Selection on Modern CPUs

Steffen Zeuch Johann-Christoph Freytag

Databases and Information Systems Group
Department of Computer Science, Humboldt-Universität zu Berlin
{zeuchste,freytag}@informatik.hu-berlin.de

ABSTRACT

Modern processors employ sophisticated techniques such as speculative or out-of-order execution to hide memory latencies and keep their pipelines fully utilized. However, these techniques introduce high complexity and variance to query processing. In particular, these techniques are transparent to DBMS operations since they are managed by processors internally. To fully utilize the sophisticated capabilities of modern CPUs, it is necessary to understand their characteristics and adjust operators as well as cost models accordingly.

In this paper, we extensively examine the execution of a relational selection operator on modern hardware in an in-depth performance analysis. We show, that branching behavior and memory exploitation are two main contributors to run-time. Based on these insights, we show how two common cost models would predict execution costs and why they fall short in determining run-time behavior for parallel execution. We reveal, that cost models which exploit only one performance parameter to determine execution costs are not able to predict the non-linear performance characteristics of modern CPUs.

1. INTRODUCTION

Today's processors implement many sophisticated features to accelerate the performance of general-purpose applications. These features are transparent to applications like DBMSs and their usage depends on internal processor states such as resource or memory bandwidth utilization. Research in the field of single-thread DBMS performance shows, that main performance contributors are correctness of speculative execution [14], exploitation of out-of-order execution [13], prefetching [5], utilization of the instruction pipeline [1], and exploitation of the multi-level cache hierarchy [1, 12, 3, 8]. Another field of DBMS performance research focuses on the aspect of parallelization. Research in this area examines exploitation of hyper-threading [18], multiple cores for join operations [2], and multiple sockets [9].

IMDM '15, August 31 2015, Kohala Coast, HI, USA
© 2015 ACM. ISBN 978-1-4503-3713-7/15/08...$15.00
DOI: http://dx.doi.org/10.1145/2803140.2803145

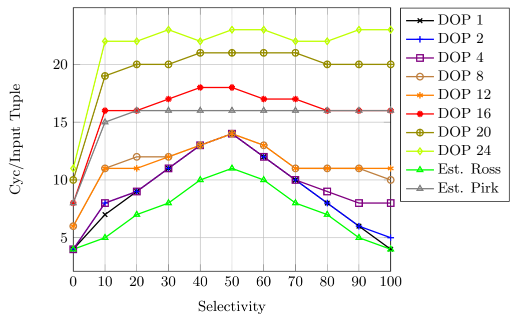


Figure 1: Selection Scalability.

Performance analyses in both research areas could be divided into two categories. Studies in the first category examine DBMS workloads based on overall run-time. The general assumption is, faster execution methods utilize hardware resources more efficiently. Studies in this category mainly investigate the relational join as one of the most complex and time consuming database operation [2]. In contrast, studies in the second category analyze DBMS workloads based on performance counters [1, 8, 5]. These counters measure the efficiency of different CPU components such as branch prediction or pipeline utilization as well as the exploitation of the multi-level cache hierarchy. Studies in this category mainly investigate entire DBMS workloads such as different TPC benchmarks to show, how efficiently a DBMS exploits its available resources.

Studies that utilize performance counters exhibit several shortcomings. At first, several studies are performed on CPU simulators instead of real processors [13, 8]. A simulator enables processor configurations which are most probably not available in any existing CPU. Second, these studies were conducted 15 years ago and thus rely on outdated processors. For example, they do not take multiprocessor technology into account. Third, row-oriented data layouts are examined instead of today's commonly used column-oriented data layouts. Finally, these studies run OLTP or OLAP workloads on commercial DBMSs to infer their exploitation of hardware resources. In these workloads, multiple operators interfere with each other during execution and thus characteristics of individual operators could be misinterpreted. Furthermore, commercial DBMSs introduce several layers of complexity for logging, locking and other maintenance tasks which could potentially distort a single operator analysis.

In this paper, we argue, that complex workloads on commercial DBMSs do not reveal the performance characteristics of individual operators. Therefore, we isolate the re-

	CPU 1	CPU 2	CPU 3	CPU 4
Type	Intel Xeon	Intel Core i7 Mobile	Intel Xeon	Intel Core i7 Mobile
Model	E5-2630 v2	i7-2640M	E7-4870	i7-4900MQ
Microarchitecture	Ivy Bridge	Sandy Bridge	Nehalem	Haswell
Fequency	2.6 GHz	2.8 GHz	2.4 GHz	2.8 GHz
Physical Cores	6	2	10	4
L1 Instruction Cache	6x32KB 8-way	2x32KB 8-way	10x32KB 4-way	4x32KB 8-way
L1 Data Cache	6x32KB 8-way	2x32KB 8-way	10x32KB 8-way	4x32KB 8-way
L2 Unified Cache	6x256KB 8-way	2x256KB 8-way	10x256KB 8-way	4x256KB 8-way
L3 Unified Cache	15MB 20-way	4MB 16-way	30MB 24-way	8MB 16-way

Table 1: Test Systems.

lational selection operator (called *selection* in the remainder) as a basic building block in complex DBMS workloads and execute micro-benchmarks to analyze its performance characteristics on modern processors. Because selections are commonly pushed down in the execution plan and thus are applied to many tuples, performance characteristics of selections are very important for overall query execution time. We show, how processor features like branch prediction or multi-level cache hierarchies impact selection performance, especially for parallel execution. By sampling sequential and parallel selection execution, we reveal their different run-time characteristics.

