# Analysis of Memory Sensitive SPEC CPU2006 Integer Benchmarks for Big Data Benchmarking

Kathlene Hurt and Eugene John
Department of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, United States
kathlene.hurt@gmail.com, eugene.john@utsa.edu

## ABSTRACT
Benchmarking for Big Data is done at the system level, but with processors now being designed specifically for Cloud Computing and Big Data applications, optimization can now be done at the node level. The purpose of this work is to analyze three SPEC CPU2006 Integer benchmarks (libquantum, h264ref and hmmer) that were deemed "highly memory sensitive" in other works to determine their potential as Big Data processor benchmarks. Program characteristics like instruction count, instruction mix, locality, and memory footprint were analyzed. Through this preliminary analysis, these benchmarks were determined to be potential Big Data node-level benchmarks, but more analysis will have to be done in future work.

## General Terms
Performance.

## Keywords
SPEC, cache, memory, Big Data, benchmarks.

## 1. INTRODUCTION
Big Data systems introduce many new challenges for system level benchmarking. Big Data benchmarking generally involves processing unstructured data using Hadoop or NoSQL across all the nodes of a system. However, processors are now being designed specifically for Cloud Computing and Big Data applications, like IBM's POWER8 processor [9], but Big Data workloads cannot be used for benchmarking at the node or processor level. This introduces a need for benchmarks for processor design that simulate the behavior of a Big Data systems, and have the characteristics of Big Data workloads [11], like large memory footprints and memory parallelism [8]. It is possible that many benchmarks already exist with these characteristics.

In [2], the SPEC CPU2006 benchmark suite, a common benchmark suite used in both academia and industry, was analyzed to find the similarities and redundancies between each of the programs. The work also explore the benchmarks' sensitivity to certain performance characteristics, changes from previous releases of the suite, and similarities between input sets. Of the 12 integer SPEC benchmarks, three were determined to be highly sensitive to microarchitectural memory changes. The purpose of this work is to discover what program characteristics make these benchmarks more sensitive to memory changes than the rest of the SPEC CPU2006 integer benchmark suite, and whether they exhibit characteristics that would make them good candidates for Big Data processor benchmarking. The instruction count, instruction mix, locality, and memory footprint of each program will be analyzed. The floating point benchmarks will not be considered. Section 2 will give a brief overview of the behavior of Big Data workloads. Section 3 will outline the details of how this memory sensitivity ranking was determined. Section 4 will describe the three highly sensitive benchmarks (libquantum, h264ref. and hmmer) in more detail and describe their input sets. In Section 5, the dynamic instruction count and instruction mix of each benchmark will be analyzed. In Section 6, the locality characteristics of each benchmark will be analyzed. In Section 7, the cache behavior, with a focus on working sets, will be analyzed. Section 8 will summarize the findings of this work, and Sections 9 will be the concluding remarks and applications of this work. Sections 10 and 11 contain acknowledgments and references, respectively.


*PABS'15*, February 1, 2015, Austin, TX, USA.

http://dx.doi.org/10.1145/2694730.2694732

## 2. BIG DATA WORKLOAD BEHAVIOR
To determine whether these three SPEC benchmarks are suitable for node-level Big Data benchmarking, a description of Big Data benchmark behavior is necessary. Though this work will not make any firm conclusions about their relationship, a more in-depth analysis will be completed in future work.

At the node-level, Big Data workloads have very large memory footprints, with some common Big Data benchmarks having memory footprints between 10GB and 100GB [8][10]. They also have high memory bandwidths, generally between 20 GB/s and 40 GB/s [8]. Some works have considered Big Data benchmarks to have extremely cache friendly behavior, observing cache hit rates around 80% in a 1GB cache [8]. This implies that a small percentage of their memory footprint is re-referenced frequently, which means these Big Data benchmarks have strong locality.

## 3. L1 D-CACHE SENSITIVITY RANKING
In [2], SPEC CPU2006 benchmarks were ranked based on their sensitivity to L1 data cache (D-cache) configuration changes. Simulations were run on five different machines with various compiler and instruction set architectures to reduce the bias due to program characteristics that are micro-architecturally dependant. The programs were then ranked by the variance of their L1 D-cache miss rates across the five machines. Programs with a high

variance were considered highly sensitive to L1 D-cache characteristics, while programs with little to no variance were considered not sensitive. The microarchitectural details of the five machines has not been made available due to the researchers' confidentiality agreement with SPEC. Table 1 summarizes the resulting classifications of the 43 benchmark and input set combinations from the work in [2]. The benchmark relevant to this research have been bolded.

