BlackjackBench: Portable Hardware Characterization

Anthony Danalis[*]
University of Tennessee
Knoxville, TN, USA
adanalis@eecs.utk.edu

Piotr Luszczek
University of Tennessee
Knoxville, TN, USA
luszczek@eecs.utk.edu

Gabriel Marin
Oak Ridge National Lab.
Oak Ridge, TN, USA
maring@ornl.gov

Jeffrey S. Vetter[†]
Oak Ridge National Lab.
Oak Ridge, TN, USA
vetter@ornl.gov

Jack Dongarra[‡]
University of Tennessee
Knoxville, TN, USA
dongarra@eecs.utk.edu

ABSTRACT

DARPA's *AACE* project aimed to develop Architecture Aware Compiler Environments that automatically characterizes the hardware and optimizes the application codes accordingly. We present the *BlackjackBench* – a suite of portable benchmarks that automate system characterization, plus statistical analysis techniques for interpreting the results. The BlackjackBench discovers the effective sizes and speeds of the hardware environment rather than the often unattainable peak values. We aim at hardware characteristics that can be observed by running standard C codes. We characterize the memory hierarchy, including cache sharing and NUMA characteristics of the system, properties of the processing cores affecting instruction execution speed, and the length of the OS scheduler time slot. We show how they all could potentially interfere with each other and how established classification and statistical analysis techniques reduce experimental noise and aid automatic interpretation of results.

1. INTRODUCTION

Automatic detection of hardware characteristics is necessary given the dramatic changes undergone by computer hardware. Several system benchmarks exist in the literature [14, 13, 3, 9, 4, 6, 10, 8]. As hardware becomes more complex, new features need to be characterized and assumptions revised or completely redesigned.

In this paper, we present *BlackjackBench*, a system characterization benchmark suite. The contribution of this work is twofold:

• Portable micro-benchmarks that can probe the hardware and record its behavior while control variables are varied.

• A statistical analysis methodology, implemented as a collection of scripts for result parsing, that examines the output of the micro-benchmarks and produces the desired system characterization information, e.g. effective speeds and sizes.

Important performance related decisions take into account *effective* values of hardware features, rather than their peak values. An effective value of a hardware feature affects the user level application written in C running on that hardware. This is in contrast with values that can be found in vendor documents, or through assembler benchmarks, or specialized instructions and system-calls.

BlackjackBench goes beyond the state of the art in system benchmarking by characterizing features of modern multicore systems, taking into account contemporary hardware characteristics such as modern cache prefetchers, and the interaction between the cache and TLB hierarchies, etc. BlackjackBench combines established classification and statistical analysis techniques with heuristics tailored to specific benchmarks, to reduce experimental noise and aid automatic interpretation of the results.

[*]Also with Oak Ridge National Laboratory (ORNL).

[†]Also with Georgia Institute of Technology

[‡]Also with ORNL and the University of Manchester

2. RELATED WORK

Some benchmarks, such as those described by Molka et. al [7], aim to analyze the micro-architecture of a specific platform in great detail and thus sacrifice portability and generality. Others [2] sacrifice portability and generality by depending upon specialized software such as PAPI [1]. Autotuning libraries such as ATLAS [12] rely on micro-benchmarking for accurate system characterization for a very specific set of routines which need tuning. These libraries also develop their own characterization techniques [11], that need to be subsumed to target a much broader feature spectrum.

CacheBench [8], or `lmbench` [6, 10] are higher level, portable, and use similar techniques to those we use – such as *pointer chasing* – but output large data sets or graphs that need human interpretation instead of values that characterize the hardware platform.

X-Ray [14, 13] is a suite that discovers only some of the features that we chose. There are also differences in methodology which we mention throughout. One important distinguishing feature is X-Ray's emphasis on code generation – we give more emphasis on analyzing the resulting data.

P-Ray [3] is a micro-benchmark suite that complements X-Ray by characterizing multicore hardware features such as cache sharing and processor interconnect topology. We extend the P-Ray contribution with new tests and measurement techniques as well as a larger set of tested hardware architectures.