In Figure 1, we present these run-time characteristics as *CPU time* in *cycles per input tuple* (y-axis) for a single selection using different degrees of parallelism (dop) and selectivities (x-axis). CPU time measures the total time spent by all CPUs individually instead of execution time (so-called wall-clock time). By analyzing these results, we reveal three important performance characteristics of a selection. First, a selection does not scale linearly with the number of cores. A linear scaling would be indicated by the same CPU time but less wall-clock time (shown in Section 6). Second, curves change their trends from a peak at 50% selectivity with two declining edges (dop 1) to a sharp increase followed by constant pathway (dop 24). Between these extremes, there are several transitional curves. Third, common cost models (Ross et al. [14] and Pirk et al. [12]) approximate correct execution costs only for a subset of the entire dop range. We show, that these cost models are insufficient to predict the selection performance on modern processors.

The rest of this paper is structured as follows. In Section 2, we show how selections are transformed into executable code. As a next step, we introduce two major performance factors that determine the performance of a selection. First, Section 3 introduces the branch prediction and its induced wrong prediction penalties. Second, we analyze the number of induced cache accesses including their impact on the memory subsystem in Section 4. Based on these performance analysis, we show how parallelization changes selection characteristics in Section 5. Finally, we show how common cost models predict these changing characteristics in Section 6 before presenting related work in Section 7 and concluding the paper in Section 8.

2. BACKGROUND

In the remainder of this paper, we use the following SQL query as a running example to analyze performance characteristics of a selection:

```
Select Sum(B) From tab Where A <= selValue
```

Our in-memory data set consists of 10M synthetically generated, randomized integer values in two column-oriented arrays (A and B). We adjust selectivity based on `selValue`. In Table 1, we present our test environment that contains four different CPUs based on Intel's latest microarchitectures. If not stated otherwise, we present test results on CPU 1. If they differ from CPU 2-4, we present them explicitly. Our example SQL query can be transformed into the following C++ code (assuming column-oriented data layout):

```
    for (int i = 0; i < tupleCnt; i++)
        if(A[i] <= selValue)
            sum += B[i];
```

This C++ code iterates over all elements in the data set. For each tuple, it first checks if its attribute value $A[i]$ is less or equal to the current selection value. If $tuple_i$ qualifies, its attribute value $B[i]$ is added to the overall sum. Thus, a selection is implemented as a conditional *if* statement in C++.

As a final step, a compiler translates C++ code into machine instructions. For each selection, the compiler generates one comparison instruction followed by a conditional jump instruction. Additionally, one such pair and a loop counter is generated for the entire loop. The conditional jump instruction introduced by the selection determines the execution path as follows. If $tuple_i$ qualifies the selection predicate, the branch/jump is *not taken* and thus the execution continues with the next instruction. On the other hand, if $tuple_i$ does not qualify, a branch/jump is *taken* and thus the program execution jumps to the end of the loop code to test the loop condition.

```
1  LOOP_START:
2      cmpq $rcx,(%rax,%rdx,1) ;compare A[i] to selVal
3      ja LOOP_END ;jump if above (!qualified)
4      add (%rbx,%rdx,1),%r12; sum+= B[i]
5  LOOP_END:
6      add $0x8,%rdx; array offset (i) += 8
7      cmp $x320,%rdx ;compare i to loop counter
8      jne LOOP_START ;jump to LoopStart if !=
```

Listing 1: Assembler Code

Listing 1 (compiled with gcc 4.9) shows a simplified assembler implementation of our running query using two comparison instructions, two conditional branches, and two arithmetic additions. In a prolog (not shown), start addresses of the input arrays and a `selValue` are loaded into registers and a loop offset and temp sum variable are initialized. Registers are preloaded with the following values: `rax` = *start of array A*, `rbx` = *start of array B*, `rcx` = *selValue*, `rdx` = *array offset i*, and `r12` = *temporal sum*. For better readability, we omit physical addresses and introduce `LOOP_START` and `LOOP_END` as jumping labels. In Line 2, $A[i]$ is loaded and compared to `selValue` (in `rcx`). The `rax` register specifies the start address of column A and `rdx` stores the current array offset which represents the loop counter variable `i` in the C++ code. In Line 3, the outcome of the previous comparison is evaluated. If $A[i]$ is greater than `selValue` and thus $tuple_i$ does not qualify, execution jumps to the end of the loop (to Line 5). Otherwise, execution is continued in Line 4 by adding $B[i]$ to the overall sum in register `r12`. Note, this instruction is only executed if $tuple_i$ qualifies.

In Listing 1, loop counter `rdx` is simultaneously used as array offset. Therefore, we increment `rdx` by 8 (size of one tuple) in Line 6 to prepare the next iteration. In Line 7, the new offset is compared to the number of required iterations. Listing 1 assumes 100 tuples and thus the loop terminates if the offset is equal to 800 ($hex = x320$). Finally, in Line 8, the execution jumps either to the beginning of the loop (Line 1) if another iteration is required, or otherwise leaves this code fragment by terminating the loop.

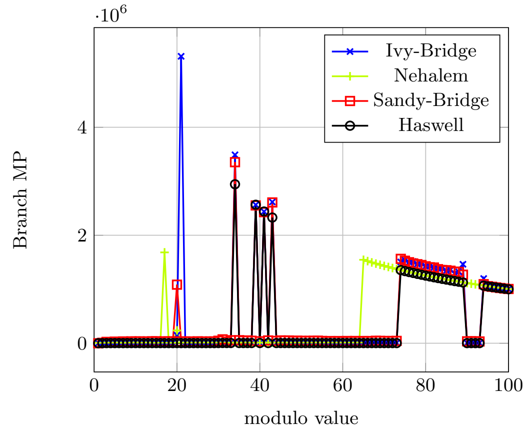


Figure 2: Branch History Buffer.