Many of the benchmarks have multiple input sets. For some benchmarks, like gcc and h264ref, the behavior of the program can be strongly affected by the input set, so multiple input sets are included to simulate a wide range of behavior. For other benchmarks, like libquantum and astar, the input does not strongly affect the program characteristics, and therefore, only one input is required. A reportable SPEC result for each benchmark must include all of its input sets, but researchers often only use one input set [2].

In Table 1, the various input sets are indicated by a number following the benchmark name. For example, the hmmer benchmark has two inputs: hmmer-1 and hmmer-2. Three benchmarks, for a total of four benchmark/input set combinations, have high L1 D-cache sensitivity: libquantum, h264ref-2, h264ref-3, and hmmer-1. Two of the three input sets to h264ref (h264ref-2 and h264ref-3) and one of the two input set of hmmer (hmmer-1) were considered highly sensitive. The other input set of hmmer (hmmer-2) has a medium sensitivity. The third input set of h264ref (h264ref-1) has low sensitivity. The effect of the different inputs on the program behavior will be further explored in the following sections.

**Table 1. Sensitivity of SPEC CPU2006 Integer Programs to L1 D-Cache Miss-Rate**

| L1 D-cache Sensitivty | Benchmarks |
|---|---|
| High | **libquantum, h264ref-2, h264ref-3, hmmer-1** |
| Medium | **hmmer-2,** perlbench-2, perlbench-3, gobmk-3, gcc-7 |
| Low | gcc-8, xalancbmk, astar-2, perlbench-1, astar, **h264ref-1**, gobmk, astar-1, gobmk-4, omnetpp, mcf, gcc-9, gcc-3, gobmk-2, bzip2-3, bzip2-5, gobmk-1, gcc-6, gcc-5,bzip2-2, bzip2-6, gcc-2, gcc-1, bzip2-1, bzip2, gcc-4, bzip2-4, gobmk-5, sjeng |

# 4. BENCHMARK SYNOPSES

All of the memory sensitive benchmarks are integer benchmarks in the SPEC CPU2006 benchmark suite. These benchmarks have a wide range of modern application areas including quantum computing, Big Data, and biomedical research. They are all written in C, and each have static instruction counts in the few trillion. The following subsections describe the application area, general behavior, and excepted input for each of the benchmarks. For hmmer and h264ref, the various inputs will be compared.

## 4.1 libquantum

Libquantum, short for "library quantum", is a C library that simulates a quantum computer. The quantum computer implements an algorithm for the factorization of numbers called *Shor's factorization*. It is able to model quantum registers, some quantum gates, and decoherence, which is one of the many obstacles in fully realizing quantum computers. The program accepts the number to be factorized as an input, and outputs the factors of the number, or an error if the factorization was unsuccessful. Unlike h264ref, and hmmer, libquantum only has one input defined by the SPEC benchmark suite.

## 4.2 h264ref

The h264ref benchmark simulates a modern video compression standard used in Blu-ray Discs and video broadcasting. The benchmark varies marginally from the original source code, with only small changes made to ensure fairness across different machines. The program accepts raw video data in YUV format and a configuration file to tell the benchmark how to process the video data.

In the SPEC benchmark suite, h264ref is executed with three different configuration and video data combinations. The first two configurations (h264ref-1 and h264ref-2) use the same video data: a video of 120 frames and a resolution of 176x144 pixels. The h264ref-1 benchmark uses the basic profile configuration, which performs good compression with fast encoding, and decoding. The h264ref-2 benchmark performs a higher quality compression used in applications where no data can be lost. The third configuration (h264ref-3) uses a sequence from a video game of 171 frames with a resolution of 512x320 pixels and with good compression. Using the same configuration, larger frames result in a larger dynamic instruction count. The type of configuration also affects the dynamic instruction count. Higher quality compressions, which result in less data loss during decompression, have a higher dynamic instruction count [3]. Since h264ref's input sets had different D-cache memory sensitivity rankings, these programs will be compared to each other in the rest of this work to determine what makes h264ref-2 and h264ref-3 more sensitive.

