Abstract—Merging is a building block for many computational domains. In this work we consider the relationship between merging, branch predictors, and input data dependency. Branch predictors are ubiquitous in modern processors as they are useful for many high performance computing applications. While it is well known that the performance and the branch prediction accuracy go hand-in-hand, these have not been studied in the context of merging. We thoroughly test merging using multiple input array sizes and values using the same code and compile optimizations. As the number of possible keys increase, so do the number of branch mis-predictions - resulting in reduced performance. The reduction in performance can be as much as 5X. We explain this phenomenon using a visualization technique called Merge Path that intuitively shows this. We support this visualization approach with modeling, thorough testing, and analysis on multiple systems.

Index Terms—Performance evaluation; Performance analysis; Merging; Sorting;

I. INTRODUCTION

Merging sorted arrays is a building block for many algorithms and applications including sorting, database query joins, and graph contractions. Merging is similar to set intersection which has many applications found in graph theory including for computing clustering-coefficients [1].

In this work, we show that the merge algorithm and branch predictors “do not get along” in many cases and that this conflict significantly reduces performance. When the branch predictor is considerably accurate, the performance is good due to out of order execution which is supported by many modern architectures. However, when the branch prediction accuracy rate is low, the performance goes down significantly.

We show that merging, which seems like a simple and straightforward algorithm is in fact highly data-dependent. This data dependency causes the branch predictor (on multiple systems) to predict with a low level of certainty the outcome of the branch and in practice makes off-line analysis significantly more challenging. One outcome of our work suggests that load-balancing of merging is more challenging than was previously thought: even if each core receives an equal amount of work (i.e. the same number of elements to merge), one core can become the execution bottleneck is it may take upto 4X-5X more time to merge the same number of elements due to branch mis-predictions. The actual performance for a parallel merge will be based on the distribution of the keys. The reader is referred to [2], [3], [4], [5], [6] for additional reading on parallel merge algorithm. Note, that none of these take into consideration the number of different keys in the input.

The key contribution of this work is to explain the why the operation of merging of two different inputs can take a different amount of time despite having an equal number of comparisons. While this might seem straightforward, the results are surprising and offer new insights for creating better merging and sorting algorithms. We show that as the level of “randomness” increases so do the number of branch misses increases which in turn reduces the overall performance.

While the focus of this work is to explain why merging and branch predictors do not get along, we also introduce a simple workaround the branch-predictor - a branch-avoiding merging algorithm. This branch-avoiding algorithm is immune to the randomness of the data as it is not dependent on the result of the branch predictor. The approach that we present in this paper uses additional arithmetical operations. We extend the discussion in future sections.

In this work, we study the impact of branch prediction with the help of Merge Path[4], [5] - a visual approach to merging. Using Merge Path, we can simplify the performance analysis of merge. The remainder of this section introduces Merge Path and explains the significance of the input’s key distribution. Section 2 discusses related works. Section 3 presents our experimental setup and empirical results. In Section 4 we present some final thoughts.

A. Problem Statement

Given two sorted arrays $A$ and $B$ of length $|A|$ and $|B|$, respectively, the output of a merge is the array $C$ of length $|C|$ such that $C$ is sorted and made up of all the elements of $A$ and $B$ where $|C| = |A| + |B|$. Simplified pseudo code of the algorithm for doing a merge can be found in Alg. 1 and an extended discussion can be found in [7].

B. Merge Path

Merge Path [5], [4] restates the merging operation to be considered as a traversal of a 2D grid of size $|A| \times |B|$. This grid is known as the Merge Path matrix. The traversal starts at the top-left corner and finishes at the bottom right corner, Fig. 1(a). Array $A$ has been placed as a column stack on the left of the matrix and array $B$ has been placed over the top of the matrix. The only legal moves are to the right (when $B[b_i] < A[a_i]$) or downwards (when $B[b_i] \geq A[a_i]$), where $a_i$ and $b_i$ are the indices of the elements being merged in both arrays. The decision per move is based on the values of $A$ and $B$ at the given indices. Essentially, the order in which elements are merged is equivalent to the traversal of the path. The path is discovered sequentially in the process of the merge. We will further discuss the use of the path in the context of the number of possible keys in the following sections. Note, that Merge
Algorithm 1: Serial merge.