3. BRANCH PREDICTION

Branch prediction is the first major performance contributor for a selection. A branch predictor in modern CPUs has two alternative options to predict the outcome of a branch. The correctness of its prediction is essentially to fully utilize processor pipelines. First, *static* branch prediction determines that forward jumps (e. g. like a if statement) are not taken and backward jumps (e. g. at the end of a loop) are taken. This simple prediction scheme is applied if no other information is accessible for a branch [7]. Second, *dynamic* branch prediction determines the outcome of a branch based on its branch history. A *Branch Target Buffer (BTB)* saves the branch address as well as its last outcomes to recognize patterns of branches taken/not taken.

To examine the pattern size that modern CPUs recognize, we create different patterns based on a modulo division (if($value_{1...n}\%x$)). In Figure 2, we vary the modulo values from one (all tuples qualify and thus all branches are not taken) to 100 (only each 100th branch is not taken). In general, the x value indicates, that only each x-th branch is not taken and all other branches are taken. If the pattern is not recognized by a CPU, each x-th branch will be mispredicted (plotted on the y-axis). As shown, Ivy-Bridge, Sandy-Bridge, and Haswell CPUs (CPU 1,2,4 in Table 1) detect patterns with up to 72 different outcomes. Starting from $x = 72$, each *branch not taken* induces one branch misprediction and thus we deduce that these patterns are too long to be recognized. Because a BTB stores patterns in a circular manner [7], starting from $x = 72$, each additional outcome overwrites an existing entry. Nehalem as the oldest microarchitecture exhibits a smaller BTB and is capable of detecting patterns up to 64 different outcomes (CPU 3 in Table 1). We emphasize that we cannot point out a particular reason for spikes around $x = 20$ and $x = 40$ in Figure 2 as well as improved branch prediction between $x = 90$ and $x = 93$. Although they are reproducible, information in the Intel manual does not explain their occurrence [7]. However, we conclude for a selection that modern CPUs recognize branching patterns that repeat their history within less than 72 consecutive outcomes. These patterns are mostly introduced by predicates with very high or very low selectivities and explain their excellent right prediction rate.

In general, the branching pattern of a selection is determined by its selectivity p. Following Ross et al. [14], we assume a processor with a *perfect* branch predictor. For a selectivity below 50%, it predicts that each tuple does not qualify and thus each branch will be taken. On the other hand, for a selectivity above 50%, it predicts that each tuple qualifies and thus each branch will not be taken. Because the number of output tuples is equal to the number of branches not taken (BNT), we calculate the number of mispredictions by:

$$BRMP(p) = \begin{cases} BNT(p), \text{if p} \leq 0.5 \\ BNT(1-p), \text{if p} > 0.5 \end{cases} \quad (1)$$

Thus, for a selection with a selectivity below 50%, the branch predictor predicts that each tuple does not qualify (branch is taken) and therefore *mispredicts* each qualifying tuple (branch not taken). Hence, the number of branch mispredicts is equal to the number of branches not taken. On the other hand, for a selection with a selectivity above 50%, the branch predictor predicts that each tuple qualifies (branch is not taken) and thus mispredicts each not qualifying tuple (branch taken). Based on the number of mispredictions and the number of conditional branches (taken + not taken branches), we calculate the number of right predictions by:

$$BRRP(p) = Conditional\ Branches - BRMP(p) \quad (2)$$

Note that, a loop itself induces as many branches as input tuple exists. However, these branches are almost always taken and thus correctly predicted (except for the last iteration).

In Figure 3, we evaluate Equation 1 on the latest four Intel microarchitectures; Nehalem, Sandy-Bridge, Ivy-Bridge, and Haswell. As shown, the estimated number of branch mispredictions matches the measured branch mispredictions for all microarchitectures. However, around 50% selectivity, CPUs mispredict slightly more branches than Equation 1 estimates. Additionally, branch prediction on Nehalem deviates more to Equation 1 compared to Intel's latest three microarchitectures. Following Ross et al. [14], we could estimate branch-induced costs for a selection by combining the estimated mispredictions with a penalty.

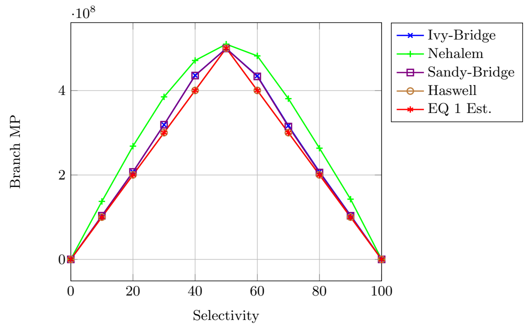


Figure 3: Branch Misprediction.

Figure 4 summarizes relationships between branch-related counters for a selection. First, the number of conditional branches are constant for the entire selectivity range. Second, branches not taken (BNT) and branches taken (BT) converge with increasing selectivity. Because the number of conditional branches remains constant, each additional qualifying tuple reduces the number of not qualifying tuples by one. At zero percent selectivity, no tuple qualifies and thus the branch is taken for each tuple. Additionally, each loop iteration (back to the loop start) induces one BT; thus, the number of branches taken are twice as many as the number of input tuples. For a selectivity of 100%, each tuple qualifies and thus each branch is not taken by the predicate evaluation. Additionally, one branch is taken for each loop iteration.