## 4.3 hmmer

Short for "Profile Hidden Markov Model", hmmer is a program that statistically models multiple sequence alignments, which are used to search for patterns in DNA sequences in protein sequence analysis. Essentially, this benchmark searches a database for patterns, which has applications in many fields. hmmer can accept a workload and database as inputs, or a workload and parameters to randomly generate a database.

In the SPEC benchmark suite, hmmer is executed with each type of input set. hmmer-1 and hmmer-2 search for different sequence patterns, and obtain the database to search in different ways. hmmer-1 uses a database and sequence pattern to search for. hmmer-2 randomly generates a database, and searches for a less complex sequence pattern than hmmer-1. hmmer-1 also searches for a more complex sequence pattern than hmmer-2. Only the length and number of sequences being searched significantly affect the program behavior [3]. The dynamic instruction count increases as the length and number of sequences increases, which means that hmmer-1 has a higher dynamic instruction count than hmmer-2. As with h264ref, since hmmer's two input sets were ranked differently, their program behavior will be compared to investigate D-cache memory sensitivity.

## 5. DYNAMIC INSTRUCTION COUNT AND INSTRUCTION MIX

Figure 1 shows the dynamic instruction count and instruction mix of the SPEC CPU2006 integer benchmarks. The dynamic instruction count is on the y-axis and the instruction mix breakdown can be observed within each bar. The benchmarks are in descending order based on the number of load instructions. These characteristics were collected on a system using the Intel C/C++ compiler [2].

Libquantum, h264ref, and hmmer have the three highest dynamic instruction counts out of all of the integer benchmarks. Each of the benchmarks have over three trillion dynamic instructions, while the benchmark with the next highest count, bzip2, has only about 2.5 trillion dynamic instructions. This indicates that these benchmarks are either manipulating large amounts of data or executing very complex algorithms. Manipulating large amounts of data requires room in the D-cache for all of the data, and executing complex algorithms requires room in the D-cache for intermediate values to be stored. This assumption is consistent with the behavior of the three benchmarks. Libquantum performs a complex algorithm using quantum gate simulation. The h264ref benchmark accesses, and manipulates very large matrices of data. Lastly, hmmer accesses a large database, and performs complex databases searches.

Libquantum, h264ref, and hmmer also have the highest number of dynamic load instructions of all of the integer benchmarks. All three benchmarks have over 1.2 trillion load instructions, while the other nine integer benchmarks have, on average, about half a trillion loads. This further supports that these benchmarks access and manipulate a large amount of data, which could contributes to their L1 D-cache sensitivity. However, bzip2 has about 900 billion load instructions, only 300 billion less than libquantum, but was classified as having a low memory sensitivity. The high load counts of the three memory sensitive benchmarks can be misleading, because a benchmark with a large number of load instructions does not necessarily have a large memory footprint. The memory footprint of a benchmark is based on how many distinct memory accesses occur. For example, if a load instruction is repeated, the load count will increase, but the memory footprint is unchanged. A large memory footprint will put pressure on the D-cache, causing conflict and capacity cache misses, resulting in poor D-cache behavior. The memory footprints of these benchmarks will be further explored in Section 6.

## 6. LOCALITY CHARACTERISTICS

According to [2], all three of the memory sensitive benchmarks have high locality, meaning that a majority of the dynamic instructions are spent in a few of subroutines. A summary of their findings can be found in Appendix I. The table shows what percentage of the dynamic instruction count is spent in its "hottest", or most frequently called, subroutine, and its 5, 10, and 20 hottest subroutines. Programs with high locality tend to have better performance, because a cache can exploit the programs temporal locality, but there are other microarchitectural characteristics to take into account. For example, if a loop is too large to fit into the I-cache, or includes a larger number of data accesses than the D-cache can handle, then caching will not increase performance. Instruction cache (I-cache) size, D-cache size, memory latencies, replacement algorithms and the presence of prefetching are all other factors that can effect memory performance. Since the details of the five machines used to determine the SPEC benchmarks' memory sensitivity ranking are not available, it will be challenging to conclude the effect that locality has on the L1 D-cache miss rate of these benchmarks.