\[
\begin{aligned}
& a, \ b, \ c, \ \text{\leftarrow} \ 0; \ c_i \ \text{\leftarrow} \ 0 \\
& \text{while } a_i \ < \ |A| \ \text{and} \ b_i \ < \ |B| \ \text{do} \\
& \quad \text{if } A[a_i] \ < \ B[b_i] \ \text{then} \\
& \quad \quad C[c_i++] \ \leftarrow \ A[a_i++] \\
& \quad \quad \text{else} \\
& \quad \quad C[c_i++] \ \leftarrow \ B[b_i++] \\
& \quad \text{\} \ \text{copy remaining elements into C}
\end{aligned}
\]

Path was originally used for load-balancing a parallel merge. In this paper, Merge Path is used for a different purpose - to explain why merging is tough on branch predictors.

In Fig. 1(b), a Merge Path matrix with alternate paths is shown. We have added to the path from Fig. 1 (a) three additional paths - dotted, dashed, and solid. Each one of these paths represents a different input with a different set of keys.

C. Key Distribution

The performance of algorithms in many cases is input dependent - this is also the case for merging where the number of the different keys to be merged has a big role to play. As such, we need a way to quantify the input key distribution.

Consider two input arrays A and B and output array C, such that \(|A| = |B| = L\) and \(|C| = 2 \cdot L\). Elements are uniform randomly picked from the range \([0, 1, 2, \ldots, N - 1]\) where N is largest possible key into the input arrays A and B. These are then merged into the output array C. The probability of any particular key occurring in a particular input array is \(L/N\).

For example, keeping \(L\) constant at 1000, \(N = 100\) means that on average each key will appear \(L/N = 10\) times. For \(N = 1000\) each key will appear on average \(L/N = 1\) time. In Merge Path terms, the more the repetitions of a key, the longer its segment will be “straight”. From a branch predictor’s perspective, longer segments allow for better prediction as the segments are directly correlated to the \(TAKEN\) or \(NOT\_TAKEN\) states of the predictor.

Consider \(N = 2\) and \(L = 8\), the 8 elements of each array are made up of all 0’s and 1’s assuming a uniform random distribution. Since the elements are randomly picked and then sorted, it is quite likely that the first 4 elements are 0’s and the next 4 elements are 1’s. For simplicity assume that this is the case for both the arrays. This path is shown as a dashed line in Fig. 1(b).

When \(N = 8\) and \(L = 8\), on average it is expected that each array will consist of 8 different values. As such the path will change more rapidly as depicted by the solid line. This path is a finer grained staircase than the case of \(N = 2\). The dotted path depicts the case for \(N = 1\) and \(L = 8\). In this case the different arrays consist of a single value.

For good performance, a branch predictor must deal well with all these patterns and many more. In truth, many branch predictors will deal well with “steps” that follow some deterministic pattern; however when a little randomness is added these no longer can predict the path.

While we discussed in this subsection the construction of the path assuming a uniform distribution, in the later sections it will become apparent that different distributions can be categorized and explained using similar concept. For example, when a Gaussian distribution is used for generating the keys, it is likely that the their will be many keys that are in the proximity of the mean and few that are distanced from the mean. From the Merge Path perspective this will result with few and long segments in proximity to the mean. As such this is a private case for low-key distribution of the uniform distribution.
II. RELATED WORKS

A. Merging and Sorting

Merging and sorting are building blocks for many algorithms and applications. In [8] a merging algorithm for the GPU is shown that is based off the Merge Path algorithm. In [5], [4] the first cache-aware merging algorithm is given. In [9] two sorting algorithms for the GPU are shown, radix sort and merge sort. In [10] a cache oblivious sorting algorithm is presented.