In contrast, branch prediction shows a different trend. For selectivities below 50%, each not qualifying tuple (BT) results in a right branch prediction and each qualifying tuple (BTN) in a branch misprediction (indicated by the overlapping lines). In contrast, for selectivities above 50%, this correlation switches such that each qualifying tuple result in a right prediction and each not qualifying tuple in a branch misprediction. Thus, a predictable branching behavior (few mispredictions) are induced by very high or very low selectivities. It is important to note, that the number of branches taken and not taken are processor-independent because their occurrence is determined by the input data. In contrast, their prediction depends on the CPU internal branch prediction algorithm. Figure 4 reveals, that these branch prediction algorithms did not change among the latest four Intel microarchitectures. Therefore, branch-related behavior of a selection on modern Intel CPUs is deterministic and can be approximated accurately.

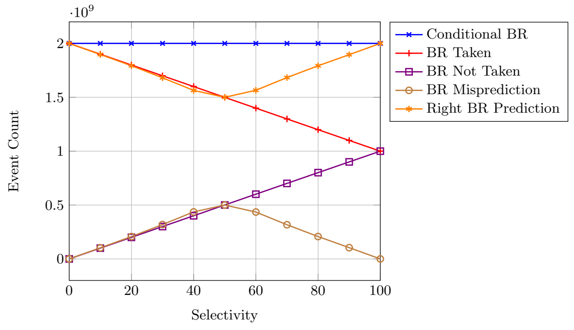


Figure 4: Branch-related Counter.

4. CACHE MISSES

The second major performance contributor of a selection is the number of cache accesses. The extension of the generic cost model (see Manegold et al. [11]) by Pirk et al. [12] allows us to model cache accesses for a selection by combining two access patterns. First, a selection introduces a *sequential traversal* access pattern that in turn induces one random cache miss for accessing the first cache line and one sequential miss of each subsequent cache line. Second, each subsequent operation introduces a *sequential traversal with conditional reads* access pattern which induces cache accesses depending on the selectivity of the previous selection. In our example query, the aggregation function conditionally accesses column B only for tuples that qualify on column A. Therefore, a selectivity of zero percent represents a baseline for accessing only column A. For increasing selectivity, additional work in form of cache line accesses to column B and branch misprediction penalties are induces.

In Figure 5a, we plot L3 cache-related performance counters for our example query using a dop of one. Because of the streaming access pattern (without tuple reuse) and the inclusive property of the L3 cache, the L1 and L2 counter show similar values; thus, they are not presented in this paper. Note, we discuss the rest of this figure in Section 5. In Figure 5a, L3 cache accesses increase up to a selectivity of 20% and then remain constant. The cost model by Pirk et al. [12] estimates this trend exactly by considering the probability of a cache line access. In the selectivity range from 0% to 20%, some cache lines are not accessed and thus random memory accesses occur. With increasing selectivity up to 20%, the probability that two memory references access the same cache line increases. For a selectivity larger than 20%, each cache line is accessed and thus the number of cache accesses remain constant among the entire selectivity range from 20% to 100%. Note that, the actual switch point depends on the number of tuples per cache line [12].

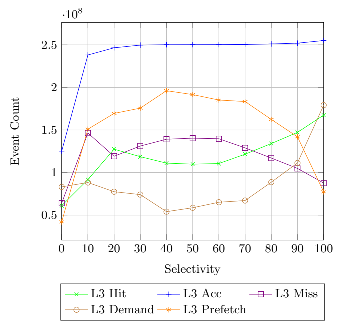
L3 cache accesses are composed of demand accesses (created by load instructions) and prefetch accesses (created by CPU prefetchers). As shown in Figure 5a, demand accesses are induced more frequently for low and high selectivities. Towards a selectivity of 50%, they decrease and increase thereafter (indicated by a dip). In contrast, prefetch accesses show an opposite trend. For high and low selectivities, less prefetches are induced by CPU prefetching units. Towards a selectivity of 50%, most prefetches are induced with falling edges to both sides. The main reason for these trends is the branch prediction which shows the same characteristics as the prefetch accesses. At 50% selectivity, most branches are mispredicted and thus many unnecessary instructions (e. g. data loads) for not taken execution paths are induced. Thus, prefetchers trigger more often and thus the number of demand accesses decreases.

In general, a demand or prefetch cache line request can either be a hit or a miss. The ratio between hits and misses depends on the temporal gap between demand and prefetch accesses as well as the branch prediction. At first, a prefetch from a mispredicted execution path induces one cache miss because its cache line is never used. In contrast, two cache misses occur if a prefetch of a useful cache line is issued either too early (evicted before used) or issued too late (not completed when accessed by demand); thus, memory bandwidth is wasted by prefetching. In the best case, a prefetcher requests a cache line timely such that the prefetch itself misses the L3 but the following demand access hits.

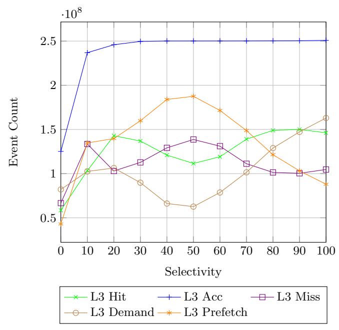
Again, hit and miss curves show contrary trends in Figure 5a with a switch point at 50% selectivity. Whereas L3 hits follow the trend of L3 demand accesses, L3 misses follow the trend of L3 prefetches. The prefetching units in modern CPUs produce these effects. In general, prefetches induce cache misses because they access tuples most probably at first. If prefetches are issued in time and from correct execution paths, only one L3 miss occurs. With increasing branch mispredictions, the number of unused prefetches increase and thus the number of cache misses. On the other hand, issuing prefetches not in time is shown in Figure 5a by sequential access to column A (0% selectivity) and sequential access to columns A and B (100% selectivity). Although branches are predicted almost always correctly, L3 misses are still induced because the memory bandwidth is overexerted. However, the number of demand accesses and hits are also high.