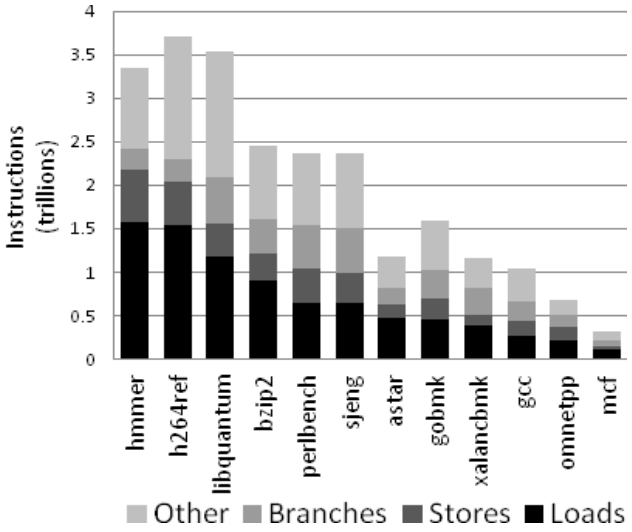


**Figure 1. Dynamic Instruction Count and Instruction Mix of SPEC CPU2006 Integer Benchmarks**

The hmmer benchmark has remarkable locality, with almost all (>95%) of its dynamic instructions belonging to a single subroutine. Libquantum also has good locality, with 98.38% of its dynamic instructions spent in the top five subroutines. The h264ref benchmark has the worst locality of the three benchmarks, requiring 20 subroutines to account for at least 90% of its dynamic instructions. The h264ref-2 benchmark has the worst locality of the three h264ref benchmarks, which is expected, since it simulates a higher quality compression. In comparison to the other SPEC CPU2006 integer benchmarks, all three benchmarks have above average locality [2]. While high locality can decrease I-cache misses [5], memory access patterns are more complex. However, it is likely that higher locality could lead to a smaller memory footprint, since some loads are likely to be repeated, thus decreasing memory sensitivity. Decoding data memory access patterns is a research area of great interest [2][5][6].

Though the static length of these hot subroutines is included in Appendix I, it is not helpful in drawing conclusions about the D-cache behavior. The length of each loop would only affect the I-cache performance, not the D-cache performance. However, since these three benchmarks have a significantly large amount of loads compared to other integer benchmarks, these hot subroutines most likely contain a large amount of D-cache accesses. To make a firm conclusions about this though, more analysis would need to be done to confirm a high number of D-cache accesses in each loop.

## 7. CACHE BEHAVIOR

To investigate the memory footprint of the three memory sensitive benchmarks, simulation were run to analyze the effect of D-cache size on D-cache miss rate. This experiment can give some insight into the memory footprint of the benchmarks. This data was collected using SimpleScalar 3.0, with all simulations were run by fast forwarding 100 billion instructions and then collecting performance metrics for an additional 2 million instructions.

Table 2 summarizes the SimpleScalar configuration used for the simulations. The I-cache was increased to a size sufficient to not hinder the performance of the D-cache. Figures 2, 3 and 4 show the results of these simulations for libquantum, hmmer, and h264ref, respectively. The y-axis is the D-cache miss rate and the x-axis is the D-cache size in kilobytes.

What is interesting to note in these figures is when a steep decrease occurs from one data point to the next. Though the trends can indicate whether a program has cache friendly behavior, a steep drop in cache miss rate can give insight into the number and size of a program's *working sets*. A working set is the memory that a program accesses. Each large drop in cache miss rate implies that one of the working sets can now fit into the cache [7], thus reducing conflict misses. A program with a high locality should see a few, steep drops in D-cache miss rate, as the working set size associated with each function is accommodated.

Libquantum's behavior is very interesting, because it does not show an increase in cache performance as the cache size is increased, but instead, only shows one steep decrease in the D-cache miss rate when the D-cache is 64B (annotated as 1/16 in Figure 2). In other works, libquantum has been found to have a very large working set, and require a 32MB cache to reduce the miss rate to nearly zero [7]. This is why this data does not show its miss rate approaching zero like with h264ref and hmmer. Further simulations with larger D-cache sizes would need to be run for a complete understanding of libquantum's working sets. However, from the data that is available, and considering what is known about libquantum's locality, it is likely that libquantum's most common functions have a high number of D-cache accesses. Libquantum spend more than 98% of its dynamic instructions in 5 functions, and about 65% of its dynamic instructions in one single function. Increasing the cache size incrementally results in a 50% reduction in the cache miss rate, which implies that one of libquantum's working sets can now fit into the cache. It is likely that this working set is from one of libquantum's most frequently references functions. This behavior could have contributed to libquantum's high memory sensitivity ranking, because different sized caches could result in very different performance. Though more data is needed to confirm this, libquantum's memory sensitivity is likely caused by having a few, very large working sets.