Bitonic sort [11], also known as Batcher’s algorithm, is a parallel approach for sorting using a network with a depth of \( \log^2(n) \) steps. While bitonic sort has sub-optimal time complexity it has a benefit of being highly scalable and is suitable for both sorting and merging. In [12] a parallel merge-sorting algorithm is given that uses a 4x4 bitonic network per thread for merging. The 4x4 bitonic network is implemented using SIMD instructions, specifically Intel’s SSE instruction set. While SIMD instructions do not effect the hardware flags that are used by the branch-predictor, the SIMD approach of [12] still requires branches in the last phase of each iteration. This means that for each 4x4 bitonic network (8 inputs), only 4 elements are merged in a single iteration.

In [13], a SIMD set-intersection implementation is presented. This implementation uses Intel’s SSE instructions. This implementation can intersect up to 8 2-byte values at once. Similar to the bitonic sort implementation of [12], this implementation is also dependent on branches for pointer movement at the end of each iteration. Unfortunately, the approach of [13] cannot be adapted to merging as the given SSE instructions only find common values in the two inputs and do not copy values in the proper order. Further, many applications require merging with keys that are larger than \( 2^36 - 1 \) which is the limit of a 2-byte variable.

In [14] several different parallel merging algorithms are compared. Each algorithm is tested with multiple inputs, where each input uses a different number of keys. They refer to this as entropy. They show that as the entropy decreases the performance increases. In their work, only the number of keys are considered - the initial placement of the values in the unsorted array is not considered and this too can have an impact on the performance as we show in this work.

B. Branch Predictors

One of the classic works to consider branch prediction is due to Smith [15]. Smith reviews several types of branch predictor strategies that are based on the type of the branch: conditional, non-conditional, forward, . The 2-bit state machine branch predictor is discussed as well, including the four states: STRONGLY-TAKEN, WEAKLY-TAKEN, WEAKLY-NOT-TAKEN, STRONGLY-NOT-TAKEN. See Fig. 2 for a schema of a 2-bit state machine branch predictor. Smith also uses a sorting benchmark and shows that predicting for sorting is considerably difficult. Lee and Smith [16] introduce the Branch Target Buffer. Yeh and Patt [17] [18] introduce a two-level branch predictor that uses a global history shift register and allows for better branch prediction. In [19], the authors suggests a set of synthetic micro benchmarks for figuring out the type of branch predictor available in older Intel Pentium systems.

Many changes have been made to the the classic branch predictor: the Agree predictor [20], the Bi-Mode predictor [21], and YAGs [22]). While these predictors have improved performance greatly for some applications, we empirically show that for applications such as merging and intersection these predictors do not improve performance. Further, little is known about the actual implementation of the branch predictor in processors as vendors do not clearly state which predictor is implemented as part of a system. Ongoing empirical research tries to demystify these implementations [19], [23] using synthetic benchmark that test the branch predictor. Further, many branch predictors focus on optimizing performance for regular type algorithms such the SPECint2006 and SPECfp2006 benchmarks.

In [24] an experimental study of sorting and branch predictors is given. Using the SimpleScalar simulator they check the performance of multiple sorting algorithms for the two-bit branch predictor. They show empirically the average number of branch misses for each key. The results of our paper explain at a finer granularity the number of branch misses required for sorting. For merge-sort, each key is accessed \( O(\log(n)) \) times; as such, dividing the average number of branch misses by the number of time a key is accessed results in average number of branch misses per access. Our work confirms these results and gives a clear model that explains the branch mis-prediction rates that further clarifies their model.

III. EMPIRICAL Merging Results

A. Experimental Setup

To verify the impact of the key distribution on the execution time, we implemented the merge algorithm from Alg. 1 using the C language and the gcc compiler. We discuss two additional merging implementations: branch-avoiding (Section III-D and bitonic (Section III-E). For our experiments we used multiple systems , Table I. We used additional systems for testing - all of which offered similar results. Note that each test case was run multiple times to get better timing accuracy. For our experiments a single core is used. The outcome of our analysis is also relevant parallel merging algorithm [3], [2], [4], [5]. These parallel algorithms have two stages: partitioning (work-load balancing) followed by a sequential merge on each core. As such the findings of this paper is relevant for these sequential merges on multiple cores. We discuss this in Section III-F.
In all experiments the two arrays are of equal size. The different size arrays used are: \(0.1M, 1M, 10M, \) and \(100M\). The arrays are initially filled with randomly generating numbers from a uniform distribution from the range of \([0, 1, \ldots, N - 1]\) where \(N\) represents the number of possible keys. After generating the arrays, the arrays are sorted. The array generation and sorting are not timed. Finally the arrays are merged - this is timed.