Servet [4] attempts to subsume X-Ray and P-Ray. It adds measurements of interconnect parameters for communication that occurs in a distributed memory. The methodology and measurement techniques in Servet complement those of X-Ray and P-Ray and remain in sharp contrast with our work. We do not focus on the actual hardware parameters. Rather, we seek the observable parameters that often are below vendors' specifications. Servet's portability extends only Intel Xeon and Itanium clusters.

3. BENCHMARKS

We assert that, by observing variations in the performance of benchmarks, all hardware characteristics that can significantly affect the performance of applications can be discovered. Conversely,

if a hardware characteristic cannot be discovered through performance measurements, it is probably not very important.

The memory hierarchy in modern architectures is complex, with hardware prefetchers, victim caches, etc. Unlike benchmarks [4, 9] that use constant strides when accessing their buffers, we use a technique known as pointer chasing (or pointer chaining) [3, 6, 14]. To achieve this, we use a buffer that holds pointers (`uintptr_t`) instead of integers, or characters. We initialize the buffer so that each element points to the element that should be accessed next, and then we traverse the buffer in a loop that reads an element and dereferences the value read to access the next element: `ptr=(uintptr_t *)*ptr`. The benefits of pointer chasing are threefold. A) The initialization of the buffer is not part of the timed code. Therefore, it does not cause noise in the performance measurement loop even when sophisticated pseudo-random number generators are used; B) it eliminates the possibility that a compiler could alter the loop that traverses the buffer, since the addresses used in the loop depend on program data (the pointers stored in the buffer itself); C) it eliminates the possibility that the even sophisticated hardware prefetcher(s) can guess the address of the next element.

Cache Hierarchy Improved cache utilization is one of the most performance critical optimizations in modern hardware. As a result of the increasing gap between processor and memory speed, most modern processor designs incorporate complex, multi-level cache hierarchies that include both shared and non-shared caches between the processing elements.

Cache Line Size We assume that *upon a cache miss the cache fetches a whole line* (A1). Two consecutive accesses to memory addresses that are mapped to the same cache line will result in, at most, one cache miss. In contrast, two accesses to memory addresses that are farther apart than a cache line may result in two cache misses.

We allocate a buffer large enough to exceed the cache, and perform memory accesses in pairs. The buffer is aligned to 512 bytes, which is larger than the cache line size. Each access is to an element of size equal to the size of a pointer (`uintptr_t`). Each pair of accesses touches the first and the last elements of a memory segment with extent D of a random buffer location. The pairs are chosen such that every odd access, starting with the first one, is at a random memory address within the buffer and every even access is $D - sizeof(uintptr\_t)$ bytes away from the previous one. The random nature of the access pattern and the large size of the buffer guarantees, statistically, that the vast majority of the odd accesses will result in cache misses. However, the even accesses can result in either cache hits or misses depending on the extent D. If D is smaller than the cache line, each even access will be in the same line. If D is larger than the cache line size, the two addresses will map to different cache lines.

An access that results in a cache hit is served at the latency of the cache, while a cache miss is served at the latency of lower-level caches, which leads to significantly higher latency. Our benchmark forces this behavior by varying the value of D, expecting a significant increase in average access latency when D becomes larger than the cache line size. A sample run of this benchmark can be seen in Figure 1(a).

Cache Size and Latency Using the cache line size, we detect the number of cache levels, their sizes, and access latencies. The enabling assumption is that *performing multiple accesses to a buffer that fully resides in the cache is faster than performing the same accesses to a buffer that does not fully reside in the cache* (A2).

The benchmark allocates a buffer that is expected to fit in the smallest cache of the system. The buffer has only a few cache lines. Then we access the whole buffer with a stride equal to the cache line size. We estimate the average latency per access by making the access pattern random to avoid prefetching effects. And by continuously accessing the buffer until every element has been accessed multiple times we amortize start-up and cold misses overhead.

The benchmark varies the buffer size and performs the same random access traversal for each new buffer size, recording the average access latency at each step. Due to assumption A2, we expect that the average access latency will be constant for all buffers that fit in a particular cache of the hierarchy, but there will be a significant access latency difference for larger buffers. By varying the buffer size, we expect to generate a set of Access Latency vs. Buffer Size data that looks like a step function with multiple steps. A step in the data set should occur when the buffer size exceeds the size of a cache. The number of steps will be the number of caches plus one for the main memory. The Y value at the apex of each step will be the access latency of each cache.