**Table 2. SimpleScalar Configuration for Cache Behavior Simulations**

| **L1 cache (**Data/Instruction) | | **branch predictor** | |
|---|---|---|---|
| Size | 4/16KB | 2K-entry bimodal with 512 entry BTB (direct-mapped) | |
| Line size | 32 | | |
| Associativity | 1/4 | | |
| Repl. Policy | LRU | | |
| Latency | 1 cycle | | |
| **L2 cache (**Data/Instruction) | | **micro-architecture** | |
| Size | varied/64KB | fetch/issue/decode | 4 |
| Line size | 32 | functional units | 4 |
| Associativity | 1/8 | | |
| Repl. Policy | LRU | **memory** | |
| Latency | 6 cycles | Latency | 18 cycles |

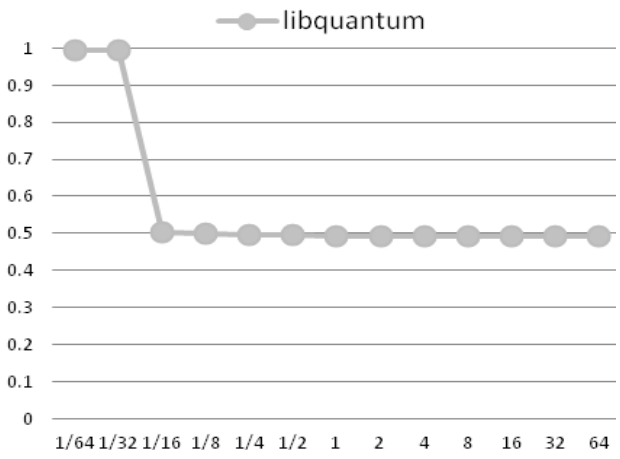


**Figure 2. D-Cache Miss Rate for Varying D-Cache Size for Libquantum Benchmark**

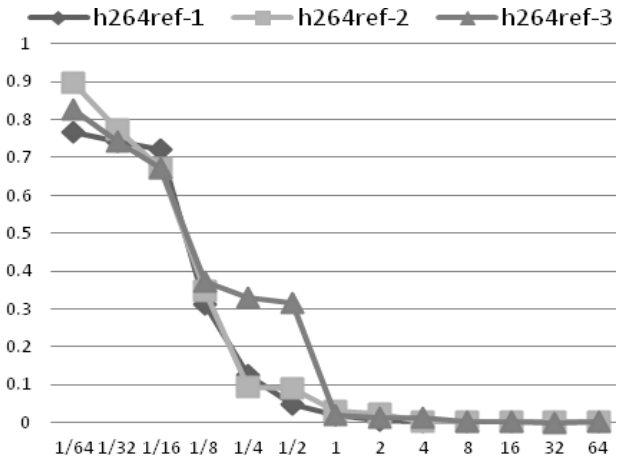


**Figure 3. D-Cache Miss Rate for Varying D-Cache Size for h264ref Benchmark**

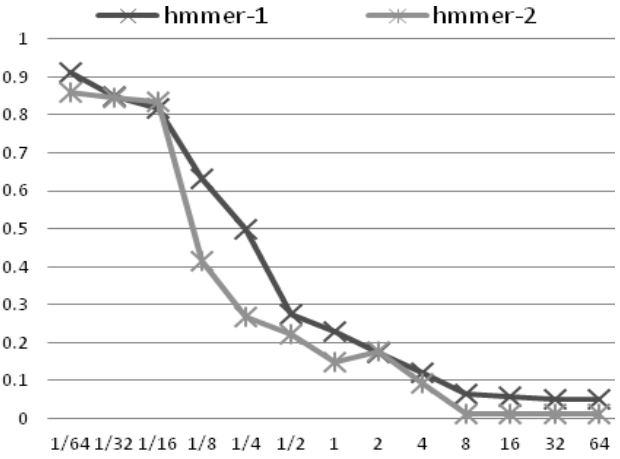


**Figure 4. D-Cache Miss Rate for Varying D-Cache Size for Hmmer Benchmark**

Since the h264ref and hmmer benchmarks have multiple input sets with different memory sensitivities, comparing their behavior across input sets can provide insight into what makes a benchmark memory sensitive. For h264ref, h264ref-1 had a low sensitivity, while h264ref-2 and h264ref-3 have a high sensitivity. In Figure 3, h264ref-3 has slightly different behavior, while h264ref-1 and h264ref-2 are very similar, though h264ref-2 has a slightly larger working set size. All three input sets share a steep drop at 128B (1/8), but h264ref-3 requires a larger D-cache to achieve a miss rate of less than 0.1. This is consistent with what we know of h264ref-3's input set characteristics. The h264ref-3 benchmark's input is video data that has a higher resolution and more frames than h264ref-1 and h264ref-2, which means that its working set is larger. This could explain h264ref-3's high memory sensitivity, but what about h264ref-2? The different memory behavior could be attributed to the slight differences in locality and working set size. The h264ref-2 benchmark has the worst locality of the three, and a larger working set size than h264ref-2 would which make its memory footprint larger than h264ref-1 and closer to the memory behavior of h264ref-3. Therefore, h264ref-2's memory sensitivity can be attributed to its poor locality, therefore increasing its overall memory footprint. Since h264ref-2 simulates a higher quality compression than h264ref's other two input sets, this result makes sense.

The hmmer benchmark has two inputs: hmmer-1 with a high memory sensitivity and hmmer-2 with a medium memory sensitivity. In Figure 4, hmmer-1 has fairly friendly cache memory behavior, with some steep drops, and hmmer-2 exhibits similar behavior, but with a significant steep drop in miss rate at 128B (1/8). The hmmer benchmark has incredible locality and spends most of its time in one function. That would imply that it has a smaller memory footprint than the benchmarks with lower localities. This assumes though that each iteration of the function accesses the same data each time, which may not be true. Take for example, a function that accesses rows for a matrix. While each loop may look the same, the data that it accessing will be different every