To access the hardware branch prediction counters we used both PAPI [25] and KCacheGrind [26]. We report the results only from PAPI, yet note that the results from KCacheGrind are nearly identical. The slight difference is due to the randomness of the data. In addition to the PAPI and KCacheGrind, we also simulate (via software) a simple 2-bit state machine for predicting the IF statement of the merge algorithm (Alg. 1). Most branch predictors should easily predict the WHILE statement - as such this statement is of little interest. We will see that the software simulated 2-bit state machine gives similar results to the actual hardware-implemented predictors. This means that the advances made to branch predictor have little impact on the performance for merging algorithms.

### B. Empirical Results

Using the Merge Path scheme, we address the question on how significant is the path granularity on performance. To be precise, what is the impact of merge segments such as “right-down-right” or “down-right-down” on the total performance of the algorithm? How well does the branch predictor deal with these? We refer to “right-down-right” or “down-right-down” as “transition-swapping” as one can think of the 2-bit state machine and how the states of the branch predictor transition between the states due to the path granularity.

For all the subfigures of Fig. 3 and Fig. 4, the abscissa is the number of possible keys in log-base. For most of the subfigures we show an average value per merged element. This allows us to place multiples in the same figure within the same range. Fig. 3 (a) and Fig. 4 depicts the average time to merge a single element. All the subfigures in Fig. 3 are for the Sandy Bridge system.

Fig. 3 (b) depicts the average number of branch miss predictions per element using a software simulated 2-bit state machine. This average takes into consideration only the branches of the IF statement of Algorithm 1. The branches of the WHILE statement are not considered as these are normally taken. This does not change the qualitative results - from a quantitative perspective the difference would be in the normalization of what is “average”.

In Fig. 3 (c) we do take into consideration the branches of the WHILE statement. This subfigure depicts three curves for merging \(10M\) elements (all average values). These curves are as follows: 1) diamond markers - represents the miss rate of the software simulated 2-bit state machine for the IF statement, 2) circle markers - conditional branch misses per element of the standard algorithm as counted by PAPI divided by number (this represents the IF statement, and 3) triangle markers - the ratio of conditional branch miss out of all the branches (conditional and unconditional) as counted by PAPI - this last curve in fact represents the actual miss rate of the entire algorithm.

As vendors do not state which branch predictors are implemented for a given system [23], [19], we do not know if the misses represented by the second curve are attributed to the IF statement or the WHILE statement. Ideally, the branch misses should be for the IF statement, yet two different hardware implementation issues make it hard to be certain. The first issue is the compiler, while the C code has two branches, it is possible that the generated code will have more than two branches. We saw this in multiple inspections of the assembly code. The second is related to the global branch history - the misses could possibly be attributed to the WHILE. While predicting the WHILE statement is “easy”, the merge path of the IF statement can cause multiple 2-bit state machines to be inaccurate. Regardless, the average number of misses is considerably high for both the normalized (triangle curve) and non-normalized (circle curve) cases. Note that the simple 2-bit state machine and actual branch predictor overlap considerably in the average number of misses.

Given an array of length \(L\) with \(N\) possible keys: if \(N << L\) then the path is likely to be coarse and there will be few missed predictions. As \(N\) grows and gets nearer to \(L\), the number of keys in each array will also increase. This leads to finer paths. As the number of possible keys increases beyond \(L\), it is expected that the path will continue to be fine grained, but the path will not be a “perfect” stair-case as the uniform random generator might create paths such as \(R, R, D, R, D, D, R, D, D, R, R\). Fig. 3(d) presents the average transition-swapping per element using the 2-bit state machine. The knees in this subfigure start for a larger key distribution that the respective curves in Fig. 3 (b) than the step of the branch misses. This is due to the requirement that the number of different keys should be greater or equal than the number of elements in the input arrays.

\[R\] stands for right and \(D\) stands for down.