In order to eliminate false steps, the cache benchmarks minimize the effects of the TLB by accessesing the buffer in a not uniformly random manner. Instead, the buffer is split into segments equal to a page size. The benchmark accesses the elements of each segment in random order, but exhausts all the addresses in a segment before proceeding to the next segment. This approach guarantees that, in a system with a cache line size S_L and a page size S_P, there are at least S_P/S_L cache misses for each TLB miss, which in a typical modern architecture is around 64. Figure 1(b) show how this approach enables us to infer the characteristics of the cache hierarchy from data sets gathered from real hardware.

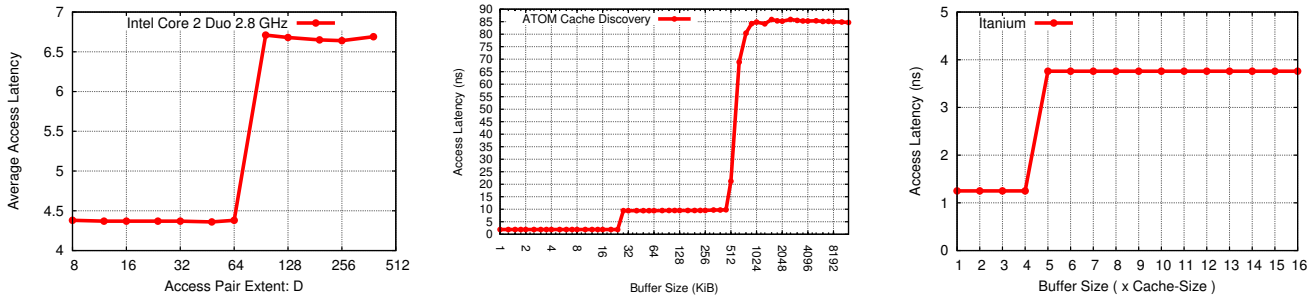
Cache Associativity We assume is that *in a system with a cache of size S_c, two memory addresses A_1 and A_2, such that $A_1 \% S_c = A_2 \% S_c$, map to the same set of an N-way associative cache* (A3).

We allocate a buffer many times larger than the cache size, $M \times S_c$. Next, the benchmark starts accessing a part of the buffer that is $K \times S_c$ large (with $K \leq M$), repeating the experiment for different, integral values of K. For every value of K the access pattern consists of randomly accessing addresses that are S_c bytes apart; this process is repeated a sufficient number of times. Since all such addresses will map into the same cache set, as soon as K becomes larger than N, the cache will start evicting elements to store the new ones, and therefore some of the accesses will result in cache misses. Consequently, the average access time should be significantly lower for $K \leq N$ than for $K > N$. The output from a sample run of this benchmark on an Itanium processor is shown in Figure 1(c).

We note that a victim cache may cause the benchmark to detect an associativity value higher than the real one. Thus, our benchmark accesses elements in multiple sets, instead of just one.

Asymmetries in the Memory Hierarchy Shared caches and integrated memory controllers have become the norm. Both of these hardware features can potentially create asymmetries in the memory system. The cores may communicate faster with other cores from the same subset than with cores from another subset. For NUMA, data allocated on one NUMA node is accessed faster by the local cores than by cores from a different node.

We use a multi-threaded benchmark whereby two threads access a shared memory block. One thread allocates and then initializes the memory block. The benchmarks assume that a NUMA system implements the first-touch page allocation policy. Next, the two threads take turns incrementing all the locations in the memory block. One thread reads, and then modifies the entire memory block before the other thread. The two threads synchronize using busy waiting on a volatile variable. We use padding to ensure that the locking variable is allocated in a separate cache line, to avoid false sharing. This approach causes the two threads to act both as producers and consumers, switching roles after each complete up-



(a) Line Size Characterization on Core 2 Duo (b) Cache Characterization on Atom (c) Cache Associativity on Itanium II

Figure 1: Cache characterization on different platforms.

date of the block. We repeat this process many times to minimize noise and we compute the achieved bandwidth for different block sizes. We use a range of block sizes, from a size smaller than the L1 cache to a size larger than the last level of cache. We test all pairs of cores. To be OS independent, we must assume that a thread is executed on the same core. We control the placement of the threads using an OS specific API for pinning threads to cores. The communication profiles are analyzed in decreasing order of the memory block size, to detect any potential cliques of cores that are close to each other. The algorithm starts with a large clique that includes all the cores. At each step, the data for the next smaller memory block is processed. The communication data between all cores of a clique identified at a previous step is analyzed to identify any sub-cliques of cores that communicate faster for a smaller block.

