist in the analysis of the measurements, but in our experiments we found that the simple formula in Equation 1 is sufficient.

For each native event $E_i$ the seven different benchmarks will produce seven different measured slopes, $\beta_m^1, \beta_m^2 \cdots \beta_m^7$. Using this formula we can compare these slopes against the different rows of Table 1 and get a quantitative assessment of the proximity of event $E_i$ to the category represented by each row.

As we mentioned earlier, the benchmarks that stress cache related events do not produce slopes, but rather step functions. However, these step functions can be readily converted to signatures, if we ignore the multiple values at each plateau and we only keep the actual transitions, which as we showed in Figures 12 and 13 are unique for each cache event category.

As part of our research effort we also developed an LSTM neural network that we trained to recognize the patterns produced by our benchmarks and it proved to by fairly successful when we tried it on architectures other than the one we trained it on (i.e., we trained it on Intel x86 and used it on IBM power8), but the details of this approach our outside the scope of this paper.

Using our benchmarks along with the analysis described in this section produces an automatic categorization of events in a system where the user does not already know which native events belong to each category. Conversly, in a system where the user knows what each native event is supposed to measure, our automatic event categorization can be used for verification of specific native events.

## 6 Related Work

There are several tools and APIs for accessing hardware counters. PAPI [1] is one of the most widespread, part due to its strength as a cross-platform and cross-architecture API. PAPI provides short descriptions of the events that can be measured, but these descriptions are not always self-explanatory, especially so for application developers that are not experts on a given architecture. For Linux platforms the *perf tool* [11] makes use of the *perf_event* API which is part of the Linux kernel. perf_event is even more low-level and the information returned requires considerable interpretation to be useful to application developers.

Further, processor vendors supply tools for reading performance counter results such as *Intel VTune* [16], *Intel VTune Amplifier*, *Intel PTU* [8], and *AMD's CodeAnalyst* [5], but none of these tools and APIs come with a set of benchmarks whose behavior is easy to understand and yet demonstrate behaviors of the underlying hardware that affect application performance.

The closest to the work presented in this paper is the *likwid* lightweight performance tools project [15]. In addition to allowing accessing of performance counters through direct access to the hardware, likwid offers a set of micro-benchmarks that stress different aspects of the hardware. However, unlike our work, these micro-benchmarks are written in a custom low level language that maps directly to x86 assembler and are aimed at calibrating the tool, not to educate application developers about the higher level meaning of different events, or discover the meaning of events on diverse architectures.

In terms of system benchmarks, there are multiple offerings [10, 4, 12, 9, 14, 18, 17, 6, 7] aiming to achieve different goals, such as analyzing the micro-architecture of a specific platform in great detail sacrificing performance, or offering an extendable base of micro-kernels so that more complex benchmarks can be built on top of them. While we have learned valuable lessons from several of these efforts, and we have borrowed techniques, such as the pointer chaining, none of these benchmarks was developed having in mind the goals of characterizing hardware events to provide application developers with a more intuitive high-level understanding of the concepts that are being counted.

## 7 Conclusions

In this paper we presented our work on the Counter Inspection Toolkit; a collection of micro-benchmarks and analyses developed in an effort to categorize and abstract hardware events and map them to higher level performance concepts. The driving force in this effort has been our desire to illuminate the way for application developers that are keen on performance optimization, but are not experts on every esoteric detail of the latest hardware micro-architecture.

We discussed several interesting and non-obvious findings, we demonstrated the feasibility of categorizing events into logical groups, and discussed how automatic analyses can be employed to assist in this categorization. In future work, we are planning to extend the toolkit by adding multi-threaded benchmarks that stress parts of the hardware that related to resource sharing. We also plan to publicly release the code.

## References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.
2. Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, part 2*, 2017.
3. Anthony Danalis, Piotr Luszczek, Gabriel Marin, Jeffrey S. Vetter, and Jack Dongarra. Blackjackbench: Portable hardware characterization with automated results analysis. *The Computer Journal*, 57(7):1002, 2014.
4. Jack Dongarra, Shirley Moore, Phil Mucci, Keith Seymour, and Haihang You. Accurate cache and TLB characterization using hardware counters. In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science*, volume 3036 of *Lecture Notes in Computer Science*, pages III:432–439, Krakow Poland, June 2004. Springer-Verlag Berlin Heidelberg. ISBN 3-540-22114-X.
5. P.J. Drongowski. *An introduction to analysis and optimization with AMD CodeAnalyst$^{TM}$Performance Analyzer.* Advanced Micro Devices, Inc., 2008.
6. Alexandre X. Duchateau, Albert Sidelnik, María J. Garzarán, and David A. Padua. P-ray: A suite of micro-benchmarks for multi-core architectures. In *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*,

volume 5335 of *Lecture Notes in Computer Science*, pages 187–201, Edmonton, Canada, July 31 - August 2 2008. Springer-Verlag.

7. J. Gonzalez-Dominguez, G.L. Taboada, B.B. Fraguela, M.J. Martin, and J. Tourio. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPS.2010.5470358.
8. Intel Performance Tuning Utility. http://software.intel.com/en-us/articles/intel-performance-tuning-utility/.
9. Larry McVoy and Carl Staelin. `lmbench`: portable tools for performance analysis. In *ATEC'96: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, USENIX Association., January 24-26 1996. USENIX Association.
10. Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, Raleigh, North Carolina, September 12-16 2009. IEEE Computer Society, Washington, DC, USA.
11. I. Molnar. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/, 2009.
12. Phillip J. Mucci and Kevin London. The CacheBench Report. Technical report, Computer Science Department, University of Tennessee, Knoxville, TN, 1998.
13. Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine*, 5(50):157–175, 1900.
14. Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX 1998 Annual Technical Conference*, pages 155–166, New Orleans, Louisiana, January 15-18 1998. USENIX Association.
15. J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proc. of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, September 2010.
16. J.H.III Wolf. *Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the VTune$^{TM}$Performance Enhancement Environment*. Intel Corporation, 1999.
17. Kamen Yotov, Sandra Jackson, Tyler Steele, Keshav Pingali, and Paul Stodghill. Automatic measurement of instruction cache capacity. In *Proceedings of the 18th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 230–243, Hawthorne, New York, October 20-22 2005. Springer-Verlag.
18. Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.