iteration. This may be the case with hmmer. The hmmer benchmark looks up patterns in a database, so each iteration of its main function is probably accessing a different entries in the database. Unlike with h264ref-2, whose poor locality was a contributing factor of its memory sensitivity, the opposite is likely true for hmmer. If every iteration of its loop is different, its memory footprint would be quite large. As for the difference between hmmer-1 and hmmer-2, hmmer-1's input data is, which increases its dynamic instruction count. If it's true that each iteration of its main loop contributes to its memory sensitivity, then hmmer-1 would have a larger memory footprint than hmmer-2, and thus, be more sensitive to memory changes.

# 8. SUMMARY

Libquantum, h264ref-2, h264ref-3 and hmmer-1 were deemed to have a high L1 D-cache memory sensitivity [2]. For h264ref and hmmer, their other inputs, h264ref-1 and hmmer-2, had a low and medium sensitivity, respectively. Having multiple input sets with different memory sensitivities made it easier to draw conclusions about their behavior, since the input sets could be compared. Libquantum, h263ref and hmmer have the highest dynamic instruction count, as well as the highest total number of load instructions of all of the SPEC CPU2006 integer benchmarks. They had varying localities and memory footprints.

Libquantum's high memory sensitivity ranking can be attributed to having a few, very large working sets. This would explain why a large variance in L1 D-cache behavior was observed across the five machines, resulting in libquanutm receiving a high memory sensitivity ranking. Even if the memory systems in these machines were very similar in size, as shown in Figure 2, a small increase in cache size can result in a dramatic decrease in cache miss rate. From what is known of libquantum's locality, it is likely that a majority of D-cache accesses were made in its most frequently re-referenced functions.

Like libquantum, the h264ref benchmark's h264ref-2 and h264ref-3 also have a high memory sensitivity, because of their large memory footprint. The h264ref-2 benchmark has a large memory footprint, because it simulates a very high quality compression, which reduces the amount of data lost during compression. This not only increases the memory footprint, but decreases the locality, which also contributes even more to the memory footprint. The h264ref-3 benchmarks has a high memory footprint simply because it modifies the largest amount of data; it video data input has more frames and a higher resolution than the other two h264ref benchmarks.

The hmmer benchmark can also attribute its high memory sensitivity ranking to a large memory footprint. The hmmer programs had the best locality, with 99% of hmmer-1's dynamic instruction being spent in one function. For hmmer-2, it spent 96% of its dynamic instructions in one function. Based on what is known of hmmer's behavior of accessing a large database repeatedly, it was concluded that its largest function must access different data during every iteration, thus contributing to its overall memory footprint. Since hmmer-1 has a more complex input than hmmer-2, it has a higher memory sensitivity.