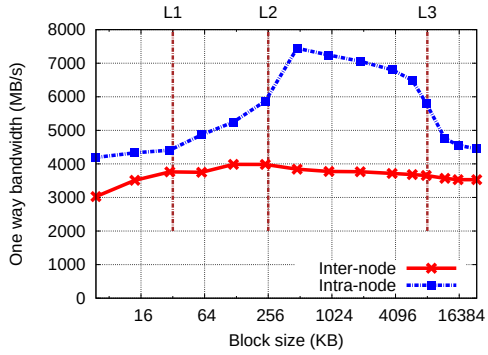


Figure 2: One-way, inter-core communication bandwidth on a dual-socket Intel Gainestown with Hyper-Threading disabled.

In Figure 2, the *x* axis represents the block size, and the *y* axis represents the single thread bandwidth. The bandwidth is computed as *number_updated_lines* × *cache_line_size* / *time*, where the number of cache lines updated by the first thread is *number_updated_lines*. Since the two threads update an equal number of lines, the values shown in the figure represent only half of the actual two-way bandwidth. As expected for such a system, the benchmark captures two distinct communication patterns: (1) cores reside on different NUMA nodes, curve labeled *inter-node*; and (2) cores are on the same NUMA node, curve labeled *intra-node*.

When the data for the largest memory block size is analyzed, the algorithm divides the initial set of eight cores into two cliques of size four, corresponding to the two NUMA nodes in the system. The cores in each of these two cliques communicate among themselves at the speed shown by the *intra-node* point, while the communication speed between cores in distinct cliques corresponds to the *inter-node* point. The algorithm continues by processing the data for the smaller memory block sizes, but no additional asymmetries are uncovered in the following steps. The locality tree for this system has just two levels, with the root node corresponding to the entire system at the top, and two nodes of four cores at the next level corresponding to the two NUMA nodes in the system.

3.1 TLB Hierarchy

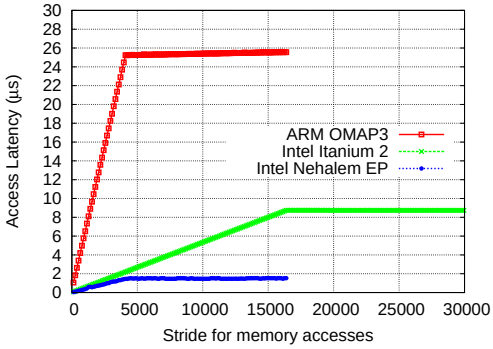


Figure 3: Timing results for discovering the TLB page size.

TLB hierarchy is an important part of the memory system that bears some resemblance to the cache hierarchy. However, TLB is sufficiently different to warrant its own characterization method.

Crucially, we ensure that our TLB benchmarks alleviate cache effects on the measurements. We chose for the effects of data cache misses to be filtered out during the analysis of the results. An added benefit is capturing the rare events when the TLB and the data cache are inherently interconnected, such as when TLB fits the same number of pages as there are data cache lines.

To determine the page size, our benchmark maximizes the penalty coming from the TLB misses. We do it by traversing a large array multiple times with a given stride. The array is large enough to exceed the span of any TLB level – this guarantees a high miss rate if the stride is larger or equal to the page size. If the stride is less than the page size, some of the accesses to the array will be contained in the same page, and thus, will decrease the number of misses and the overall benchmark execution time. The false positives stemming from interference of data cache misses are naturally eliminated by the high cost of a TLB miss in the last level of TLB. Typical timing curves for this benchmark are shown in Figure 3. The figure shows results from three very different processors: ARM OMAP3, Intel Itanium 2, and Intel Nehalem EP. The graph line for each system

has the same shape; for strides smaller than the page size, the line raises as the number of misses increases because fewer memory accesses hit the same page. And for strides that exceed the page size, the graph line is flat because each array access touches a different page so the per-access overhead remains the same. The page size is determined as the inflection point in the graph line: 4KB for both ARM and Nehalem and 16KB for Itanium.