In Figure 5a, all counters increase steeply for a selectivity between zero and ten percent. In this selectivity range, demand and prefetch accesses as well as cache hits and misses follow the steep increase in L3 accesses, which are in turn explained by the probability of a cache line access [12]. The sharp increase of cache misses in this range follows a sharp increase in prefetching.

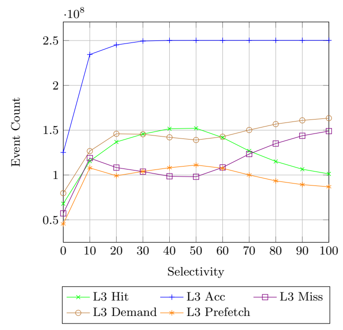
In Figure 6a, we show a detailed breakdown of demand-related L3 cache counters. As shown, demand misses are most frequent for low and high selectivities. In contrast, demand accesses often hit the cache, especially in the medium-selectivity range. This observation explains why prefetches are issued only rarely in low and high selectivity ranges, but



(a) L3 Cache (DOP 1).



(b) L3 Cache (DOP 12).



(c) L3 Cache (DOP 24).

Figure 5: L3 Cache Overview.

frequently in the medium-selectivity range. For high selectivities, the number of demand misses suddenly increases. In this case, the increased number of accesses shortens the amount of time between two accesses; thus, the prefetcher is not fast enough to prefetch each cache line access. Additionally, prefetchers decrease their efforts in this range.

Overall, prefetchers in modern CPUs perform very well for sequential access patterns which is indicated by the small gap between hits and accesses. To enable efficient prefetching, CPUs exploit two types of prefetchers [7]. First, the L1 and the L2 *streaming prefetcher* fetch the next cache line. Second, the L1 and the L2 *stride prefetcher* exploits load histories to detect and prefetch strided forward or backward loads. For a selection, sequential access to column A induces a simple streaming pattern which is well suited for these prefetchers. However, access to column B induces irregular strides, especially for medium-selectivity ranges, which results in less efficient prefetching. In Section 5 we will show, that an increased number of prefetches for a dop of one is less adverse compared to larger dops.

Finally, Figure 6a shows a detailed breakdown of prefetch-related L3 cache counters. Surprisingly, not each prefetch access results in a prefetch miss as one might expect. Prefetch hits could be induced by different prefetchers. Thus, if two prefetchers prefetch the same cache line within a specific temporal gap, the first will miss but the second will be successful (i. e. hit). Note that, L1 *line fill buffers* catch accesses to multiple tuples within the same cache line and forward only one load or prefetch request to lower cache levels (L2 and L3 cache) [7].

5. PARALLEL EXECUTION

This section examines the impact of parallel execution on selections. In Section 5.1, we show how branch-related and cache-related counters change their characteristics for different dops. Then, we present a time distribution of cycles spent in different CPU components in Section 5.2. Section 5.3 relates different run-time characteristics to performance counters. Finally, we investigate selection scalability in Section 5.4.

5.1 Degree of Parallelism

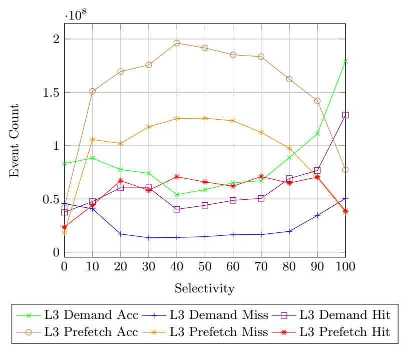
As shown in Sections 3 and 4, branch prediction and cache accesses are the major contributors to selection performance. First, branch-related counters and thus branching behavior do not depend on the number of CPUs involved in the processing because they are *instruction-dependent*. If a selection is partitioned among multiple cores and executed in parallel, the number of conditional branches, branches taken, and branches not taken do not change. Instead, branches are distributed among partitions and their sum remains equal for a selection using one or multiple cores. Instead, the branching behavior of a selection depends on the selectivity and parallelism represents an orthogonal parameter.

In contrast, cache-related counters and thus memory utilization depend on the number of CPUs involved in query processing because they are sensitive to the *memory-bandwidth*. Figure 5 shows L3 cache-related counters for different dops. Note that, CPU 1 has 12 physical and 24 logical cores and thus, starting from a dop of 12, hyper-threading is applied. In Figure 5, only L3 accesses remain constant among different dops. With increasing parallelism, three cache-related characteristics change. First, demand and prefetch accesses converge to each other. Second, more demand and less prefetch accesses are induced. Third, the correlation between hits and demand accesses as well as misses and prefetch accesses merge such that they partially overlap.