---

TABLE I  
**X86 systems used for testing. The results in Fig. 3 are for the Sandy Bridge system.**

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Processor</th>
<th>Microprocessor</th>
<th>Frequency</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>2600K</td>
<td>17 Sandy Bridge</td>
<td>3.4 GHz</td>
<td>32KB</td>
<td>8MB</td>
</tr>
<tr>
<td>Intel</td>
<td>4770K</td>
<td>17 Haswell</td>
<td>3.4 GHz</td>
<td>32KB</td>
<td>8MB</td>
</tr>
<tr>
<td>Intel</td>
<td>1610QMM</td>
<td>17 Ivy-Bridge</td>
<td>2.3 GHz</td>
<td>32KB</td>
<td>6MB</td>
</tr>
<tr>
<td>AMD</td>
<td>10-7850K</td>
<td>Kaveri</td>
<td>3.7 GHz</td>
<td>16KB</td>
<td>4MB</td>
</tr>
<tr>
<td>ARM</td>
<td>Cortex A-15</td>
<td></td>
<td>1.7 GHz</td>
<td>32KB</td>
<td>4MB</td>
</tr>
</tbody>
</table>
Fig. 3. (a) Average time per element for the standard merge (Alg. 1) for Sandy Bridge system (See Table ??). (b) Average number of misses per element using simulated 2-state machine branch predictor. (c) Average number of misses per element using simulated 2-state machine branch predictor and hardware counters read by PAPI [25]. (d) Average number of “transition-swaps” per element. (e) Average time per element for the branch-avoiding merge (Alg. 2). (f) Comparison of standard and branch-avoiding algorithm for 1M elements.

Algorithm 2: Branchless serial merge - assuming 32-bit data.

\[
\begin{align*}
& a_i \leftarrow 0; b_i \leftarrow 0; c_i \leftarrow 0 \\
& \text{while } a_i < |A| \text{ and } b_i < |B| \text{ do} \\
& \quad \text{flag} \leftarrow \text{Right Shift}(A[a_i] \cdot B[b_i], 31) \\
& \quad C[c_i + \ast] \leftarrow (\text{flag}) \cdot A[a_i] + (1 - \text{flag}) \cdot B[b_i] \\
& \quad a_i \leftarrow a_i + \text{flag} \\
& \quad b_i \leftarrow b_i + (1 - \text{flag}) \\
& \end{align*}
\]

// copy remaining elements into C

C. Important observations from Fig. 3

1) Note that multiple subfigures in Fig. 3 have the same structure step-like structure for all sizes. The “knee” points are at different locations and dependent on the size of the input arrays.

2) Recall that the arrays were created with a uniform random distribution. With a very low probability will the following path be generated: R, D, R, D, R, D, R, D... will be generated. In practice, we never had such a path come out of the generator. For testing purposes, we did create such a path where \( A = B = [0, 1, 2, \ldots, |L| - 1] \) (for multiple array sizes) and checked the performance of this. Not surprisingly, the branch predictors did a good job in predicting the branches. The execution time was 20% above the optimal execution time (small number of keys). Unfortunately, merging does not typically give these easy predictions patterns.

\(^2\)See Section 1.C for extended discussion.
3) In the transition phase of Fig. 3 (c), moving from a small number of keys to a large number of keys, the 2-bit state machine software simulated over-estimates that number of misses as it does not benefit from global history and pattern learning hardware implemented in modern branch predictors. Note that the over-estimation is not significant.

4) The average number of branch misses per element for large key distributions is 1/2 according to Fig. 3 (c) for both the simulated 2-bit predictor and actual predictor only considering the IF statement (and ignoring the WHILE statement). This is important for merge-sorting. We extend discussion on this in Section III-G.