Once the actual TLB page is known, it is possible to proceed with Discovering the sizes of the levels of the TLB hierarchy involves minimizing the impact of the data caches. The common and portable technique is to perform repeated accesses to a large memory buffer at strides equal to the TLB page size. This technique is prone to creating as many false positives as there are data cache levels and a slight modification to this technique is required [9]. On each visited TLB page, our benchmark chooses a different cache line to access, thus, maximizing the use of any level of data cache. As a side note, choosing a random cache line within a page utilizes only half of the data cache on average. Figure 4 shows the timing graphs where both Level 1 and Level 2 TLBs are identified accurately. We aim at the observable parameters and even if the TLB size is 512 entries, some of these entries will not contain user data. Specifically, there is a need for the function stack and the code segment which will use one TLB page each, thus reducing the observable TLB size by two.

3.2 Arithmetic Operations

Instruction latencies The latency $\mathscr{L}(O,T)$ of an operation O, with operands of type T, is calculated as the number of cycles it takes from the time one such operation is issued until its result becomes available to subsequent dependent operations. BlackjackBench reports all operation latencies relative to the latency of a 32-bit integer addition, since the CPU frequency is not directly detected. For each combination of operation type O and operand type T, our code generator outputs a micro-benchmark that executes in a loop a large number of chained operations of the given type. The loop is executed twice, using the same number of iterations but with different unroll factors. The difference of the two execution times is divided by the difference in operation counts between the two loops. This approach eliminates the effect of the loop overhead, while keeping the unroll factors small. To account for run-time variability, each benchmark is executed six times, and the second lowest result is selected.

Instruction throughputs The throughput $\mathscr{T}(O,T)$ of an operation O, with operands of type T, represents the rate at which one thread can issue and retire independent operations of a given type. Throughput is reported as the number of operations that can be issued in the time it takes to execute a 32-bit integer addition.

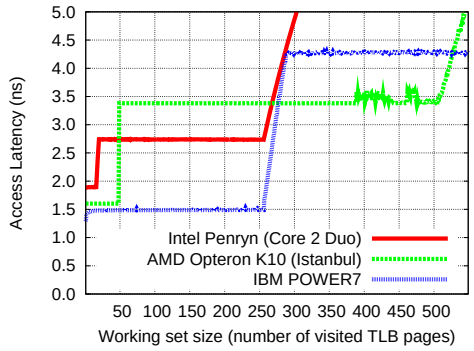


Figure 4: TLB characterization on three platforms.

To determine the maximum rate at which operations are issued, micro-benchmarks must include many independent operations as opposed to chained ones. At the same time, using too many parallel operations increases register pressure, causing unnecessary delays due to register spills/unspills. For each operation type O and operand type T, multiple micro-benchmarks are generated each with a different number of independent streams. The number of parallel operations is varied between 1 and 20, and the minimum time per operation across all versions, L_{min}, is recorded. Throughput is computed as the ratio between the latency of a 32-bit integer addition and L_{min}.

Operations in flight The number of operations in flight $\mathscr{F}(O,T)$ for an operation O, with operands of type T, measures how many such operations can be outstanding at any given time in a single thread. This measure is a function of both the issue rate and the pipeline depth of the target processor, and is a unitless quantity.

For each operation type O and operand type T, multiple micro-benchmarks, with different numbers of independent streams of operations, are generated and executed. To find the number of operations in flight, we are looking at the cost per loop iteration. Each independent stream has the same number of chained operations, and the total number of operations in one loop iteration grows linearly with the number of streams. When we increase the number of independent streams in the loop, the cost per iteration should remain constant as long as the processor can pipeline all the independent operations. Thus, the number of operations in flight supported by one thread of control is given by the inflection point where the cost per iteration starts to increase.