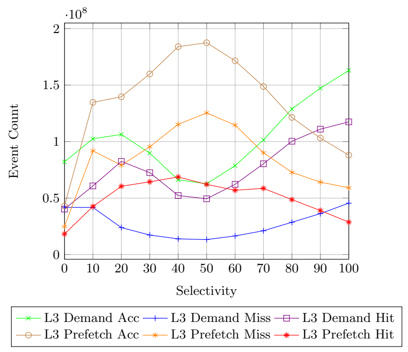
Figure 6 splits up demand and prefetch accesses as well as their induced misses and hits. Among all dops, prefetching works well such that the majority of demand accesses hit the L3 cache. However, demand accesses and hits change their trends as well as their occurrence if parallelism is applied. For small to medium dops (1 and 12 cores), less demand accesses are induced because prefetchers work more efficiently and thus cache accesses can be satisfied on higher cache levels. Typically, cache lines are brought to L2 cache unless it is *heavily* loaded with missing demand requests. Hence, prefetchers in modern CPUs are sensitive to the overall memory bandwidth and thus the number of prefetches decrease with higher memory bandwidth utilization [7]. Therefore, prefetching for low to medium dops work more efficient because they require less memory bandwidth. In particular, prefetchers increase their prefetching efforts in the medium-selectivity range for low to medium dops because memory bandwidth is available. In contrast, selections using 24 cores overexert memory bandwidth and thus less prefetches are induced. The reduced number of prefetches combined with a longer prefetching latency induced by the memory bottleneck result in more demand accesses.

5.2 Time Distribution

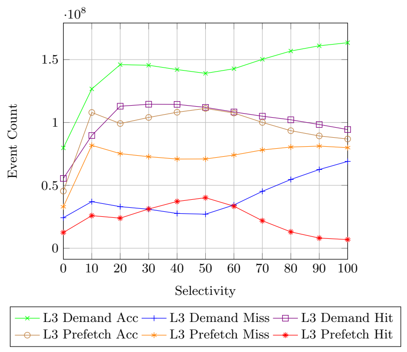
In this section, we derive a time distribution for different CPU components following the *Intel optimization guide* [6]. Intel provides special counters to monitor buffers that feed micro-ops supplied by the front end to the out-of-order back end. Using these counter, we are able to derive which CPU component stalls the CPU pipeline for how long.



(a) L3 Demand/Prefetch (DOP 1).



(b) L3 Demand/Prefetch (DOP 12).



(c) L3 Demand/Prefetch (DOP 24).

Figure 6: L3 Demand and Prefetch.

For our example query, we plot a time distribution of cycles spent in four CPU components based on the *Intel optimization guide* [6] in Figure 7. First, the *front end* delivers up to four micro-ops per cycle to the back end. If the front end stalls, the rename/allocate part of the out-of-order engine will starve and thus execution becomes *front end bound*. Second, the *back end* processes instructions issued by the front. If the back end stalls because all processing resources are occupied, the execution becomes *back end bound*. Third, with *bad speculation*, the pipeline executes speculative micro-ops that never successfully retire. This component represents the amount of work wasted by branch mispredictions. Fourth, *Retiring* refers to the amount of cycles that are actually used to execute useful instructions. This component represents the amount of useful work done by the processor.

Back end stalls can be further splitted into memory-related stall time and core-related stall time. Memory-related stall time corresponds to stalls related to the entire memory subsystem, e. g. cache misses that may cause execution starvation. In contrast, core-related stall time originates from execution starvation or non-optimal execution ports utilization, e. g. long latency instructions may serialize execution [7]. For our example query, the ratio between core stalls and memory stalls is determined by the ratio between front end and back end stalls. Front end stalls as well as core stalls predominate a selection using one core. In contrast, back end stalls and memory stalls predominate a selection using all logical cores. Due to space limitations, we omit a detailed presentation of back end stalls.

In Figure 7, for a dop of one and small selectivities, the majority of time is spent in the back end and for retiring the useful results. Thus, the processor is efficiently utilized and the memory bandwidth constitutes the limiting factor. For medium selectivities, the majority of stall time shifted towards bad speculation and front end stalls and thus a selection becomes front end bound. In general, bad speculation leads to a significant amount of wasted cycles and prevent instructions from entering the pipeline at the front end (front end pollution) [7]. Figure 7 shows this relation by the correlation between bad speculation and front end stalls. For very large selectivities, bad speculation decreases in favor for more back end stalls and the time spent for useful computation (retiring) increases. These characteristics are similar to a very low selectivity. A detailed back end analysis for a selection using one core reveals, that the back end is dominated by the core time. Thus, the CPU uses the out-of-order execution engine inefficiently for medium-selectivities. In contrast, for very high and low selectivities, the back end time is dominated by memory stalls in the cache hierarchy. To summarize, front end stalls and bad speculation prevail for a selection using one core and thus branch misprediction is the main contributor to the run-time.

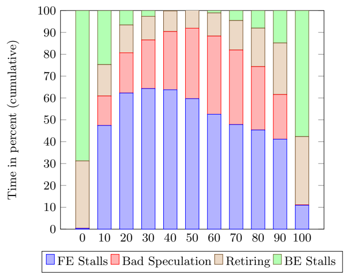
For a selection using all logical cores (24), this characteristics change completely. First, the overall time distribution shifted towards back end stalls. Second, the back end becomes predominated by memory stalls. However, the general trend for bad speculation and front end stalls remains with a peak at 50% selectivity but with a smaller portion of the overall time. As a result, back end stalls prevail for a selection using all logical cores and thus cache accesses mainly contribute to the run-time.

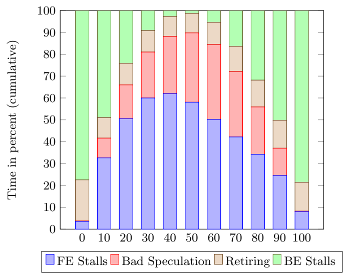
Finally, a selection using all physical cores (12) represents a middle ground between these both extremes and its main contributor to run-time depends on the selectivity. For low and high selectivity ranges (0% to 30 and 70% to 100%), the time spend in the back end increases compared to one core execution and thus cache accesses are more determining. In contrast, branch mispredictions prevail as the main contributors to run-time in the selectivity range from 40% to 60%.