D. Branch-Avoiding Merging

Due to the high variability of the serial merge, we implemented a branch-avoiding merge algorithm, Algorithm 2. The key difference between Algorithm 1 and Algorithm 2 is that instead of using the IF statement, we compute the difference of \( A[a_i] - B[b_i] \) and store it as a flag value. If the difference is smaller than zero, this means that \( A[a_i] \) is smaller and the sign of the difference will be 1, otherwise the sign will be 0. The sign, denoted as \( \text{flag} \) in the pseudo code, allows to copy the correct value and increase the corresponding index using arithmetic operations.

Note, some compilers might replace the IF statement with a conditional move instruction. Such an optimization is compiler based, flag based, and is not consistent. When the compiler is successful, then the knees in the subfigures can not be seen. This branch-avoiding algorithm behaves the same as a function of the number of possible keys.

Fig. 3 (e) depicts the average time to merge elements using the branch-avoiding algorithm. The average time per element for the branch-avoiding algorithm is near constant and is independent of the key distribution. This explains the near constant curves. Note, that the branch avoiding algorithm requires a shift operation, two multiplications, six addition operations, and several additional store operations in comparison with the branch-based algorithm that requires simple a branch, a store, and two index increment operations. Even with the increase in the number of arithmetic operations, the branch-based algorithm can outperform the branch-based algorithm.

In Fig. 4 the different system merges equal size arrays with \( 1M \) elements. We do not show all the curves for all array sizes as these repetitive. Note, the straight-forward algorithm and the branch-avoiding algorithm behave similarly on different architectures.

E. Bitonic Sort Using SIMD

In addition to the branch-based and branch-avoiding algorithms, we implemented the SIMD algorithm of [13]). This algorithm uses the Intel SSE instruction set. We make note of this as the SSE implementation is not portable across all architectures. Specifically, the ARM based system that we used could not make use of it. The limited portability of this
implementation means that a SIMD algorithm needs to be implemented for each architecture using its specific instruction set. The SSE implements a 4x4 bitonic network as described in [12]. This network will be used iteratively as the sorted arrays are traversed. In the initial phase, each array inputs 4 elements (8 in total). The output of the network is 8 sorted elements. Of those 8 elements, only 4 can be used and are copied to the output array. The remaining 4 elements are used in the next iteration of the bitonic network. The last phase of each iteration requires fetching 4 additional elements from either A or B - this will be used in the next iteration of the algorithm. This decision uses a branch and as such, the SIMD bitonic algorithm is prone to branch mispredictions, especially for a large number of keys. This is depicted in Fig. 3 (f) and Fig. 4.

In comparison with the two C-based implementations there are two significant differences worth noting in terms of performance: 1) the SSE-based implementation is a “low-level” and optimized whereas the C-based implementations are not implemented assembly and 2) The C-based approach was highly compiler and compiler-flag dependent - the SSE based approach gave nearly consistent times regardless of the flags. We did not implement the C-based algorithms in assembly as this would limit their portability to a specific ISA. Further, this was not the focus of this work.

Fig. 3 (f) and Fig. 4 compare the times of the standard algorithm, branch-avoiding algorithm, and bitonic-sort (when applicable). The bitonic sort outperforms the other algorithms as it merges four elements in each iteration of the algorithm. This means that that are 4-times fewer iterations. Resulting in 4-times fewer branches. From these figures, we can see that the performance of the SSE based implementation is also branch dependent. The difference between a small number of keys and a large number of keys can be as high as 40%.

In summary, the benefit of the SIMD algorithm is two-folds: merging several elements concurrently and reducing the number of branches. The limiting factors of the SSE bitonic implementation is: 1) the implementation is architecture dependent and as different systems also have ISAs, this would require multiple implementations, 2) not all systems support SIMD instructions, and 3) SIMD programming is not trivial and typically requires an expertise that most programmers do not have.