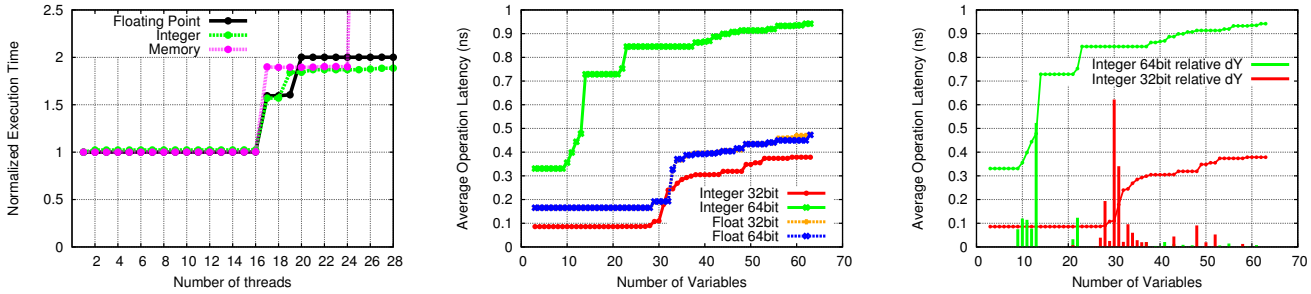
3.3 Execution Contexts

We use the term "Execution Context" to refer to the minimum hardware necessary to effect the execution of a compute thread. Several modern architectures implementing virtual hardware threads exhibit selective preference over different resources. For example, a processor could have private integer units for each virtual hardware thread, but only a shared floating point unit for all hardware threads residing on a physical core. The Blackjack benchmarks attempt to discover the maximum number of: a) floating point, b) integer and c) memory intensive execution contexts.

Intuitively, we are interested in the maximum number of compute threads that could perform floating point computations, integer computations, or intense memory accesses, without having to compete with one another, or wait for the completion of one another. Thus the assumption is that *in a system with N execution contexts, there should be no delay in the execution of each thread for $M \leq N$ threads, but for $M > N$ at least one should be delayed* (A4).

To utilize this assumption, our benchmark instantiates M threads that do identical work (floating point, integer, or memory intensive, depending on what we are measuring) and records the total execution time, normalized to the time of the case where $M = 1$. The experiment is repeated for different values of M until the normalized time exceeds some small predetermined threshold, typically 2 or 3. Due to assumption A4, the normalized total execution time will be constant and equal to 1 (with some possible noise) for $M \leq N$ and greater than one for $M > N$.

To avoid erroneous results due to memory limitations, our benchmark limits memory accesses of each thread to a small memory block that fits in the level 1 cache and uses pointer chasing to traverse it. For the case of the integer and floating point contexts, the corresponding benchmarks execute a tight compute loop with divisions, which are among the most demanding arithmetic operations. An output of this benchmark on Power7 is shown in Figure 5(a).



(a) Execution Contexts on Power7 (b) Live Ranges Characterization on Power7 (c) Live Ranges and Relative dY

Figure 5: Execution contexts and live ranges characterization.

3.4 Live Ranges

We use the term "Live Ranges" to describe the maximum number of concurrently usable registers by a user code which could be different from the actual hardware registers, for example, either the stack pointer or the frame pointer may need to occupy a register and prevent from being used for application variable.

The compiler has to introduce a piece of code that "spills" the extraneous variables to memory when out of registers. This incurs a delay because: 1) memory (even the L1 cache) is slower than the register file and 2) the extra instructions needed to transfer the variable and calculate the appropriate memory location consume CPU resources. Thus, the enabling assumption of this benchmark is that *in a machine with N Live Ranges, a program using $K > N$ registers will experience higher latency per operation than a program using $K \leq N$ registers* (A5). To measure this effect, our benchmark runs a large number of small tests (130), each executing a loop with K live variables (involved in K additions) with K ranging from 3 to 132. By executing all these tests and measuring the average time per operation, we detect the maximum number of usable registers by detecting the step in the resulting data.

The structure of the loop is similar to the one used by X-Ray [14], but is modified in two ways.

• The `switch` statement inside the loop is unnecessary and detrimental. It is unnecessary as the chained dependences of the operations are sufficient to prohibit the compiler from aggressive optimizions. On some architectures, compilers choose to implement the switch with a jump determined by a register value – one less register could be reported by the benchmark.