5.3 Run-time Characteristics

Figure 8 presents run-times of our example query using a dop of one, 12, and 24. As shown, run-time characteristics differ largely between these dops. For a dop of one, run-time peaks at 50% with falling edges to both sides. In contrast, a selection executed by 24 logical cores exhibits a steep increase in run-time between zero and ten percent selectivity before staying constant among the remaining selectivity range up to 100%. Finally, a selectivity executed by 12 physical cores exhibits a middle ground between a dop of one and 24. Therefore, it shows the same peak at 50% selectivity with falling edges at both sides, but passes over to constant pathways very abruptly.

Our time distribution analysis in the previous section enables us to explain these different trends (see Section 5.2). In Figure 8b, we contrast run-times to performance counters which exhibit similar trends. For a dop of one, run-time follows branch mispredictions. This run-time characteristic is in line with results presented by Ross et al. [14]. In contrast, run-time for a dop of 24 follows L3 cache accesses. This run-time characteristic is in line with results presented by Pirk et al. [12]. In between these two extremes, a selection executed by 12 physical cores follows L3 cache misses.


(a) Time Distribution (DOP 1).


(b) Time Distribution (DOP 12).

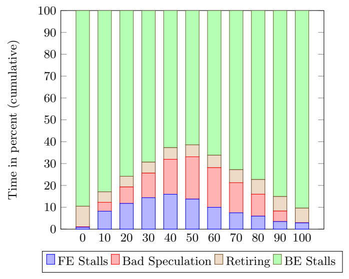
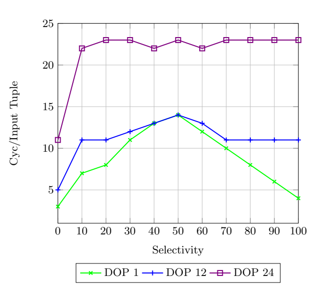

(c) Time Distribution (DOP 24).

Figure 7: Time Distribution.


(a) Cycles for different DOP.

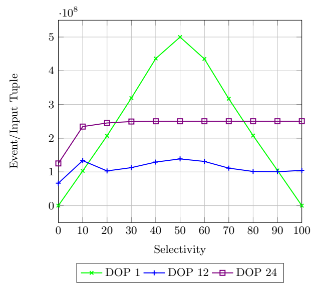

(b) Performance Counters.

Figure 8: Cycles and their Relationships.

There are two main reasons for changing run-time characteristics. First, as shown in Sections 3 and 4, branch mispredictions and L3 cache accesses are two major performance factors that determine the performance of a selection. If a selection is executed in parallel, each additional core reduces the effective bandwidth per core. We demonstrate in Figure 7a, that memory bandwidth is no limiting factor for a selection executed by one core because the bandwidth is not fully utilized. By disabling one major performance factor, the other has an increasing impact. Therefore, selections on one core are *branch prediction bound* and thus the number of introduced branch mispredictions determine the run-time. In Figure 8b, we show that branch mispredictions exhibit the same trend as the run-time of a selection using one core. In contrast, a dop of 24 overexerts memory bandwidth. Thus, a selection spent the majority of its cycles for waiting on data transfers from memory (see Figure 7c). However, branch mispredictions still occur but they can be overlapped with memory accesses. Thus, they contribute only minor to the overall run-time such that a selection using all logical cores become *memory bound*. In Figure 8b, we show that L3 cache accesses exhibit the same trend as the run-time of a selection using all logical cores.

Finally, a selection using all physical cores (12) spent less time in waiting on data than a selection using all logical cores (24) (see Figure 7b). Because processor vendors commonly align memory bandwidth to the number of physical cores [7], selections using only physical cores are more memory efficient. Thus, the curve in Figure 8b is composed of two intervals. In the first selectivity interval, from 0% to 30% and 70% to 100%, run-time is memory bound. In the second selectivity interval, from 40% to 60%, branch misprediction could not be entirely overlapped with memory accesses and thus the selection is branch prediction bound. Finally, L3 misses exhibit the same trend a selection using 12 physical cores. In sum, selections shift their run-time characteristics from a branch prediction bound to a memory bound trend with several transitional trends.

5.4 Scalability

In Figure 9, we plot run-time speed-up for a selection using different dops compared to a dop of one. Selections using two cores show a linear speed-up. Starting from a dop of four, the speed-up changes among the entire selectivity range. For example, a selection using four cores scales only linear in the selectivity range from 20% to 80%. Different speed-ups among the selectivity range in Figure 9 originate from different run-time trends shown in Figure 1. Run-time trends change with increasing parallelism from a curve with a peak and falling edges to a curve with a steep increase followed by a constant pathway. Besides different speed-up curves, selections scale non-linearly for larger dops. Using 8 and 12 physical cores, the linear speed-up is only reached for medium selectivities and this range becomes smaller with increasing dop. Finally, if hyper-threading is applied for 16, 20, and 24 logical cores, the speed-up becomes sub-linear for the entire selectivity range.

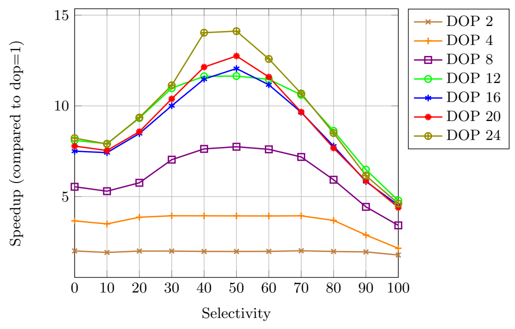


Figure 9: Speedup.