# 9. CONCLUSION AND FUTURE WORK

In this work, the L1 D-cache sensitivities of three SPEC CPU2006 benchmarks, libquantum, h264ref, and hmmer, were evaluated, as they are possible Big Data processor benchmarks. The memory footprint of each benchmark were deemed to be the source of their high sensitivity to memory changes, but for different reasons. Libquantum has a few, large working sets, while h264ref and hmmer simply access a lot of unique data. While h264ref's poor locality increased its memory footprint, hmmer's exceptionally high locality contributed to it.

Since all three benchmarks have large memory footprints, they show potential as Big Data benchmarks. However, libquantum shows the most promise. Typical Big Data benchmarks have memory footprints greater than 10GB and working sets around 1GB [8]. Though the memory footprint was not explicitly determined in this work, libquantum's working set size is the only benchmark of the three to be larger than 64KB. In future work, the memory footprint of these benchmarks will be determined. The strong locality of libquantum, h264ref, and hmmer also makes them good candidates for Big Data benchmarking.

Though most optimization is currently done at the system-level, cores are now being designed specifically for Cloud Computing and Big Data applications [9][11], so node-level optimization is in the near future. In future work, the role of these memory sensitive benchmarks in designing processors for Big Data applications will be further explored. More work needs to be done to understand their memory access patterns, parallelism, and memory footprint.

## 10. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under award number CNS-1063106.

## 11. REFERENCES

[1] "SPEC CPU2006." Internet: http://www.spec.org/cpu2006/, Sept.7, 2011 [April 8, 2014].

[2] A. Phansalkar et al., "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," in ISCA, 2007.

[3] V. Escuder and R. Rico, "Reduced input data sets selection for SPEC CPUnt2006," Universidad de Alcala, Spain, Rep. TR-HPC-02-2009, April 2009.

[4] K. Hoste. *SPEC CPU2006 command lines* [Online]. Available: http://boegel.kejo.be/ELIS/spec_cpu2006/spec_cpu2006_command_lines.html

[5] G. Thompson, B. Nelson and J. Flanangan, "Transaction Processing Workloads - A Comparison to the SPEC Benchmarks Using Memory Performance Studies, " in MASCOTS, 1996.

[6] Y. Chen and Y. Liu, "Dual-addressing memory architecture for two-dimensional memory access patterns, " in DATE, 2013.

[7] A. Jaleel. "Memory Characterization of Workloads Using Instrumentation-Driven Simulation, " to be published in 2015. [Online]. Available: http://http://www.glue.umd.edu/~ajaleel/workload/

[8] M. Dimitrov et al., "Memory System Characterization of Big Data Workloads, "in Big Data, 2013

[9] *OVH Launches Cloud Service Based on IBM POWER8 Processor* [Online]. Available: http://www.hpcwire.com/off-the-wire/ovh-launches-cloud-service-based-ibm-power8-processor/

[10] F. Liang et al., "Performance Characterization of Hadoop and Data MPI Based on Amdahl's Second Law, " in ICNAS, 2014.

[11] W. Buros et al. "Understanding Systems and Architecture for Big Data, " IBM Research Report, 2013.

[12] W. Xiong, "A Characterization of Big Data Benchmarks, " in ICBD, 2013.

**Appendix 1. Subroutine Profile Summary of Memory Sensitive SPEC CPU2006**

| Cumulative | Hottest Subroutine | | 5 Hot Subroutines | | 10 Hot Subroutines | | 20 Hot Subroutines | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Percentage Dynamic** | **Static Count** | **Percentage Dynamic** | **Static Count** | **Percentage Dynamic** | **Static Count** | **Percentage Dynamic** | **Static Count** |
| hmmer-1 | 99.10% | 11080 | 99.76% | 143630 | 99.91% | 385550 | 99.96% | 993103 |
| hmmer-2 | 96.79% | 11080 | 99.76% | 220363 | 99.99% | 458965 | 100.00% | 711948 |
| libquantum | 65.18% | 901 | 98.38% | 12897 | 100.00% | 39182 | 100.00% | 89909 |
| h264ref-1 | 41.21% | 63541 | 75.28% | 360800 | 90.46% | 733452 | 96.06% | 1281878 |
| h264ref-2 | 35.20% | 63541 | 65.66% | 263448 | 81.81% | 632195 | 91.08% | 1103854 |
| h264ref-3 | 36.20% | 63541 | 71.16% | 263448 | 83.30% | 638070 | 92.28% | 1162685 |