F. Parallel Merging

In this paper we focused on the performance dependency of merging with respect to the input data. Our experimentations focused on single-core systems, yet are findings are broader and relevant to parallel merging algorithms. Consider a parallel merge with two cores and the path example in Fig. 5. The first core receives the elements in the dashed path and the second-core receives the elements of the solid path. Recall, that these paths are not known at run-time and will be found out in the process of merging. As a result of the input data dependency of merging, the core responsible for the dashed line will benefit from a good prediction rate while the core responsible for the solid line will take more time - up to 5X more. This means that even if perfect load-balancing is achieved in terms of the number of elements that each core receives, the actual load-balancing maybe off due to the data dependency.

G. Branch Predictors in MergeSort

The results of this paper align with those found in [24] for merge-sort. In [24], an emphasis was placed on an quantitative study of the number of branch misses per key when sorting an array using merge-sort and additional sorts. The results of our current work is a qualitative study that supplements and explains those results in a more accurate fashion.

When sorting an array of size \( L \) using merge-sort, each key is accessed \( \log(L) \) times for each of the \( \log(L) \) iterations. The worst case scenario where there are a large number of keys will give an upper bound of \( 0.5 \cdot \log(L) \) misses per element, where 0.5 represents the worst-case empirical probability of a branch miss in each iteration. Intuitively, think about the first iterations of the merge-sort algorithm where the array sizes are small and the number of keys are large (discussed in Section I-C).

Example, when \( L = 4096 \), there will be 12 iterations where each element is accessed. Therefore, the branch mispredictions per element for the entire merge-sort will be \( 1/2 \cdot \log(4096) = 6 \) - this is also the number reported in [24] for their quantitative study.

IV. Conclusions

In this paper we showed empirically that the performance of merging is dependent on the implementation (branch-based vs. branch-avoiding) and the key distribution. While merging may seem like a simple and straightforward algorithm, we show that it has an irregular behavior. To the best of the authors knowledge this is the first paper that takes in depth look at the relationship between merging, data dependency, and branch predictors. The significance of our results can help design faster queries for data bases, better performing sorting
algorithms, and help new intersection algorithms for graph contractions and graph triangle counting.

For the branch-based algorithm, when there are few keys in the input data, there is good performance due to the high prediction rate of the branch predictor. However, when the number of possible keys is large, the branch predictor fails and brings with it a high performance penalty (in comparison with the workload requirement of the algorithm). The difference in the performance between a high prediction rate and a lot prediction can be as high $3X - 5X$ and is system dependent. We showed that a simple 2-bit branch predictor can accurately represent the actual number of branch misses and confirmed the accuracy of this model by comparing it with hardware branch-prediction counters.

On the other-hand, we showed the a simple branch-avoiding algorithm has near constant performance on all systems and for all input, meaning that the performance is invariant to the input. For a small number of keys, this approach adds overhead to the runtime and makes it slower than the high-speed branch-based algorithm. For a large number of keys, this approach outperforms the branch-based approach. The exact cutoff point when the branch-based algorithm is preferable over the branch-avoiding algorithm is system dependent as performance of the branch-avoiding algorithm is dependent on the ALUs of that system. Nonetheless, the cutoff point is always near the transition point of a small number of keys to a large number of keys in comparison with the array size. Our branch-avoiding algorithm is a simple and naive implementation. There are additional optimizations that can be applied to this approach such as the use of conditional moves or rewriting the code at a low-level assembly as was done for the SIMD implementation. In some cases, these optimizations can limit the portability of the code from one system to the other. Further, it is unlikely that a non HPC expert will want to apply these optimizations, whereas, the approach taken in this paper is simple and straightforward.

It seems that a hybrid approach between the branch-based and branch-avoiding algorithms would be desirable as such an implementation would benefit from the best of both worlds. How to create the algorithm raises many interesting questions and ideas. Would this be compiler based? Could the OS monitor the hardware counters available in the system and detect when each of the algorithms would be preferable? Or perhaps, could a simple sample technique of the data help recognize if there is a small or large number of keys. We leave these as open-ended questions and encourage others to continue this research.

ACKNOWLEDGMENT

The author would like to thank Aakash Goel and Richard Vuduc for their lengthy discussions and useful pointers throughout this work.

REFERENCES