• The simple dependence chain suggested in X-Ray: $P_{i\%N} = P_{i\%N} + P_{(i+N-1)\%N}$ can lead to inconclusive results in architectures with deep pipelines and sophisticated ALUs with spare resources. To reduce the impact of this effect we used the pattern: $P_{i\%N} = P_{i\%N} + P_{(i+N-\frac{N}{2})\%N}$ for the loop body. This pattern enables $\lfloor N/2 \rfloor$ operations to be pipelined provided sufficient pipeline depth. As a result, the execution of the operations in the loop puts much more pressure on the CPU and makes it harder to hide the memory overheads.

Hand optimizations of the code, such as loop unrolling, were used to amortize the loop overhead and reduce the importance of the induction variable for register allocation. An example run of this benchmark, on a Power7 processor, can be seen in Figure 5(b).

The actual output data of this benchmark shows that for very small number of variables the average operation latency is higher than the minimum operation latency achieved. However, since a register spill can only increase latency, the curves shown in the Figure and used to extract the Live Ranges information, were obtained after applying *monotonicity enforcement* to the data. This technique is discussed in Section 4.

3.5 OS Scheduler time slot

The time duration during which a thread will run on the CPU uninterrupted is called a scheduler time slot. We rely on the assumption that *when a program is executed on a CPU with frequency F, the latency between instructions should be on the order of $1/F$, where two instructions that execute in different time slots are separated by a time interval on the order of the OS Scheduler time slot, T_s, which will be several orders of magnitude larger than $1/F$* (A6).

Our benchmark consists of several threads. Each thread executes a loop that performs a small number of arithmetic operations, takes a timestamp, and records the elapsed time since the previous timestamp, T_i, if it was larger than a predefined threshold. The threshold is chosen such that it is much longer than the time the few operations and the timestamp function take to execute, but shorter than the expected value of the scheduler time slot, T_s. A handful of arithmetic operations and a call to a timestamp function, commonly take less than a microsecond. The scheduler time slot is typically in the order of, or larger than, a millisecond. A threshold of $20\mu sec$ safely satisfies our criteria. Since we do not record the duration, T_i, of any loop iteration with $T_i < threshold$, the only values recorded will be from iterations whose execution was interrupted by the OS and spanned more than one scheduler time slot.

The benchmark assumes knowledge of the number of execution contexts on the machine and oversubscribes them by a factor of two by generating twice as many threads as there are execution contexts. Since all threads are compute-bound, perform identical computation, and share an execution context with some other thread, we expect that, statistically, each thread will run for one time slot and wait during another. Regardless of scheduling fairness decisions and esoteric OS details, the mode of the output data distribution (i.e., the most common measurement taken) is the duration of the scheduler time slot, T_s which was confirmed on several hardware platforms and OSes.

4. STATISTICAL ANALYSIS

Monotonicity enforcement In all our benchmarks, except for the micro-benchmark that detects asymmetries in the memory hierarchy, the output curve is expected to be monotonically increasing. If any data points violate this expectation, it is due to random noise, or esoteric hardware details that are beyond the scope of this benchmark suite. Therefore, as a first post-processing step we enforce monotonic increase using the formula: $\forall i : X_i = \min_{j \geq i} X_j$

Gradient Analysis Most of our benchmarks result in data that resemble step functions. Therefore the challenge is to detect the

location of the step, or steps, that contain the useful information.

First Step The Execution Contexts benchmark produces curves that start flat (when the number of threads is less than the available resources), then exhibit a large jump (when the demand exceeds the resources), and then continue with noisy behavior and potentially additional steps as can be seen in Figure 5(a).

To extract the number of Execution Contexts, we are we locate the first jump from the straight line to the noisy part. To eliminate influence of small jumps in the flat part, we systematically define the difference between a small and large jumps. We first calculate the relative value increase in every step $dY_n^r = (Y_{n+1} - Y_n)/Y_n$ and then compute the average relative increase $\langle dY^r \rangle$. The data point that corresponds to the jump we seek is the first data point i for which $dY_i^r > \langle dY^r \rangle$. The rationale is that the average of a large number of very small values and a few much larger values will be a value higher than the noise, but smaller than the steps. Thus $\langle dY^r \rangle$ gives us a good threshold between small and large values.