6. COST MODELS

There are two common approaches to determine costs for a selection. Ross et al. [14] determine costs based on induced branch mispredictions. They claim, that branch mispredictions are the main contributor to the run-time of a selection. In contrast, Pirk et al. [12] determine costs based on induced cache accesses because they claim that cache accesses are the main contributor.

Both cost models consist of two parts. First, they predict the number of performance impacting events according to their assumption about the main performance contributor. Then, they multiply these numbers by a penalty which estimates the costs per event. Whereas both models predict the occurrence of their performance impacting event very precisely, resulting run-time estimations are more inaccurate.

In Figure 1, we show both estimated as well as measured cycles for different dops. Although both cost models do not take parallelism into account, their estimation could not be adjusted by a scalar factor to estimate the entire range of parallel execution. Ross et al. [14] estimate cycles as well as trends more precisely for low dops because they take branch misprediction into account. As our evaluation shows, branch mispredictions are the major cost contributor for a selection using one core (see Figure 5.2). In contrast, Pirk et al. [12] estimate a different trend which fits parallel execution using high dops because they take L3 accesses into account. As our evaluation shows, L3 accesses are the major cost contributor for a selection using all logical cores (see Figure 5.2). Note, neither of these cost models take L3 misses into account and thus they misestimates a selection using all physical cores.

In summary, both cost models take only one performance factor into account and therefore fail to predict run-times correctly as well as their trends for different dops. Overall, our evaluation reveals, that characteristics of modern CPUs are too complex and interrelated such that a cost model based on one parameter is not able to model the entire range of parallelization. However, the cost model by Ross et al. [14] and Pirk et al. [12] can be used as reference points for low and high dops, respectively. We argue, that modern CPUs require a combined cost model which takes multiple characteristics into account to reflect the interrelated processor characteristics of modern CPUs.

7. RELATED WORK

Previous work sampled commercial DBMS workloads to identify the distribution between time spent for computation and time spent for waiting on data [8, 13, 1, 16]. Ailamaki et al. [1] examined four major commercial database systems. They discovered, that on average, half of the execution time is spent in stalls while 90% of the memory stalls are due to L2 data cache misses and L1 instruction cache misses. Other research show similar distributions [8, 13]. Tözün et al. [16] point out, that L1 instruction cache misses have deeper impact than data cache misses for OLTP workloads. However, most studies use old CPU generations with only two cache levels [1]. Additionally, they sample entire DBMS and do not examine the effects of parallelization and prefetching. In contrast, we analyze micro-benchmarks on the latest four Intel microarchitectures to identify the characteristics of a selection. Our results are independent of a particular DBMS implementation. Additionally, our time distributions differ greatly on the new CPU generation compared to over 15 years old CPUs used in older studies.

In the context of different scan variants, Broneske et al. [4] and RÄČducanu et al. [15] examine different implementations of a scan operator. They showed, that the best variant depends on parameters such as selectivity, data distribution, and hardware architecture. Additionally, some approaches exploit SIMD to accelerate scans [17] or introduce bit-parallelism [10]. However, these approaches compare different variants only by run-time. In contrast, we examine a basic implementation and reveal its efficiency on the current CPU generations. Furthermore, we use performance counter to reveal which CPU component contributes most to run-time.

8. CONCLUSION

In this paper, we extensively studied performance characteristic of selections on modern Intel CPUs. We showed, that branch mispredictions as well as cache accesses can be predicted by common cost models. However, these models fall short to estimate run-times for different dops. By revealing deterministic behaviors of modern CPUs, we pave the way for more accurate cost estimations in DBMS. Furthermore, our insights from an in-depth selection analysis can be used to improve cost estimations for other DBMS operators such as projections, joins, or aggregations.

In future work, we attempt to develop a cost model that takes branch mispredictions as well as cache accesses with different weighting into account. The weighting will depend on the number of executing cores and the costs per event in individual CPU components.

9. REFERENCES

[1] A. Ailamaki, et al. Dbmss on a modern processor : Where does time go ? *VLDB*, 1999.
[2] C. Balkesen, et al. Main-memory hash joins on multi-core cpus : Tuning to the underlying hardware. *ICDE*, 2013.
[3] P. Boncz, et al. Database architecture optimized for the new bottleneck: Memory access. *VLDB*, 1999.
[4] D. Broneske, et al. Database scan variants on modern cpus: A performance study.
[5] N. Hardavellas, et al. An analysis of database system performance on chip multiprocessors. *ISCA*, 2007.
[6] Intel. *Intel® 64 and IA-32 Architectures Optimization Manual*. 2012.
[7] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2012.
[8] K. Keeton, et al. Performance characterization of a quad pentium pro smp using oltp workloads. *ISCA*, 1998.
[9] V. Leis, et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. *SIGMOD*, 2014.
[10] Y. Li and J. M. Patel. Bitweaving. *SIGMOD*, 2013.
[11] S. Manegold, et al. Generic database cost models for hierarchical memory systems. *VLDB*, 2002.
[12] H. Pirk, et al. Cpu and cache efficient management of memory-resident databases. *IEEE*, 2013.
[13] P. Ranganathan and S. Adve. Performance of database workloads on shared-memory systems with out-of-order processors. *ASPLOS*, 1998.
[14] K. A. Ross. Selection conditions in main memory. *TODS*, 2004.
[15] B. RÄČducanu, et al., P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. *SIGMOD*, 2013.
[16] P. Tözün, B. Gold, and A. Ailamaki. Oltp in wonderland: Where do cache misses come from in major oltp components? In *DaMoN*, 2013.
[17] T. Willhalm, et al. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *VLDB*, 2009.
[18] J. Zhou, et al. Improving database performance on simultaneous multithreading processors.