Biggest Step The Live Ranges benchmark produces curves that start flat then potentially grow slightly, then exhibit a large step when the first spill to memory occurs, and then continue growing in a non-regular way, see Figure 5(b). Due to the increase in latency before the first spill, the previous approach for detecting the first step is not appropriate here. However, the steps caused by the additional spills will be no larger than the step caused by the first spill. Furthermore, since the additional steps have higher starting values than the first step, the relative increase $\frac{(Y_{n+1}-Y_n)}{Y_n}$ for every n higher than the first spill will be lower than the relative increase of the first spill. To demonstrate this point, Figure 5(c) shows the data curves for the integer live ranges along with the relative dY^r values.

The biggest relative step technique can also be used for processing the results of the cache line size benchmark and the cache associativity benchmark. For the TLB page size, where the desired information is in the last large step, the analysis seeks the biggest scaled step $dY^s = dY \times Y$.

Quality Threshold Clustering The benchmark for detecting the cache size, count, and latency has multiple steps that carry useful information. We can group the data points into clusters based on their Y value (access latency) such that each cluster includes the data points that belong to one cache level. For the clustering, we use a modified version of the quality threshold clustering algorithm [5]. We modify the cluster diameter threshold used by the algorithm to determine if a candidate point belongs to a cluster or not. Unlike regular QT-Clustering, where the diameter is a constant value predetermined by the user of the algorithm, our version uses a variable diameter equal to 25% of the average value of each cluster. An example use of this analysis, on the data from a Power7 processor, can be seen in Figure 6.

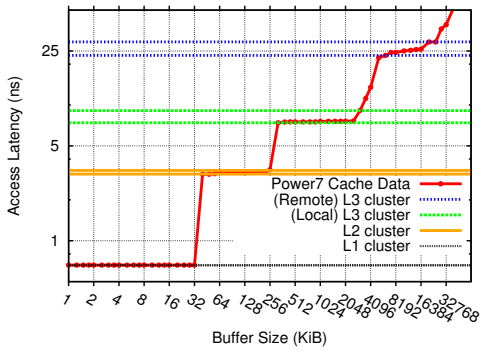


Figure 6: QT-Clustering applied to Power7 Cache Data

5. CONCLUSION

We have presented the *BlackjackBench* suite of micro-benchmarks goes beyond the state of the art in benchmarking by:

- Offering micro-benchmarks that can exercise a wider set of hardware features than most existing benchmark suites do.
- Emphasizing portability by avoiding low level primitives, specialized software tools and libraries, or non-portable OS calls.
- Providing statistical analyses capable of inferring useful values that describe the hardware from the raw results of the micro-benchmarks.
- Emphasizing the detection of hardware features through variations in performance. BlackjackBench detects the *effective* values of hardware characteristics, which is what a user level application experiences when running on the hardware.

6. REFERENCES

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[2] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You. Accurate Cache and TLB Characterization Using hardware Counters. In *ICCS*, June 2004.

[3] A. X. Duchateau, A. Sidelnik, M. Garzarán, and D. Padua. P-ray: A suite of micro-benchmarks for multi-core architectures. In *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, volume 5335 of LNCS, pages 187–201, Edmonton, Canada, 2008.

[4] J. Gonzalez-Dominguez, G.L. Taboada, B.B. Fraguela, M.J. Martin, and J. Tourio. Servet: A Benchmark Suite for Autotuning on Multicore Clusters. In *IPDPS*, 2010.

[5] L. J. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: Identification and analysis of coexpressed genes. *Genome Research*, 9(11):1106–1115, 1999.

[6] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *ATEC'96: USENIX 1996*, pages 23–23, Berkeley, CA, 1996.

[7] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *PACT '09*, pages 261–270, Washington, DC, USA, 2009.

[8] P. Mucci and K. London. The CacheBench Report. Technical report, University of Tennessee Knoxville, 1998.

[9] R. Saavedra and A. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Trans. Computers*, 44(10):1223–1235, 1995.

[10] C. Staelin and L. McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX 1998*, pages 155–166, New Orleans, LA, January 15-18 1998.

[11] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software: Practice and Experience*, 38(15):1621–1642, April 2008.

[12] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[13] K. Yotov, S. Jackson, T. Steele, K. Pingali, and P. Stodghill. Automatic measurement of instruction cache capacity. In *Proceedings of the 18th LCPC Workshop*, 2005.

[14] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.